

Machine-checked Cut-elimination for Display Logic

Jeremy E. Dawson* and Rajeev Goré†

Department of Computer Science and Automated Reasoning Group

Australian National University

Canberra ACT 0200, Australia

jeremy@arp.anu.edu.au rpg@arp.anu.edu.au

7 November 2006

Abstract

Belnap’s Display Logic is a generalised sequent-style framework which is able to capture many different logics in one uniform setting. Its main attractions are twofold: a “display property” which allows every formula to be introduced as the whole of the left or right side of a sequent, and a single cut-elimination theorem which works for all “proper” display calculi. Belnap’s original proof of this theorem is short and succinct, but it has been criticised by some as being too informal. The only other published proof turns out to contain an error.

Modern interactive proof assistants like Nqthm, HOL, Isabelle, Coq and PVS invariably trace their origins to the simply-typed lambda-calculus of Church (although Mizar is an exception). Such tools have matured to the point where deep theorems in logic and mathematics like Gödel’s Incompleteness Theorem and the Four Colour Theorem can be expressed and proved by expert users using these systems.

Here, we present a new proof of cut-elimination for all display calculi which does not rely on the usual double induction on the grade and weight of a cut. Simultaneously, we present a high-level description of how we used the proof assistant Isabelle to find and check this proof. We believe that the use of such proof assistants will soon become common-place as proofs become more and more complicated.

*Supported by an Australian Research Council Large Grant

†Supported by an Australian Research Council QEII Fellowship

Contents

1	Introduction	3
2	Logical Frameworks and Interactive Proof Assistants	6
3	Related Work on Mechanised Proof Theory	7
4	Relation Algebras, $\delta\mathbf{RA}$, and Isabelle/HOL	9
4.1	Relation Algebras	9
4.2	The Display Calculus $\delta\mathbf{RA}$	9
4.3	Isabelle and its HOL theory	10
4.4	Deep and Shallow Embeddings	11
5	A Deep Embedding of $\delta\mathbf{RA}$ in Isabelle/HOL	11
5.1	Representing Formulae, Structures, Sequents and Rules	11
5.2	Representing Derivations as Trees	14
5.3	Handling Substitutions Explicitly	15
5.4	Reasoning About Derivations	16
5.5	Reasoning About Derivability	18
6	An Operational View of Cut-Elimination	20
6.1	Principal Cuts and Belnap’s Condition C8	21
6.2	Transforming Arbitrary Cuts Into Principal Ones	22
7	A Direct Proof of Cut-Admissibility (Weak-Normalisation)	24
7.1	Functions for Reasoning About Cuts	24
7.2	Dealing With Principal Cuts	24
7.3	Making A Cut (Left) Principal	26
7.4	Turning One Cut Into Several Left-Principal Cuts	28
7.5	Putting It All Together	29
7.6	Tracing Belnap’s Conditions of the Display Calculus	34
8	Wansing’s Strong Normalisation Proof	35
9	Discussion	38
10	A New Proof of Strong Normalisation	38
10.1	Preliminary Definitions and Results for Strong Normalization	38
10.2	Defining Strongly Normalising Derivations	39
10.3	Various Well-Founded Orderings	40
10.4	Inheriting Strong Normalisation from Immediate Subtrees	43
10.5	Reasoning About Cut-Reductions	45
10.6	Strong-Normalisation	47
10.7	Making A Cut (Left) Principal	48
10.8	Dealing With Principal Cuts	54
10.9	Cut-Elimination Via Strong Normalisation	55
11	Conclusion and Further Work	58
12	Appendix: Kracht’s Formulation of Belnap’s Conditions	60

1 Introduction

Sequent calculi [37] provide a rigorous basis for meta-theoretic studies of logics. The central theorem is cut-elimination, which states that detours through lemmata can be avoided, and allows us to prove many important meta-theoretic properties like consistency, interpolation, and Beth definability in a straightforward way [38]. Cut-free sequent calculi are also useful for automated deduction [19], nonclassical extensions of logic programming [35], and studying the deep connections between cut elimination, lambda-calculi and functional programming [20]. Sequent calculi, and their extensions, therefore play an important role in both logic and theoretical computer science.

Display Logic [5] is a generalised sequent framework for non-classical logics, based on Gentzen’s sequent calculus [37]. Since it is not really a logic, we prefer the term “display calculi” and use it from now on. Display calculi extend Gentzen’s language of sequents with extra, complex, n -ary structural connectives, in addition to Gentzen’s sole structural connective, the “comma”. Whereas Gentzen’s comma is usually assumed to be associative, commutative and inherently poly-valent, no such implicit assumptions are made about the n -ary structural connectives in display calculi. Properties such as associativity are explicitly stated as structural rules.

Such explicit structural rules make display calculi more modular than traditional sequent-style calculi: the logical rules remain constant and different logics are obtained by the addition or deletion of structural rules only. Display calculi therefore provide an extremely elegant sequent framework for “logic engineering”, encompassing a vast range of substructural logics like the bi-intuitionistic and bi-classical versions of the Lambek calculus, linear logic, relevant logic, BCK-logic, and their modal and tense logical extensions, in a uniform way [40, 17]. The most remarkable property of display calculi is a generic cut-elimination theorem, which applies whenever the rules for the display calculus satisfy certain, easily checked, conditions [5]. The resulting cut-elimination theorem is not only modular, but is robust under the addition or deletion of a vast array of purely structural rules [21].

Many traditional proofs of cut-elimination in sequent calculi do not eliminate the cut rule directly, due to problems in eliminating applications of contraction above cut. Gentzen [37] first replaced the cut rule with the mix rule, showed that cut was derivable from mix, and then eliminated the mix rule. Borisavljevic *et al* [6] have shown that direct cut-elimination is by no means trivial, requiring a further detour through a special normal form for derivations. The problems caused by contraction can also be avoided by building the effect of the contraction rule into the other logical rules, thereby giving “contraction-free” calculi [12, 31]. Then, all sequents obtained by an application of contraction can be replaced by their “uncontracted” counterparts [38]. We cannot use this strategy as the ability to include or omit

explicit structural rules like contraction give display calculi, and the associated cut-elimination theorems, their modularity.

In [5], Nuel Belnap gives a direct proof that the cut rule is *admissible* in display calculi: he transforms a derivation whose only instance of cut is at the bottom, into a cut-free derivation of the same end-sequent. Given a derivation with multiple cuts, such a procedure repeatedly eliminates the top-most cut, and this procedure is sometimes known as weak-normalisation. A much harder task is to give an effective procedure to eliminate an arbitrary cut rather than just the top-most one, and such procedures are sometimes known as strong-normalisation.

Although Belnap’s proof of weak-normalisation uses a double induction, he does not proceed via a mix rule, and he does not set out the proof in the traditional way involving a cut rank and cut degree à la Gentzen [37]. For this reason, Belnap’s proof has been criticised verbally by well-known proof theorists as lacking rigour, although never in writing to our knowledge. While our proof of weak normalisation is not set out to mimic Belnap’s, it seems basically similar, does not use a mix rule in place of cut, and does not require any special attention be given to the contraction rule. To settle the question, Heinrich Wansing gave a direct proof of strong-normalisation for the cut-elimination procedure for a particular display calculus in [39]. As is traditional in proofs of strong-normalisation, Wansing first defines certain transformations on derivations. He then shows that a sufficiently long sequence of such transformations, beginning with some given derivation, will terminate with a cut-free derivation of the same end-sequent. The original derivation may contain multiple instances of cut, but only certain instances of cut are amenable to the transformations. (In effect, the transformations disallowed are those where one cut is “moved” past another). The highest cuts, with no cuts above them, are always amenable to these transformations, so strong-normalisation implies the admissibility of cut (weak-normalisation).

Logical frameworks are computer programs which act as interactive proof-assistants [34]. A user can encode a formal system using a special syntax, and then reason interactively about the encoded formal system using in-built facilities for correctly handling tedious bookkeeping. The core of a logical framework typically consists of only a few hundred lines of code, providing primitive operations from which all other features and facilities are defined. The small size and public nature of the core implementation provides high confidence in, but not a guarantee of, its correctness. Proofs formalised using a logical framework are usually more detailed than pencil-and-paper proofs since every step is eventually compiled down to the primitive operations provided by the underlying core. Many logical frameworks also provide facilities for automated proof-search.

We found it difficult to understand Wansing’s proof in its entirety so we decided to formalise and check it for the display calculus $\delta\mathbf{RA}$ for relation

algebras from [15] using a logical framework. The choice of the display calculus is not important, we just happen to be very familiar with it. In [9] we outlined our initial attempts and described why we finally settled on the logical framework Isabelle/HOL. But in formalising Wansing’s proof of strong-normalisation, we found a slight anomaly: at a certain point in the proof, Wansing states that “This construction generalizes easily to the case where A is introduced . . . at more than one point . . .” [39, page ?]. We were unable to see how the generalisation would proceed.

We therefore developed our own proof of strong-normalisation for the display calculus $\delta\mathbf{RA}$. The proof is novel because it does not use the usual double-induction on the size of the cut-formula and the grade of the cut instance, but proceeds instead by showing that an appropriate combination of four complicated orderings on the transformed derivations is well-founded. Our orderings do not depend upon the size of derivations. Since strong-normalisation implies weak-normalisation, we automatically obtain a formalisation of weak-normalisation. As a separate exercise, we also formalised a more direct proof of weak-normalisation which proceeds by induction on the structure of the original derivation.

Here, we describe how we proved both the admissibility of cut (weak-normalisation) and the strong-normalisation result for $\delta\mathbf{RA}$. As far as we are aware, this is the first full formalisation of weak-normalisation in the presence of explicit structural rules using a logical framework, and the first full formalisation of strong-normalisation of cut-elimination for a sequent calculus using a logical framework. The work required approximately 9 man-months, required approximately 1800 lines of Isabelle/HOL theory files and 8500 lines of ML proof files, and takes approximately 35 minutes to fully run the proofs on a 300MHz Pentium machine.

This paper is a longer version of [11] and [10], and is set out as follows. In section 4 we describe relation algebras, the display calculus $\delta\mathbf{RA}$ for the logic of relation algebras, the basics of the logical framework Isabelle and its subtheory HOL, and related work on mechanised proof theory. In Section 5 we describe an encoding of $\delta\mathbf{RA}$ logical constants, logical connectives, formulae, structures, sequents, rules, derivations and derived rules in Isabelle/HOL. In Section 6 we describe the two main transformations required to eliminate cut and describe Isabelle/HOL functions that allow us to reason about cuts in Isabelle/HOL. In Section 7.1 we describe how we mechanised these to prove the weak normalisation theorem for $\delta\mathbf{RA}$ in Isabelle/HOL. In Section 8 we discuss the problems we had in formalising Wansing’s proof of strong normalisation. In Section 10 we describe our own proof of strong normalisation for the display calculus $\delta\mathbf{RA}$. In Section 11 we present conclusions and discuss further work.

The Isabelle code can be found at <http://csl.anu.edu.au/~jeremy/isabelle/fdeep/>.

Acknowledgements: This work was supported by the Australian Research Council via a Queen Elizabeth II Fellowship and a Large Research Grant.

2 Logical Frameworks and Interactive Proof Assistants

We now describe the architecture of typical interactive theorem provers using plain English. The description is necessarily imprecise and simplistic in many places, so we ask expert readers to bear with any infelicities.

Modern computer-based tools for interactive theorem proving like HOL, Isabelle, Coq and PVS trace their origins back to the simply-typed lambda calculus of Church and the Curry-Howard Isomorphism between normal λ -terms and normal proofs in intuitionistic logic. Others like Mizar do not use the λ -calculus but are based upon (essentially) first-order set theory. Nevertheless, it is fair to say that they are all based upon the following general principles in which we describe the layers that lie between a user and the actual computer:

Machine Level: At the lowest level is a traditional computer which runs some flavour of machine code, typically using an assembly language. We must trust that this machine runs the assembly language correctly since we have no hope of verifying that it does so.

Modern Programming Language and its Compiler: Next comes some flavour of modern computer programming language like ML or Pascal, and a compiler or interpreter which allows us to translate programs written in this programming language into machine code. Although there is much work on creating certifying compilers, there is currently little hope of constructing a verified compiler for such high level languages.

A Core Engine: Next comes a typically small amount of code written in the modern programming language which implements the basic operations of the logical heart of the interactive prover. In the case of systems based upon the λ -calculus this is an implementation capturing the syntax and core operations of the λ -calculus like β -reduction, η -reduction, and substitution. For systems like Mizar it is a Pascal implementation of the basic operations of the chosen set theory. This code is highly unlikely to be verified, but it is so small that open scrutiny by the community is likely to afford us high confidence that it correctly implements the basic operations of the formal system (λ -calculus or set theory). For example, the core of Isabelle consists of about 1400 lines of ML code. This level also has to provide a collection

of basic functions which allow a user to use this underlying core engine as a very high level programming language.

Library of Theories: Next, there is typically a library of modules (usually called theories) written using the basic functions provided by the core engine, each of which encodes some particular high-level mathematical notion. In Isabelle, even the logic used is found at this level, so the user can choose between Intuitionistic or Classical First-Order Logic, Higher-Order Logic, among others. Then further theories provide sets, lists, numbers of various forms, etc. For example, most interactive theorem provers provide an encoding of the natural numbers, allowing the user to use “0”, “1”, etc to stand for the natural numbers 0, 1 etc and to use “+” as the usual operation of addition on natural numbers. A user wanting to use this module would typically declare this at the beginning of his or her own code, which causes the core engine to load this library.

The User: Finally, there is the user who typically programs his or her own notions, either using the libraries provided, or directly by using the functions provided by the core engine.

3 Related Work on Mechanised Proof Theory

We are aware of only four other attempts to formalise proof theory, weak normalisation or strong normalisation for sequent or natural-deduction calculi. In this section we outline the major differences between this existing work and our work.

Various authors have fully formalised proofs of strong-normalisation for λ -calculi, and hence the natural-deduction calculi associated with these λ -calculi: see references in [2, 3]. But strong- or weak-normalisation in natural-deduction calculi removes only consecutive applications of the introduction and elimination rules for $\varphi \rightarrow \psi$. Although this corresponds to the removal of principal cuts in sequent calculi, there is no need to remove other applications of cut (modus ponens). Consequently, weak- or strong-normalisation results in lambda-calculi are not immediately applicable to sequent calculi. Moreover, λ -calculi are typically restricted to intuitionistic logics, and their corresponding natural-deduction calculi are usually single conclusioned. Sequent calculi, and particularly display calculi, are a much more general framework, catering to both intuitionistic and classical variants of many substructural logics and their modal and tense extensions [17]. The strong-normalisation results for sequent calculi are therefore more widely applicable. Their corresponding sequent calculi are usually “contraction-free”: that is, all structural rules are built into the other rules.

The first attempt to formalise sequent calculi appears to be the work of Pfenning where he formalises the admissibility of cut for classical, intuitionistic and linear logic using the logical framework Elf, which itself is based upon the dependent type theory LF [31, 33, 32]. Elf is based upon logic programming and uses a Prolog-like backtracking procedure for executing the encoded definitions. When initiating our work, we compared various ways to encode $\delta\mathbf{RA}$ and reported the results in [9]. We found, for example, that the Elf implementation reports success even when certain transformations, essential to cut-elimination, are deleted from the encoding! Our conclusion was that the Elf implementation was essentially a program for cut-elimination, which returns a cut-free derivation when given a derivation containing cuts, rather than a full formalisation of cut-elimination. Pfenning essentially says the same thing in the conclusion of [31] where he states:

Once the structural proof of admissibility has been found and implemented, it is natural to ask if it can also be encoded in stronger frameworks such as Coq [DFH+93] so that structural inductions are made explicit and the proof is fully formally verified.

Pfenning’s proofs of weak-normalisation have now been fully formally verified in Twelf, a successor to Elf, by Schürmann [36, Section 8.5]. But the calculi used by both Pfenning and Schürmann contain no explicit structural rules since these have been built into the other rules of the calculi. It is well-known that structural rules like contraction are usually the bane of cut-elimination, so it is not at all obvious how to extend the work of Pfenning and Schürmann to calculi with explicit structural rules. The associated *weak-normalisation* results are therefore not as modular as in display calculi since omitting one or more structural rules is then not possible (trivially).

The second is the work of Matthews *et al* [24, 4, 25, 23] which implements Feferman’s FS_0 in Isabelle, and uses it to formalise and extend various meta-theoretical systems. It should be possible to formalise weak-normalisation for cut-elimination in this implementation since Matthews outlines how it might be done using “pencil and paper”, and suggests it as a further project [23, §6,§9.2]: but it does not appear to have been done.

The third is the work of Mikhajlova and von Wright [26]. This is the closest to our work since it provides a formalisation of a sequent calculus for first-order logic in the system HOL, but this work makes no attempt to prove a cut-elimination or strong-normalisation result, and as we point out later, actually contains an unfortunate bug.

The fourth is the work of Adams [1] in which he formalises three cut-free sequent-style calculi in Coq, and “proves” cut-admissibility (weak-normalisation) for derivations with respect to certain permutations. We write “proves” because Adams actually proves three lemmata which jointly imply weak-normalisation, but does not formalise the statement of weak-normalisation itself since his encoding is unable to express this. Here too,

the calculi have no explicit structural rules since these are built into the other rules. Consequently it is not at all obvious how to “go substructural” in this framework.

4 Relation Algebras, $\delta\mathbf{RA}$, and Isabelle/HOL

In this section we introduce the display calculus for relation algebras $\delta\mathbf{RA}$, and describe the basics of the logical framework called Isabelle/HOL.

4.1 Relation Algebras

Relation algebras [22] are extensions of Boolean algebras; whereas Boolean algebras model subsets of a given set, relation algebras model binary relations on a given set. Relation algebras have operations such as relational composition and relational converse, and Boolean operations such as intersection (conjunction) and complement (negation). Relation algebras form the basis of relational databases [13] and of the specification and proof of correctness of programs, particularly in the style of Mili [27]. The following grammar defines the syntax of relation algebras where each atomic formula p_i denotes a binary relation on some underlying set, and where the other elements of the top line are Boolean and the elements of the bottom line are their relational analogues:

$$A ::= p_i \mid \top \mid \perp \mid \neg A \mid A \wedge A \mid A \vee A \\ \mid \mathbf{1} \mid \mathbf{0} \mid \smile A \mid A \circ A \mid A + A$$

4.2 The Display Calculus $\delta\mathbf{RA}$

A display calculus for relation algebras called $\delta\mathbf{RA}$ can be found in [15]. Sequents of $\delta\mathbf{RA}$ are expressions of the form $X \vdash Y$ where X and Y are built from the nullary structural constants E and I and formulae, using a binary comma, a binary semicolon, a unary $*$ or a unary \bullet as structural connectives, according to the grammar below:

$$X ::= A \mid I \mid E \mid *X \mid \bullet X \mid X ; X \mid X , X$$

Thus, whereas Gentzen’s sequents $\Gamma \vdash \Delta$ assume that Γ and Δ are comma-separated lists of formulae, $\delta\mathbf{RA}$ -sequents $X \vdash Y$ assume that X and Y are complex tree-like structures built from formulae and the constants I and E using comma, semicolon, $*$ and \bullet .

The defining feature of display calculi is that in all logical introduction rules, the principal formula is always “displayed” as the whole of the right-hand or left-hand side. For example, the rule (**LK**- $\vdash \vee$), shown below left, is typical of Gentzen’s sequent calculi like **LK**, while the rule ($\delta\mathbf{RA}$ - $\vdash \vee$)

shown on the right is typical of display calculi:

$$\frac{\Gamma \vdash \Delta, P, Q}{\Gamma \vdash \Delta, P \vee Q}(\mathbf{LK}\text{-}\vdash \vee) \qquad \frac{X \vdash P, Q}{X \vdash P \vee Q}(\mathbf{\delta RA}\text{-}\vdash \vee)$$

To use this display calculus rule downwards on a sequent $X' \vdash Y'$ containing an occurrence of the structure (P, Q) , everything other than (P, Q) must be moved to the left of \vdash creating the complex structure X , thereby displaying the structure (P, Q) as the whole of the right-hand side. There are (reversible) rules which enable any given structure to be displayed. After the rule application we can “undisplay” the moved material back to its original position (reversing the display steps used), so that the sole purpose of these moves is to “rewrite” some occurrence of (P, Q) in X' or Y' to $P \vee Q$. Note that the occurrence of (P, Q) could come from X' or from Y' : see [15] for a full account.

4.3 Isabelle and its HOL theory

Isabelle is a logical framework developed by Lawrence Paulson at the University of Cambridge, England [29]. Its logical core consists of approximately 1400 lines of ML code, implementing the typed lambda-calculus of Church [7]. The core provides the primitive operations of higher-order unification and term rewriting necessary to implement an intuitionistic typed higher-order logic and typed λ -calculus, called Isabelle’s “meta-logic”.

Isabelle accepts inference rules of the form “from $\alpha_1, \alpha_2, \dots, \alpha_n$, infer β ” where the α_i and β are expressions of the Isabelle meta-logic, or are expressions using a new syntax, defined by the user, for some “object logic” \mathcal{F} . An Isabelle user can encode a particular calculus \mathcal{C}_L for some logic L as an “object logic” by using these rule templates to encode the set of inference rules of \mathcal{C}_L . For example, if \mathcal{C}_L is a natural deduction calculus, then the α_i and β will be formulae of L , whereas if \mathcal{C}_L is a sequent calculus, then the α_i and β will be sequents of \mathcal{C}_L . Such an encoding is called an “object logic”, even though it is a (typically natural deduction or sequent) *calculus* for some particular logic L . In practice most users would build on one of the comprehensive “object logics” already supplied [30], such as Isabelle/HOL [28].

Isabelle/HOL is an Isabelle theory encoding the higher-order logic of Church [7] and is based upon the HOL system of Gordon [14]. Thus it includes keywords for quantification and abstraction over higher-order functions and predicates. The HOL theory uses Isabelle’s own type system and function application and abstraction: that is, object-level types and functions are identified with meta-level types and functions. Isabelle/HOL also contains constructs found in functional programming languages, such as `datatype` and `let`, which enable a user to re-implement a program in Isabelle/HOL, and then reason about it using high order logic. However

limitations (not found in, say, Standard ML itself) prevent defining types which are empty or which are not sets, or functions which may not terminate.

Rather than giving more details now, we shall explain these keywords and constructs as they appear in our explanations in the sequel.

4.4 Deep and Shallow Embeddings

When encoding any aspect of a logic, we must encode the basic syntactic expressions like atomic formulae, logical connectives, and formulae. But after these have been encoded, the facilities provided by the logical framework can often be used to encode the higher level syntactic expressions like the sequent arrow, the syntax of sequent rules, the instantiations required to form derivations from the rules, and even the derivations themselves. The term “shallow” embedding is widely-used to describe an encoding where the facilities provided by the logical framework are used as proxies to a greater degree, while the term “deep” embedding is used to describe an encoding where every syntactic concept is modelled explicitly by a new construct not provided by the logical framework.

The interactive theorem proving community typically uses the term “formalise” instead of “encode”, so we use these terms interchangeably.

5 A Deep Embedding of δ RA in Isabelle/HOL

In [9], we describe our initial attempts to formalise display calculi in various logical frameworks, and describe why we chose Isabelle/HOL for this work. In this section, we describe the Isabelle/HOL data structures used to represent formulae, structures, sequents and derivations. Although we have tried to make the paper self-contained, this section requires some basic familiarity with ML and logical frameworks in general. But it can be skipped entirely by readers not interested in the details of our encoding.

5.1 Representing Formulae, Structures, Sequents and Rules

An actual derivation in a Display Calculus involves formulae which are composed of primitive propositions, which we typically write as p, q, r . It uses rules which are *expressed* using structure variables like X, Y, Z and formula variables like A, B, C to represent structures and formulae made up from primitive propositions. Nonetheless, in deriving theorems or derived rules we often use more specific instances of a rule, obtained by replacing the variables of the rule with expressions involving other variables. For example, we may take the cut rule as shown below left and substitute $B \wedge C$ for A , substitute Z, D for X and substitute $C \vee D$ for Y to obtain the cut rule

```

datatype formula = Btimes formula formula ("_ && _" [68,68] 67)
                | Rtimes formula formula ("_ oo _" [68,68] 67)
                | Bplus formula formula ("_ v _" [64,64] 63)
                | Rplus formula formula ("_ ++ _" [64,64] 63)
                | Bneg formula ("--_" [70] 70)
                | Rneg formula ("_ ^" [75] 75)
                | Btrue ("T")
                | Bfalse("F")
                | Rtrue ("r1")
                | Rfalse("r0")
                | FV string
                | PP string

```

Figure 1: Isabelle/HOL Representation of Formulae of $\delta\mathbf{RA}$

```

datatype structr = Comma structr structr
                | SemiC structr structr
                | Star structr
                | Blob structr
                | I
                | E
                | Structform formula
                | SV string

```

Figure 2: Isabelle/HOL Representation of Structures of $\delta\mathbf{RA}$

(instance) shown below right, about which we have to reason.

$$(\text{cut}) \quad \frac{X \vdash A \quad A \vdash Y}{X \vdash Y} \quad \frac{Z, D \vdash B \wedge C \quad B \wedge C \vdash C \vee D}{Z, D \vdash C \vee D}$$

Our Isabelle formulation must allow this since variables such as X, Y, Z and A, B, C are not part of the language of a Display Calculus, but are part of the meta-language used when reasoning *about* Display Calculi.

Formulae of $\delta\mathbf{RA}$ are therefore represented by the datatype declaration shown in Figure 1. There is one constructor for each of the relational connectives, one constructor for each of the Boolean connectives, and one constructor each for the logical constants which make up the syntax of $\delta\mathbf{RA}$ as described in Section 4.1. The constructor FV indicates that the string following FV is the name of a formula variable like A which might appear in the statement of a rule or theorem, and which is instantiated to actual formulae of $\delta\mathbf{RA}$ when constructing derivations. The constructor PP represents a primitive proposition variable p for which we can substitute only another primitive proposition.

For example, the formula $(p \wedge q) + r$ is represented as the Isabelle term `Rplus (Btimes (PP ''p'') (PP ''q'')) (PP ''r'')` while the formula schema $A \wedge B$ is represented by the Isabelle term `Btimes FV ''A'' FV ''B''` where strings are enclosed in quotes according to the syntax of Isabelle. The strange notation to the right of the first six lines of Figure 1 allows us to use the corresponding symbols like `&&` as infix versions of the longer operator names. So, $(p \wedge q) + r$ can also be represented as `(PP ''p'' && PP ''q'') +++ PP ''r''`.

Structures of $\delta\mathbf{RA}$ are represented by the datatype declaration shown in Figure 2. Once again, there is an explicit construct for every part of the syntax of display logic structures as defined in Section 4.2. The operator `Structform` causes a formula to be “cast” into a structure. The constructor `SV` represents structure variables which appear in the statement of a rule or theorem, and which are instantiated to actual formulae of the calculus when constructing derivations.

For example, the structure $(p \wedge q) + r$ is represented as the Isabelle term `Structform (Rplus (Btimes (PP ''p'') (PP ''q'')) (PP ''r''))` while the structure $(*X) ; B$ is represented by the Isabelle term `SemiC (Star (SV ''X'')) (SV ''Y'')`.

Some complex manipulation of the syntax, available through defining “parse translations” and “print translations” of a sequent, allows structure variables and constants to be prefixed by the symbol `$`, and the notations `FV`, `SV` and `Structform` to be omitted. For technical reasons related to this, a different method was used to specify the alternative infix syntax for structures, details of which we have omitted. So, for example, the sequent $A \vdash (P \wedge Q) + R$ can also be represented as the Isabelle term `A |- (''P'' && ''Q'') +++ ''R''` while the sequent $Z \vdash (*X) ; Y$ can also be represented by the Isabelle term `$Z |- (* $''X''); $''Y''`.

In contrast to the formula and structure variables modelled explicitly by `FV` and `SV`, Isabelle has “scheme variables”, identified by a preceding ‘?’’. When a scheme variable appears in an Isabelle theorem, it means that the theorem is true when anything of the appropriate type is substituted for the scheme variable. Isabelle also has “free variables”, which represent unspecified but fixed values of the appropriate type. Note that in the expressions above, `A` and `Z` are Isabelle free variables, of type `formula` and `structr` respectively.

A sequent (`Sequent X Y`) can also be represented as `$X |- $Y`. Thus the term `Sequent (SV ''X'') (Structform (FV ''A''))` is printed, and may be entered, as `($''X'' |- ''A'')`.

We say that a structure expression is *formula-free* if it does not contain any formula as a sub-structure: that is, if it does not contain any occurrence of the operator `Structform`. A formula-free sequent is defined similarly.

Sequents and rules of $\delta\mathbf{RA}$ are represented by the Isabelle/HOL datatype below:

```
datatype sequent = Sequent structr structr
```

```
datatype rule = Rule (sequent list) sequent
```

The premises of a rule are represented using a list of sequents while the conclusion is a single sequent. Thus `Rule prems concl` means a rule with premises `prems` and conclusion `concl`. Many single-premise rules of display calculi are defined to be usable from top to bottom as well as from bottom to top: we provide a function `invert` to cater for these. Special functions `premsRule` and `conclRule` return the list of premises and the conclusion of a rule respectively.

The cut rule is common to all display calculi, and is called `cutr`. The constant `rls` represents the set of rules of $\delta\mathbf{RA}$. For example, encoded using the datatypes just described, the $\delta\mathbf{RA}$ rule ($\delta\mathbf{RA}\text{-}\vdash\vee$) shown in §4.2 is represented and printed as the rule `ors`:

```
"ors == Rule [$'X'' |- ''A'',, ''B''] ($'X'' |- ''A'' v ''B'')"
```

The actual rules `rls` of $\delta\mathbf{RA}$ are not relevant to most of this paper, since we prove our results for any set of rules satisfying Belnap’s conditions (see §11). Separately, we prove that the rules `rls` of $\delta\mathbf{RA}$ satisfy Belnap’s conditions.

We represent rules explicitly using a datatype `rule` and use explicit constructors `SV` and `FV` for structure and formula variables because we need to express and check that the rules satisfy (some of) Belnap’s conditions. For example, Belnap’s condition (C3) amounts to “a structure variable cannot appear more than once in the conclusion of a rule”. This condition could not be checked, in Isabelle itself, if we had installed the $\delta\mathbf{RA}$ rules using Isabelle’s rule, variable and substitution features. We therefore also had to handle substitutions for our `SV` and `FV` variables explicitly, via various functions. Although this approach made our task more difficult, it was necessary so that we could structure the proofs of both weak [11] and strong normalisation in terms of Belnap’s conditions. Such aspects of “deep” versus “shallow” embeddings are also discussed in [9].

5.2 Representing Derivations as Trees

We use the term “derivation” for a proof *within* the sequent calculus, reserving the term “proof” for a meta-theoretic proof of a theorem *about* the sequent calculus. We model a derivation tree (type `dertree`) using the following datatype:

```
datatype dertree = Der sequent rule (dertree list)
                | Unf sequent
```

In `Der seq rule dts` the subterm `seq` is the sequent at the root (bottom) of the tree, and `rule` is the rule used in the last (bottom) inference.

If the tree represents a real derivation, sequent `seq` will be an instance of the conclusion of `rule`, and the corresponding instances of the premises of `rule` will be the roots of the trees in the list `dts`. We then say that the root “node” of such a tree is *well-formed*. The trees in `dts` are the *immediate* subtrees of `Der seq rule dts`.

The leaves of a derivation tree are either axioms with no premises, or “Unfinished” sequents whose derivations are currently unfinished. The derivation tree for a derivable sequent will therefore have no `Unf` leaves and we call such a derivation tree *finished*. The derivation tree for a derived rule will have the premises of the rule as its `Unf` leaf sequents.

Display calculi use the initial sequent $p \vdash p$, using primitive propositions only. It is then proved that the sequent $A \vdash A$ is derivable, without using the cut rule, for all formulae A by induction on the size of A , where A *stands for* a formula composed of primitive propositions and logical connectives. We proved this for $\delta\mathbf{RA}$ as the theorem `idfpp` and therefore added a rule in Isabelle called `idf` with shape $A \vdash A$ to correspond to the derived rule of $\delta\mathbf{RA}$. Note that we can not prove this for a general display calculus (it does not follow from Belnap’s conditions but instead depends upon the actual rules).

We also need to reason about the derivation trees of derived rules which may use the (derived) rule $A \vdash A$, for arbitrary formula A . Therefore the derivation tree `Der (‘A’ |- ‘A’) idf []` stands for a finished derivation which uses the lemma that $A \vdash A$ is derivable, while `Unf (‘A’ |- ‘A’)` stands for an unfinished derivation with unfinished premiss $A \vdash A$.

For example, the unfinished derivation tree shown below at left is represented as the Isabelle/HOL term shown below at right where `‘A’ |- PP p && ‘A’` stands for $A \vdash p \wedge A$, `cA` and `ands` are the contraction and $(\vdash \wedge)$ rules, and `idf` is the derived rule $A \vdash A$:

$$\frac{A \vdash p \quad A \vdash A}{A, A \vdash p \wedge A} (\vdash \wedge) \quad \frac{}{A \vdash p \wedge A} (ctr) \quad \text{Der (‘A’ |- PP p \&\& ‘A’) cA} \quad \text{[Der (‘A’, ‘A’ |- PP p \&\& ‘A’) ands} \\ \text{[Unf (‘A’ |- PP p),} \\ \text{Der (‘A’ |- ‘A’) idf []]]}$$

5.3 Handling Substitutions Explicitly

Since a “deep” embedding models formula and structure variables explicitly, we cannot use the unification and substitution mechanisms provided by the logical framework. We therefore give definitions relating to substitution for structure and formula variables, which we require for handling substitution explicitly. In Figure 3 we first give some type abbreviations, and then the types of a sample of the functions.

Thus `sFind` is a function of two curried arguments, of types `sSubst` and `string`, and result type `structr`, and `sSubst` is the type of lists of `(string,`

```

types

fSubst = "(string * formula) list"
sSubst = "(string * structr) list"
fsSubst = "fSubst * sSubst"

consts

fFind      :: "fSubst => string => formula"
sFind      :: "sSubst => string => structr"

subDT      :: "fsSubst => dertree => dertree"
ruleSubst  :: "fsSubst => rule => rule"
seqSubst   :: "fsSubst => sequent => sequent"
strSubst   :: "fsSubst => structr => structr"
fmlSubst   :: "fSubst => formula => formula"

```

Figure 3: Functions for handling substitution explicitly

`structr`) pairs. To substitute for a variable, for example `SV 'X'`, in some object, using the substitution `(fsubs, ssubs)`, we use `sFind` to obtain the first pair in `ssubs` whose first component is `'X'`. If that pair is `('X', Y)`, then `sFind` returns `Y`, and each occurrence of `SV 'X'` in the given object is replaced by `Y`. The functions given substitute for every formula or structure variable in the derivation tree, rule, sequent, structure or formula.

5.4 Reasoning About Derivations

In this section we describe various functions which allow us to reason about derivations in $\delta\mathbf{RA}$. The types for these functions are shown in Figure 4.

`allDT f dt` holds if property `f` holds for every sub-tree in the derivation tree `dt`.

`allNextDTs f dt` holds if property `f` holds for every proper sub-tree of `dt`.

`wfb (Der concl rule dts)` holds if sequent rule `rule` has an instantiation with conclusion instance `concl` and premise instances which are the roots of the derivation trees in the list `dts`. In this case, the node (the bottom node of the derivation tree) is said to be *well-formed*.

`allDT wfb dt` holds if every sub-tree of the derivation tree `dt` is *well-formed*. That is, if every node in `dt` is well-formed. Such a derivation is said to be *well-formed*.

```

allDT      :: "(dertree => bool) => dertree => bool"
allNextDTs :: "(dertree => bool) => dertree => bool"
wfb        :: "dertree => bool"
frb        :: "rule set => dertree => bool"
premsDT    :: "dertree => sequent list"
conclDT    :: "dertree => sequent"
IsDerivable :: "rule set => rule => bool"
IsDerivableR :: "rule set => sequent set => sequent => bool"
IsDerivableB :: "rule set => sequent => sequent => bool"

```

Figure 4: Functions for reasoning about derivations

`frb rules (Der concl rule dts)` holds when the lowest rule `rule` used in a derivation tree `Der concl rule dts` belongs to the set `rules`.

`allDT (frb rules) dt` holds when every rule used in a derivation tree `dt` belongs to the set `rules`.

`premsDT dt` returns a list of all “premises” (unfinished leaves) of the derivation tree `dt`. That is, the sequents found in nodes of `dt` of the form `Unf seq`.

`conclDT dt` returns the end-sequent of the derivation tree `dt`. That is, the conclusion of the bottom-most rule instance.

A tree representing a real derivation in a display calculus naturally is well-formed and uses the rules of the calculus. Further, a tree which derives a sequent (rather than a derived rule) is finished: that is, it has no unfinished leaves.

The cut-elimination procedure involves transformations of trees, and in discussing these, we will only be interested in trees which actually derive a sequent, so we make the following definition.

Definition 5.1 *A derivation tree `dt` is valid (with respect to the rule set `rules`) if it is well-formed, it uses rules in the set of rules `rules` of the calculus, and it has no unfinished leaves.*

```

valid_def = "valid ?rules ?dt ==
  allDT wfb ?dt & allDT (frb ?rules) ?dt & premsDT ?dt = []"

```

Here:

`valid_def = ...` is the name of the Isabelle theorem whose statement is the string in quotes; this theorem holds “by definition”

`valid ?rules ?dt == ...` expresses in Isabelle/HOL that the (higher order) Boolean valued predicate `valid` takes two arguments, and that everything to the right of `==` is the definition of how to evaluate this predicate when given values for these two arguments;

`&` is the conjunction symbol from the higher order logic implemented by the Isabelle/HOL library.

The question marks in front of `rules` and `dt` indicates that they are variables, but question mark is absent from the definition in the Isabelle/HOL theory file itself.

Definition 5.2 (`IsDerivableR`) *IsDerivableR rules prems' concl holds iff there exists a derivation tree `dtr` which uses only rules contained in the set `rules`, is well-formed, has conclusion `concl`, and has premises from set `prems'`.*

```
"IsDerivableR ?rules ?prems' ?concl == (EX dtr.
  allDT (frb ?rules) dtr & allDT wfb dtr &
  conclDT dtr = ?concl & set (premsDT dtr) <= ?prems')"
```

Here, `|` is the disjunction symbol from the higher order logic of Isabelle/HOL, `set` is a function that allows us to treat its argument as a set rather than a list, and `<=` is the subset relation \subseteq .

Finally, `IsDerivable rules rule` holds iff `rule` may be obtained as a derived rule from the (unordered) set `rules`. Thus, `IsDerivable rules (Rule prems concl)` is equivalent to `IsDerivableR rules (set prems) concl`.

5.5 Reasoning About Derivability

Among the results we have proved about the derivability relation are the following theorems.

The first is a transitivity result, relating to a derivation of a conclusion from premises which are themselves derived.

Theorem 5.1 (`IsDerivableR.trans`) *If `concl` is derivable from `prems'`, and each sequent `p` in `prems'` is derivable from `prems`, then `concl` is derivable from `prems`.*

```
IsDerivableR_trans =
  "[| IsDerivableR ?rules ?prems' ?concl ;
    ALL p:?prems'. IsDerivableR ?rules ?prems p |] ==>
    IsDerivableR ?rules ?prems ?concl" : thm
```

Here, the `==>` is the implication of the higher order meta-logic from Isabelle, rather than the implication `-->` of the object logic Isabelle/HOL, and the semicolon separated list delimited by `[|` and `|]` is Isabelle syntax for the antecedents of this implication where the notation `[| P; Q |] ==> R` is syntactic sugar for `P ==> Q ==> R`. Its operands `P`, `Q` and `R` are object level (Isabelle/HOL) boolean expressions which have been recast as Isabelle meta-logic “propositions” by the built-in interface between Isabelle and Isabelle/HOL. We therefore reason in Isabelle’s meta-logic about object-level Isabelle/HOL constructs.

The appellation “: `thm`” indicates a statement that has been proved in Isabelle as a theorem, from previous Isabelle/HOL definitions: we follow this practice for theorems and lemmata throughout this paper.

The second is a different sort of transitivity result, relating to a derivation using rules which are themselves derived.

Theorem 5.2 (`IsDerivableR_deriv`) *If each `rule` in `rules`’ is derivable using `rules`, and `concl` is derivable from `prems` using the set `rules`’, then `concl` is derivable from `prems` using `rules`.*

```
IsDerivableR_deriv = "[| ALL rule:?rules'.
  IsDerivable ?rules rule ; IsDerivableR ?rules' ?prems ?concl |]
  ==> IsDerivableR ?rules ?prems ?concl" : thm
```

We note that in another reported formalization of the notion of derivations in a logical calculus [26], these two properties were, in effect, stated rather than proved. The disadvantage of proceeding that way is the possibility of stating them incorrectly. For example, [26] defines `IsDerivable` inductively as a relation which is transitive in both the senses of the results above; see the second and third clauses of their definition on [26, page 302]. However in the third clause, which deals with the case of a result being provable using derived rules, inappropriate use of an existential quantifier leads to the result that $P \rightarrow Q$ could be used as a derived rule on the grounds that one instance of it, say $True \rightarrow True$, is provable.

Finally we have a recursive characterisation of derivability.

Theorem 5.3 (`IsDerivableR_recur`) *A conclusion `concl` is derivable from premises `prems` using rules `rules` if and only if either `concl` is one of `prems`, or there exists an instance `rule` of a rule in `rules` such that the conclusion of `rule` is `concl` and the premises of `rule` are themselves derivable from `prems` using `rules`.*

```
IsDerivableR_recur = "IsDerivableR ?rules ?prems ?concl = (
  ?concl : ?prems | (EX rule. conclRule rule = ?concl &
  rule : rulefs ?rules & (ALL p : set (premsRule rule).
  IsDerivableR ?rules ?prems p)) )" : thm
```

Here, `rule : rulefs rules` means that `rule` is an instantiation of a rule in `rules`, and `premsRule rule` is the list of premises of the rule `rule`.

Our original proofs of `IsDerivableR_trans`, `IsDerivableR_deriv` and `IsDerivableR_recur` were lengthy and based on the definition of derivability in terms of the existence of a valid derivation tree.

However subsequently developed an abstract treatment of derivation, where we need only specify the required relation between the premises `ps` and conclusion `c` of a single derivation step. (Here the relevant relation is that `ps` and `c` must be the premises and conclusion of a substitution instance of a rule in `rls`: this is written `PC ‘ rulefs rls`). In this abstract treatment, we defined derivability from a list of premises (equivalent to the exact list of premises of a derivation tree) and from a set of premises (which might contain superfluous premises), and proved theorems relating these.

With this framework, it was quite easy to prove these theorems for the abstract context, where `derrec psr ps` is the set of things derivable from `ps` using rules `psr`.

```
derrec_trans =
  "[| ?prems' <= derrec ?psr ?prems; ?concl : derrec ?psr ?prems' |]
   ==> ?concl : derrec ?psr ?prems" : thm
derrec_deriv =
  "[| ?rules' <= gderl ?rules; ?concl : derrec ?rules' ?prems |]
   ==> ?concl : derrec ?rules ?prems" : thm
drs.unfoldr' = "derrec ?psr ?prems == {y. y : ?prems |
  (EX ps. (ps, y) : ?psr & set ps <= derrec ?psr ?prems)}"
```

We defined `derivableR` and proved its relationship to `IsDerivableR`.

```
"derivableR ?rules == derrec (PC ‘ rulefs ?rules)"
IsDer_derR =
  "IsDerivableR ?rules ?prems ?concl = (?concl : derivableR ?rules ?prems)"
```

which enabled us to use `derrec_trans`, `derrec_deriv` and `drs.unfoldr'` to derive `IsDerivableR_trans`, `IsDerivableR_deriv` and `IsDerivableR_recur`.

6 An Operational View of Cut-Elimination

In this section we give an operational view of cut-elimination, to give some intuition of what is involved in the overall cut-elimination procedure a la Belnap [5]. We assume familiarity with notions like “parametric ancestors” of a cut formula from [5].

$$\frac{\frac{\frac{\Pi_{ZAB}}{Z \vdash A, B}}{Z \vdash A \vee B} (\vdash \vee) \quad \frac{\frac{\frac{\Pi_{AX}}{A \vdash X} \quad \frac{\Pi_{BY}}{B \vdash Y}}{A \vee B \vdash X, Y} (\vee \vdash)}{Z \vdash X, Y} (cut)}{Z \vdash X, Y} (cut)}$$

(a)

$$\frac{\frac{\frac{\Pi_{ZAB}}{Z \vdash A, B}}{*A, Z \vdash B} (cs1) \quad \frac{\Pi_{BY}}{B \vdash Y}}{*A, Z \vdash Y} (cut)}{\frac{\frac{\frac{*A, Z \vdash Y}{Z \vdash A, Y} (\overline{cs1})}{Z, *Y \vdash A} (cs2) \quad \frac{\Pi_{AX}}{A \vdash X}}{Z, *Y \vdash X} (cut)}{Z \vdash X, Y} (\overline{cs2})}$$

(b)

Figure 5: (a) Principal cut on formula $A \vee B$ (b) after transformation

6.1 Principal Cuts and Belnap's Condition C8

Definition 6.1 *An application of the cut rule is left-principal [right-principal] if the cut-formula is the principal formula of the left [right] premiss of the cut rule.*

Given a derivation (tree) with a principal cut at the bottom, Belnap's condition (C8) on the rules of a Display Calculus ensures that the given derivation can be transformed into one in which the principal cut in question is replaced by cuts on smaller formulae (cuts elsewhere in the tree would remain). For example, the derivation containing the principal cut on $A \vee B$ shown in Figure 5(a) can be replaced by the derivation shown in Figure 5(b), where $(cs1)$, $(cs2)$, $(\overline{cs1})$ and $(\overline{cs2})$ are two of the display postulates and their inverses respectively. The replacement derivation contains cuts only on A and B , which are smaller formulae than $A \vee B$, and any cuts in Π_{ZAB} , Π_{AX} and Π_{BY} .

Belnap's condition C8 guarantees that there is one such transformation for every connective and this is the basis for a step of the cut-elimination proof which depends on induction on the structure or size of the cut-formula. The base case of this induction is where the cut-formula is introduced by the identity axiom. Such a cut, and its removal, are shown in Figure 6. We return to the actual mechanisation in Section 7.2.

$$\frac{A \vdash A \quad \frac{\Pi_{AY}}{A \vdash Y} (\text{intro-}A)}{A \vdash Y} (\text{cut}) \quad \text{becomes} \quad \frac{\Pi_{AY}}{A \vdash Y}$$

Figure 6: Principal cut where cut-formula is introduced by identity axiom

$$\frac{\frac{\frac{\Pi[A]}{Z[A] \vdash A} (\text{intro-}A)}{\Pi_L[A]} (\pi)}{X \vdash A} (\rho) \quad \frac{\Pi_R}{A \vdash Y} (\text{cut})}{X \vdash Y} (\text{cut}) \quad \frac{\frac{\frac{\Pi'[Y]}{Z[Y] \vdash A} (\text{intro-}A)}{Z[Y] \vdash Y} (\pi)}{\Pi_L[Y]} (\rho)}{X \vdash Y} (\text{cut})$$

Figure 7: Making a cut left-principal

The transformation of a principal cut on A into one or more cuts on strict subformulae of A , as shown above, is known as a “principal move”. We now need a way to turn arbitrary cuts into principal ones.

6.2 Transforming Arbitrary Cuts Into Principal Ones

In the case of a cut that is not left-principal, say we have a tree like the one on the left in Figure 7. Then we transform the subtree rooted at $X \vdash A$ by simply changing its root sequent to $X \vdash Y$, and proceeding upwards, changing all ancestor occurrences of A to Y . In doing this we run into difficulty at each point where A is introduced: at such points we insert an instance of the cut rule. The diagram on the right hand side of Figure 7 shows this in the case where $\Pi_L[A]$ has just a single branch where A is introduced.

In Figure 7, the notation $\Pi_L[A]$ and $Z[A]$ in the left derivation means that the sub-derivation Π_L and structure Z may contain occurrences of A which are parametric ancestors of the cut-formula A : thus (intro- A) is the lowest rule where A is the principal formula on the right of \vdash . The notation $\Pi_L[Y]$ and $Z[Y]$ in the right derivation means that all such “appropriate” instances of A are changed to Y : that is, instances of A which can be traced to the instance of A displayed on the right in $X \vdash A$. Belnap’s conditions guarantee that the rules contained in the new sub-derivation $\Pi_L[Y]$ are the same as the rules used in Π_L , and it can be proved that $\Pi_L[Y]$ is well-formed. The resulting cut in the diagram on the right of Figure 7 is left-principal. Notice that the original sub-derivation $\Pi[A]$ may be transformed into a *different* sub-derivation $\Pi'[Y]$ during this process since the parametric


```

cutOnFmls      :: "formula set => dertree => bool"
cutIsLP        :: "formula => dertree => bool"
cutIsLRP       :: "formula => dertree => bool"

```

Figure 9: Functions for reasoning about cuts

7 A Direct Proof of Cut-Admissibility (Weak-Normalisation)

In this section we describe how we proved the admissibility of cut in $\delta\mathbf{RA}$. The assumption is that we are given a derivation whose only instance of cut is at the bottom. The goal is to produce a cut-free derivation of the same end-sequent.

7.1 Functions for Reasoning About Cuts

We therefore need functions for reasoning about derivations which end with a cut. The types for these functions are shown in Figure 9. Each require the bottom node of the derivation tree to be of the form `Der seq rule dts`, and that if `rule` is (*cut*), then for:

`cutOnFmls s` : the cut is on a formula in the set `s`

`cutIsLP A` : the cut is on formula `A` and is left-principal

`cutIsLRP A` : the cut is on formula `A` and is (left- and right-)principal.

If `rule` is not (*cut*), then all these predicates are true. Note that it also follows from the actual definitions that a derivation tree satisfying any of `allDT (cutOnFmls s)`, `allDT (cutIsLP A)` and `allDT (cutIsLRP A)` has no unfinished leaves: we omit details.

7.2 Dealing With Principal Cuts

For each logical connective or constant c in the calculus, we prove that a derivation ending in a (left- and right-) principal cut which is otherwise cut-free, where the main connective of the cut formula A is c , can be transformed into another derivation of the same end-sequent, using only cuts (if any) on formulae which are strict subformulae of A . Some SML code is used to do part of the work of finding these replacement derivation trees. But the proof that such a replacement derivation tree is well-formed, for example, has to be done in the logical framework. Here is the resulting theorem for \forall : there is an analogous theorem for every logical connective and logical constant.

Theorem 7.1 (orC8) *Assume we are given a valid derivation tree dt whose only instance of cut (if any) is at the bottom, and that this cut is principal with cut-formula $A \vee B$. Then there is a valid derivation tree dtn with the same conclusion as dt , such that each cut (if any) in dtn has A or B as cut-formula.*

```
orC8 = "[| cutIsLRP (?A v ?B) ?dt; allDT wfb ?dt;
  allDT (frb rls) ?dt; allNextDTs (cutOnFmls {}) ?dt |]
  ==> EX dtn. conclDT dtn = conclDT ?dt & allDT wfb dtn &
  allDT (frb rls) dtn & allDT (cutOnFmls {?B, ?A}) dtn" : thm
```

Here:

`cutIsLRP (?A v ?B) ?dt` states that if the bottom-most rule of dt is cut, then it must be left-and-right principal on cut-formula $A \vee B$;

`allDT` the next two occurrences of `allDT` check that the derivation dt is well-formed;

`cutOnFmls {}` applied to a tree node is true if that node is not cut

`allNextDTs (cutOnFmls {}) dt` recursively applies this test to all proper subtrees of dt to check that these are all cut-free

`allDT (cutOnFmls {B, A}) dtn` checks that dtn may contain cuts (if any) on A and B only.

There is a version of Theorem 7.1 for each logical connective of $\delta\mathbf{RA}$, and these are collected into the following definition which captures that C8 holds for a given set of rules.

Definition 7.1 *C8 ?rules holds when: for all formulae fml and all derivations dt , if the bottom-most rule of dt is a principal cut on formula fml and dt is valid w.r.t. rule set $rules$ and the derivations of the premises of dt are all cut-free, then there exists a derivation dtn with the same conclusion as that of dt and such that dtn is also valid w.r.t. rule set $rules$ and all cuts in dtn are on strict subformulae of fml .*

```
C8_rls = "C8 ?rules == ALL fml dt. cutIsLRP fml dt &
  valid ?rules dt & allNextDTs (cutOnFmls {}) dt -->
  (EX dtn. conclDT dtn = conclDT dt & valid ?rules dtn
  & allDT (cutOnFmls (set (child_fmls fml))) dtn)"
```

Here we want an object level definition which can be used by other object-level functions so we use the object level arrow `-->` which is different from the meta-level arrow `==>` used in theorems.

Theorem 7.2 *Belnap's condition C8 holds for δRA .*

```
C8_rls = "C8 rls" : thm
```

We have thus shown that we can reduce a left-and-right principal cut into “smaller” cuts. We now need to show that an arbitrary cut can be made left-and-right principal.

7.3 Making A Cut (Left) Principal

We now describe how we encoded the operational aspects of cut-elimination which we described in Section 6. Recall that the task is to turn arbitrary cuts into principal ones by tracing the ancestors of the cut formula A and replacing the traced occurrences by another structure, say Y .

We define a binary relation `seqrep` between sequents which allows us to pinpoint their differences. For boolean b , structures X, Y and sequents `seq1` and `seq2`, the pair $(\text{seq1}, \text{seq2}) \in \text{seqrep } b \ X \ Y$ iff `seq1` and `seq2` are the same, except that (possibly) one or more occurrences of X in `seq1` are replaced by corresponding occurrences of Y in `seq2`, where, when b is `True` [`False`], such differences occur only in succedent [antecedent] positions. For two lists `seq11` and `seq12` of sequents, the pair $(\text{seq11}, \text{seq12}) \in \text{seqreps } b \ X \ Y$ iff each n th member of `seq11` is related to the n th member of `seq12` by `seqrep b X Y`.

For example, suppose `seqa` and `seqy` are the sequents $X \vdash A$ and $X \vdash Y$ in the trees $\Pi_L[A]$ and $\Pi_L[Y]$ respectively from Figure 7. Then, $(\text{seqa}, \text{seqy}) \in \text{seqrep True (Structform A) Y}$ since `seqy` is identical to `seqa` except that one succedent occurrence of A in $X \vdash A$ is replaced by a corresponding occurrence of Y in $X \vdash Y$.

Next come the main theorems used in the mechanised proof based on making cuts (left- and right-) principal. Several use the relation `seqrep pn (Structform A) Y`, where `pn` is some boolean valued variable standing for “positive” or “negative” polarity.

Theorem 7.3 (`seqExSub1`) *For some given boolean valued variable pn , if sequent pat is formula-free and does not contain any structure variable more than once, and can be instantiated to obtain sequent $seqa$, and $(\text{seqa}, \text{seqy}) \in \text{seqrep } pn \ (\text{Structform } A) \ Y$, then pat can be instantiated to obtain sequent $seqy$.*

```
seqExSub1 = "[| ~ seqCtnsFml ?pat; distinct (seqSVs ?pat);
  seqSubst (?fs, ?suba) ?pat = ?seqa;
  (?seqa, ?seqy) : seqrep ?pn (Structform ?A) ?Y |]
  ==> EX suby. seqSubst (?fs, suby) ?pat = ?seqy" : thm
```

Here:

`seqCtnsFml ?pat` is true iff the sequent `pat` contains formula variables, so that its negation `~ seqCtnsFml ?pat` tells us that `pat` is formula-free;

`seqSubst (fs, suby) pat` is the result of applying the substitution `(fs, suby)` to `pat`. Recall that `fs` is a substitution for formula variables and `suby` is a substitution for structure variables (see §5.3 and Fig. 3).

the symbol “:” in `(?seqa, ?seqy) : seqrep` is Isabelle notation for membership \in .

To see why `pat` must be formula-free, suppose that `pat` contains `Structform (FV ’’B’’)`, which means that `pat` is not formula-free. Then, this part of `pat` can be instantiated to `Structform A`, but not to an arbitrary structure `Y` as desired. The condition that a structure variable may not appear more than once in the conclusion of a rule is one of Belnap’s conditions [5].

The stronger result `seqExSub2` is similar to `seqExSub1`, except that the antecedent [succedent] of the sequent `pat` may contain a formula, provided that the whole of the antecedent [succedent] is that formula. There is an additional condition to avoid the problem mentioned above. That is, this function checks whether the formula is displayed as the whole of the antecedent or succedent, as required by one of Belnap’s conditions.

The result `seqExSub2` is used in proceeding up the derivation tree $\Pi_L[A]$, changing `A` to `Y`: if `pat` is the conclusion of a rule, and that rule instantiated with substitution `(fs, suba)` is used in $\Pi_L[A]$, then that same rule instantiated with substitution `(fs, suby)` is used in $\Pi_L[Y]$. This is expressed in Theorem 7.4 named `extSub2` below, which is one step in the transformation of $\Pi_L[A]$ to $\Pi_L[Y]$.

To explain theorem `extSub2` we define `bprops rule` to hold if the rule `rule` satisfies the following three properties, which are related to, but do not exactly correspond to, Belnap’s conditions (C3), (C4) and (C5):

- the conclusion of `rule` has no repeated structure variables
- if a structure variable in the conclusion of `rule` is also in a premise, then it has the same “cedency” (ie antecedent or succedent) there
- if the conclusion of `rule` has formulae, they are displayed (as the whole of one side)

For rule set `rules`, we define `C345 rules` to hold if `bprops rule` holds for every rule `rule` in `rules`.

Theorem 7.4 (extSub2) *Suppose we are given a rule `rule` and an instantiation `ruleA` of it, and given a sequent `conclY`, such that*

- (i) $(\text{conclRule } \text{ruleA}, \text{conclY}) : \text{seqrep } \text{pn } (\text{Structform } A) Y$ holds,
- (ii) bprops rule holds,
- (iii) if the conclusion of rule has a displayed formula on one side, then conclrule ruleA and conclY are the same on that side

Then there exists ruleY , an instantiation of rule , whose conclusion is conclY and whose premises premsY are, respectively, related to premsRule ruleA by $\text{seqrep } \text{pn } (\text{Structform } A) Y$.

```

extSub2 = "[| conclRule ?rule = ($?pant |- $?psuc);
  conclRule ?ruleA = ($?aant |- $?asuc);
  ?conclY = ($?yant |- $?ysuc);
  strIsFml ?pant & ?aant = Structform ?A --> ?aant = ?yant;
  strIsFml ?psuc & ?asuc = Structform ?A --> ?asuc = ?ysuc;
  ruleMatches ?ruleA ?rule; bprops ?rule;
  (conclRule ?ruleA, ?conclY) : seqrep ?pn (Structform ?A) ?Y ||
  ==> EX subY. conclRule (ruleSubst subY ?rule) = ?conclY &
  (premsRule ?ruleA, premsRule (ruleSubst subY ?rule))
  : seqreps ?pn (Structform ?A) ?Y" : thm

```

This theorem is used to show that if a sequent s of the tree $\Pi_L[A]$ is transformed to the corresponding sequent s' of $\Pi_L[Y]$, then the next sequent(s) above s in $\Pi_L[A]$ can be so transformed also. But this does not hold at the sequent where A is introduced (the sequent $Z[A] \vdash A$ of Figure 7); condition (iii) above reflects this limitation.

7.4 Turning One Cut Into Several Left-Principal Cuts

To turn one cut into several left-principal cuts we use the procedure described above. This uses `extSub2` to transform $\Pi_L[A]$ to $\Pi_L[Y]$, going stepwise up to each point where A is introduced, and there inserting an instance of the (cut) rule. Note that the theorems relating to this procedure assume that the derivation trees have no (unfinished) premises.

Theorem 7.5 (makeCutLP) *Assume the rules satisfy Belnap's conditions C3 to C5, and include (cut). Given well-formed cut-free derivation trees dtAY deriving $A \vdash Y$ and dtA deriving seqA , and given seqY such that seqY and seqA are the same except possibly that A in a succedent position in seqA is replaced by Y in seqY , (i.e. $(\text{seqA}, \text{seqY}) \in \text{seqrep True } (\text{Structform } A) Y$), there is a derivation tree deriving seqY whose cuts are all left-principal on A .*

```

makeCutLP = "[| C345 ?rules; cutr : ?rules;
  allDT (cutOnFmls {}) ?dtAY; valid ?rules ?dtAY;
  conclDT ?dtAY = (?A |- $?Y);
  valid ?rules ?dtA; allDT (cutOnFmls {}) ?dtA;
  (conclDT ?dtA, ?seqY) : seqrep True (Structform ?A) ?Y |]
  ==> EX dtY. conclDT dtY = ?seqY & allDT (cutIsLP ?A) dtY &
    valid ?rules dtY" : thm

```

Function `makeCutRP` is basically the symmetric variant of `makeCutLP`: so, in `makeCutRP`, `dtAY` is a cut-free derivation tree deriving $Y \vdash A$. But `makeCutRP` has an extra hypothesis that A is introduced at the bottom of `dtAY`. Its result is a derivation tree deriving `seqY` whose cuts are all (left- and right-) principal on A .

These were the most difficult to prove in this cut-elimination proof. The proofs proceed by structural induction on the initial derivation tree. The base case of the induction is when A is introduced by the axiom (*id*), in which case, tree `dtAY` is substituted for $A \vdash A$ as in Figure 8. The inductive step involves an application of `extSub2`, except where the formula A is introduced. If A is introduced by an introduction rule, then the inductive step involves inserting an instance of a (cut) into the tree, and then applying the inductive hypothesis.

Next we have the theorem expressing the transformation of the whole derivation tree, as shown in the diagrams.

Theorem 7.6 (allLP) *Assume that the rules satisfy conditions C3 to C5. Given a valid derivation tree dt containing just one cut, which is on formula A and is at the root of dt , there is a valid tree dt_n with the same conclusion (root) sequent, all of whose cuts are left-principal and are on A .*

```

allLP = "[| C345 ?rules; cutOnFmls {?A} ?dt; valid ?rules ?dt;
  allNextDTs (cutOnFmls {}) ?dt |]
  ==> EX dt_n. conclDT dt_n = conclDT ?dt &
    allDT (cutIsLP ?A) dt_n & valid ?rules dt_n" : thm

```

`allLRP` is a similar theorem where we start with a single left-principal cut, and produce a tree whose cuts are all (left- and right-) principal.

7.5 Putting It All Together

A monolithic proof of the cut-admissibility theorem would be very complex, involving either several nested inductions or a complex measure function, since the transformations above replace one arbitrary cut by many left-principal cuts, one left-principal cut by many principal cuts and one principal cut by one or two cuts on subformulae, and we need, ultimately, to replace many arbitrary cuts in a given derivation tree. We can manage

this complexity by considering how we would write a program to perform the elimination of cuts from a derivation tree. One way would be to use a number of mutually recursive routines, as follows:

`elim` eliminates a single arbitrary cut, by turning it into several left-principal cuts and using `elimAllLP` to eliminate them where ...

`elimAllLP` eliminates several left-principal cuts, by repeatedly using `elimLP` to eliminate the top-most remaining one where ...

`elimLP` eliminates a left-principal cut, by turning it into several principal cuts and using `elimAllLRP` to eliminate them where ...

`elimAllLRP` eliminates several principal cuts, by repeatedly using `elimLRP` to eliminate the top-most remaining one where ...

`elimLRP` eliminates a principal cut, by turning it into several cuts on smaller cut-formulae, and using `elimAll` to eliminate them where ...

`elimAll` eliminates several arbitrary cuts, by repeatedly using `elim` to eliminate the top-most remaining one where ...

Such a program would terminate because any call to `elim` would (indirectly) call `elim` only on smaller cut-formulae.

We turn this program outline into a proof. Each routine listed above, of the form “routine *P* does ... and uses routine *Q*” will correspond to a theorem which will say essentially “if routine *Q* completes successfully then routine *P* completes successfully” assuming they are called with appropriately related arguments.

We imitate this procedure by defining two predicates, `canElim` and `canElimAll`, whose types and meanings are given below (assuming trees which are valid with respect to the rule set `rules`):

```
canElim      :: "rule set => (dertree => bool) => bool"
canElimAll   :: "rule set => (dertree => bool) => bool"
```

When we use them, the argument `f` will be one of the functions `cutOnFmls`, `cutIsLP` and `cutIsLRP` from Figure 9.

Definition 7.2 (`canElim`) *canElim rules f holds for property f if, for any valid derivation tree dt satisfying f and containing at most one cut, which is at the bottom, there is a valid cut-free derivation tree dtn which has the same conclusion as dt.*

```
"canElim ?rules ?f == (ALL dt.
  ?f dt & allNextDTs (cutOnFmls {}) dt &
  valid ?rules dt -->
  (EX dtn. allDT (cutOnFmls {}) dtn &
  valid ?rules dtn & conclDT dtn = conclDT dt))"
```

Definition 7.3 (`canElimAll`) *canElimAll* rules f means that if every subtree of a given valid tree dt satisfies f , then there is a valid cut-free derivation tree dt' , with the same conclusion as dt , such that if the bottom rule of dt is not (cut), then the same rule is at the bottom of both dt' and dt .

```
"canElimAll ?rules ?f == (ALL dt.
  allDT ?f dt & valid ?rules dt -->
  (EX dt'. (botRule dt ~= cutr --> botRule dt' = botRule dt) &
    conclDT dt' = conclDT dt & allDT (cutOnFmls {}) dt' &
    valid ?rules dt'))"
```

We restate `allLP` and `allLRP` using `canElim` and `canElimAll`.

Theorem 7.7

- (a) `allLP'`: If we can eliminate any number of left-principal cuts on A from any valid tree, then we can eliminate a single arbitrary cut on A from the bottom of any valid tree.
- (b) `allLRP'`: If we can eliminate any number of principal cuts on A from any valid tree, then we can eliminate a single left-principal cut on A from the bottom of any valid tree.

```
allLP' = "[| C345 ?rules; canElimAll ?rules (cutIsLP ?A) |]
  ==> canElim ?rules (cutOnFmls {?A})" :thm
allLRP' = "[| C345 ?rules; canElimAll ?rules (cutIsLRP ?A) |]
  ==> canElim ?rules (cutIsLP ?A)" :thm
```

Proof:

- (a) Consider a derivation tree dt with end sequent $X \vdash Y$ and a single cut at the bottom. As `allLP` holds for dt , there exists a derivation tree dt_n with the same end sequent $X \vdash Y$, but whose cuts are all left-principal and on A . Clearly, if we can eliminate all of these to produce a cut-free derivation of $X \vdash Y$, then we have eliminated the cut from dt .
- (b) This has a corresponding proof.

Q.E.D.

Now if we can eliminate one arbitrary cut (or left-principal cut, or principal cut) then we can eliminate any number, by eliminating them one at a time starting from the top-most cut. This is easy because eliminating a cut affects only the derivation tree above the cut: there is just a slight complication because we need to show that eliminating a cut does not change

a lower cut from being principal to not principal, but this is not difficult. This gives the following three results (although the first two are limited to cuts on a particular cut-formula, while in the third, `?s` can be the set of all formulae).

Theorem 7.8 *If we can eliminate one arbitrary cut (or left-principal cut, or principal cut) from any given derivation tree, then we can eliminate any number of such cuts from any given derivation tree.*

```
elimFmls = "canElim ?rules (cutOnFmls ?s) ==>
  canElimAll ?rules (cutOnFmls ?s)" : thm
elimLP = "canElim ?rules (cutIsLP ?A) ==>
  canElimAll ?rules (cutIsLP ?A)" : thm
elimLRP = "canElim ?rules (cutIsLRP ?A) ==>
  canElimAll ?rules (cutIsLRP ?A)" : thm
```

We also have the theorems such as `orC8`, combined into `C8_rls` (see §7.2) dealing with a tree with a single (left- and right-) principal cut on a given formula. The following theorem is a simple rewrite of the condition (C8).

Theorem 7.9 *A tree with a single (left- and right-) principal cut on a given formula can be replaced by a tree with arbitrary cuts on the immediate subformulae (if any) of that formula.*

```
allch = "[| C8 ?rules;
  canElimAll ?rules (cutOnFmls (set (child_fmls ?fml))) |]
  ==> canElim ?rules (cutIsLRP ?fml)" : thm
```

Together with the theorems `elimLRP`, `allLRP'`, `elimLP`, `allLP'` and `elimFmls`, we now have theorems corresponding to the six routines described above. As noted already, a call to `elim` would indirectly call `elim` with a smaller cut-formula as argument, and so the program would terminate, the base case of the recursion being the trivial instance of Figure 6 mentioned above. Correspondingly, the theorems we now have can be combined to give `th8ch`, as follows. The difference between the two Isabelle formulations `th8ch` and `th8ch'` represents an obvious but necessary step, as often happens when doing formal proofs.

Theorem 7.10 (`th8ch`, `th8ch'`) *We can eliminate a cut on a formula if we can eliminate a cut on each of the formula's immediate subformulae ("child formulae").*

```
th8ch = "[| C345 ?rules; C8 ?rules;
  canElim ?rules (cutOnFmls (set (child_fmls ?A))) |]
  ==> canElim ?rules (cutOnFmls {?A})" : thm
```

```

th8ch' = "[| C345 ?rules; C8 ?rules;
  ALL f:child_fm1s ?A. canElim ?rules (cutOnFm1s {f}) |]
  ==> canElim ?rules (cutOnFm1s {?A})" : thm

```

From `th8ch'`, we can now use structural induction on the structure of a formula to prove that we can eliminate a cut on any given formula `fm1`: that is, we can eliminate any cut. The base cases of the induction are where `fm1` has no child formulae.

Theorem 7.11 *A sequent derivable using (cut) is derivable without (cut).*

Proof: Using `elimFm1s` from Theorem 7.8, it follows that we can eliminate any number of cuts, as reflected by the following sequence of theorems:

- (a) if the rules `rules` satisfy condition `C345` and `C8` then a bottom-most cut on a formula `fm1` (and so, on any formula) is eliminable (`UNIV` is the universal set for its type):

```

canElimFm1 = "[| C345 ?rules; C8 ?rules |] ==>
  canElim ?rules (cutOnFm1s {?fm1})" : thm
canElimAny = "[| C345 ?rules; C8 ?rules |] ==>
  canElim ?rules (cutOnFm1s UNIV)": thm

```

- (b) if the rules `rules` satisfy condition `C345` and `C8` then all cuts are eliminable:

```

canElimAll = "[| C345 ?rules; C8 ?rules |] ==>
  canElimAll ?rules (cutOnFm1s UNIV)": thm

```

- (c) if the rules `rules` satisfy condition `C345` and `C8` then if sequent `concl` is derivable using `rules` then it is derivable using `rules - cutr`:

```

cutElim    = "[| C345 ?rules; C8 ?rules;
  IsDerivableR rules {} ?concl |] ==>
  IsDerivableR (?rules - {cutr}) {} ?concl" : thm

```

Q.E.D.

Corollary 7.1 *The rule (cut) is admissible in δRA .*

7.6 Tracing Belnap’s Conditions of the Display Calculus

Belnap’s cut-elimination theorem [5] states that any Display Calculus satisfying his properties (C2) to (C8) satisfies the cut-elimination theorem. Here we trace the use of these properties in our proof for the specific case of $\delta\mathbf{RA}$. We refer to the conditions in the form given by Kracht [21].

As we have seen, we proved properties which are satisfied by each rule of $\delta\mathbf{RA}$. They use the following definitions:

rls: is the set of rules of $\delta\mathbf{RA}$

seqNoSSF seq: if a formula appears on the left/right hand side of the \vdash in **seq** then the whole of that side is that formula, where it is “displayed”.

seqSVs seq: a list of the structure variables in **seq**.

distinct list: **list** contains no member more than once.

seqSVs’ bool seq: where **bool** is **True** [**False**], a list of the structure variables in succedent [antecedent] positions in **seq**

C3 rule: the conclusion of **rule** contains no structure variable more than once.

C4 rule: if a structure variable appears in a premise and in the conclusion of **rule** then it has the same “cedency” (antecedent part or succedent part) in both places.

C5 rule: in the conclusion of **rule**, no structure has a sub-structure which is a formula, although either side may be a displayed formula: defined as **seqNoSSF (conclRule rule)**

C345 rules: for every **rule** \in **rules**, **C3 rule**, **C4 rule** and **C5 rule** hold.

Belnap’s condition (C3) is that **C3** above holds for every rule of the calculus.

His condition (C4) says that if a structure variable appears anywhere in a rule in an antecedent [succedent] position, then it does not appear in a succedent [antecedent] position. In fact, all we actually need for our proof is our **C4** above, which is weaker than Belnap’s (C4): our **C4** permits a structure variable to appear in both antecedent and succedent positions in the premises, provided that it does not appear in the conclusion. Note that **C3** prevents a structure variable from appearing in both antecedent and succedent positions in the conclusion.

In fact, most Display Calculi satisfy the condition that if a structure variable appears in the premises of a rule then it appears in the conclusion.

In Kracht’s formulation, Belnap’s (C5) is equivalent to our C5, because formula variables can never be parameters. Moreover, under Kracht’s formulation, Belnap’s (C2), (C6) and (C7) are trivially satisfied.

This is why we have proved only the results `C345_r1s` and `C8_r1s` capturing that the set `r1s` of rules of $\delta\mathbf{RA}$ satisfies `C345` (above) and `C8` (§7.2).

Note that an earlier version of our work used the following rule, which is actually stronger than required, but makes the proof simpler.

If a structure variable appears in a premise of a rule, in antecedent [succedent] position, then it also appears in the conclusion of that rule in antecedent [succedent] position.

8 Wansing’s Strong Normalisation Proof

The ideas inherent in the principal and parametric moves outlined in Section 6 are well-known, but they were first applied to strong normalisation for a display calculus by Wansing in [39] (reproduced in [40, Section 4.3]). In Wansing’s book [40], he “proves” a strong normalisation result, that any (sufficiently long) sequence of steps in the process of cut-elimination terminates. The steps allowed in this context are those described as:

principal moves such as the transformation from (a) to (b) in Figure 5

parametric moves such as the transformation in Figure 7.

When showing that cut is an admissible rule, as done by Belnap [5], it suffices to do these transformations only on a topmost cut in a derivation tree. But to show strong normalisation, as done by Wansing [40], we must allow these moves to occur for a cut which is *not* a topmost cut. The only restriction is that, for a parametric move, the portion of the derivation tree that is changed must not contain a cut: that is, there must be no cuts in the parts of the trees in Figure 7 shown below:

$$\frac{\frac{Z[A] \vdash A}{\Pi_L[A]} (\pi)}{X \vdash A} (\rho) \qquad \frac{\frac{Z[Y] \vdash Y}{\Pi_L[Y]} (\pi)}{X \vdash Y} (\rho)$$

We have implemented large parts of Wansing’s strong normalisation proof. In fact we completed it, subject to assertions (which we had intended to prove) about the effect of primitive and parametric reductions. These assertions are too complex to include and explain here, but they implied (*inter alia*) that the subtree $\Pi'[Y]$ and its conclusion $Z[Y] \vdash A$ in the right (transformed) tree of Figure 7 is the same as the tree $\Pi[A]$ and its conclusion $Z[A] \vdash A$ in the left (original) tree: that is, the transformation process does not alter Π but leaves it intact. In fact, this is true only if Z contains no parametric ancestors of A .

$$\frac{\frac{\Pi'}{X' \vdash A} \text{ (intro-}A\text{)} \quad \frac{\Pi''}{X'' \vdash A} \text{ (intro-}A\text{)}}{\frac{\frac{\Pi_L[A]}{X \vdash A} \text{ } (\rho) \quad \frac{\Pi_R}{A \vdash Y} \text{ } (cut)}{X \vdash Y} \text{ (} \pi \text{)}}$$

(i) before the transformation

$$\frac{\frac{\frac{\Pi'}{X' \vdash A} \text{ (intro-}A\text{)} \quad \frac{\Pi_R}{A \vdash Y} \text{ } (cut)}{X' \vdash Y} \quad \frac{\frac{\Pi''}{X'' \vdash A} \text{ (intro-}A\text{)} \quad \frac{\Pi_R}{A \vdash Y} \text{ } (cut)}{X'' \vdash Y} \text{ } (\pi)}{\frac{\Pi_L[Y]}{X \vdash Y} \text{ } (\rho)}$$

(ii) after the transformation

Figure 10: Branch in left subtree

We said in Section 6.2 that “This construction generalizes easily to the case where A is introduced ... at more than one point ...”, and we now elaborate on this. Some more complicated diagrams are needed.

Figure 10(i) illustrates the situation where A is introduced twice, by virtue of a branch in the left-hand-side subtree, that is, in the Π_L of Figure 7.

In this situation our assertion implies, in effect, that the subtrees Π' and Π'' and their respective conclusions $X' \vdash A$ and $X'' \vdash A$, which are the left premises of the new cuts in the transformed derivation tree, actually appear in the original tree, as indeed they do in Figure 10.

Consider now the situation shown in Figure 11 where A is $\neg B$. Here a second occurrence of A is introduced by contraction, so that A is introduced twice, but “in succession”, rather than “in parallel”.

If we now try to match the left tree in Figure 7 with the tree in Figure 11(i), we get that A of Figure 7 is $\neg B$ in Figure 11, $Z[\neg B]$ is $(*\neg B, X)$ and $\Pi[\neg B]$ is the subtree of Figure 11(i) whose root sequent is $*\neg B, X \vdash *B$. But this subtree Π does *not* appear unchanged in the transformed tree in Figure 11(ii). Rather, Figure 11(ii) has, in the corresponding position, a subtree whose root sequent is $*Y, X \vdash *B$ and which contains an extra cut. Thus our assertion failed in this case, and so we could not prove Wansing’s result. Incidentally, the subtree Π_1 from Figure 11(i) does remain unchanged since it cannot contain any parametric ancestors of the original cut-formula $\neg B$.

$$\begin{array}{c}
\frac{\Pi_1}{X, B \vdash *B} (\rho) \\
\frac{}{X, B \vdash \neg B} (\vdash \neg) \\
\frac{}{* \neg B, X \vdash *B} (\text{dp}) \\
\frac{}{* \neg B, X \vdash \neg B} (\vdash \neg) \\
\frac{}{X \vdash \neg B, \neg B} (\text{dp}) \\
\frac{}{X \vdash \neg B} (\vdash \text{cont}) \quad \frac{\Pi_R}{\neg B \vdash Y} \\
\hline
X \vdash Y \quad (\text{cut})
\end{array}$$

(i) before the transformation

$$\begin{array}{c}
\frac{\Pi_1}{X, B \vdash *B} (\rho) \quad \frac{\Pi_R}{\neg B \vdash Y} \\
\frac{}{X, B \vdash \neg B} (\vdash \neg) \quad \frac{}{\neg B \vdash Y} \\
\hline
X, B \vdash Y \quad (\text{cut}) \\
\frac{}{*Y, X \vdash *B} (\text{dp}) \\
\frac{}{*Y, X \vdash \neg B} (\vdash \neg) \quad \frac{\Pi_R}{\neg B \vdash Y} \\
\hline
*X, X \vdash Y \quad (\text{cut}) \\
\frac{}{*Y, X \vdash Y} (\text{dp}) \\
\frac{}{X \vdash Y, Y} (\vdash \text{cont}) \\
\hline
X \vdash Y
\end{array}$$

(ii) after the transformation

Figure 11: Two successive introductions of $\neg B$

We attempted to repair this defect by reference to the relevant part of Wansing’s proof [40, page 54]. Unfortunately it is based on a diagram much like Figure 7 and appears to rely on the fact that the subtree Π in Figure 7 appears unchanged in the transformed tree. Wansing states that the extension to situations more complex than that of Figure 7 is to be shown “analogously”, and he makes no distinction between the situations of Figures 10 and 11.¹

9 Discussion

A proof of cut-elimination can be attempted along the following lines. Define a measure on each cut (often called *rank*), where the relative rank of two cuts depends primarily on the size of the cut-formula, and secondarily upon the size (in some defined sense) of the derivation tree rooted at that cut. The proof then proceeds by induction on the rank of a cut, with a cut being replaced by cut(s) of smaller rank. For example, as shown in Figure 7 and Figure 10, a cut is replaced by cut(s) of smaller rank, since the subtree(s) rooted at the new cut(s) are composed of parts of the original tree. However the example in Figure 11 highlights the problem of contractions above cut: there, if we look at the lower of the two new cuts in the new subtree, we see that its left-hand branch is not part of the original tree. Many procedures are used to get around this problem of “contractions above cuts” in a Gentzen-style sequent calculus, but note that some can be quite complex (*e.g.*, [6]).

10 A New Proof of Strong Normalisation

We now give a new direct proof of strong normalisation which does not rely on the size of derivation trees.

10.1 Preliminary Definitions and Results for Strong Normalization

In this section we set up the machinery needed to prove strong-normalisation. We first define the notion of reduction, and define what it means for a derivation tree to be strongly-normalising with respect to such reductions.

¹In an email dated 20th of January 2002, Heinrich Wansing has conceded that there is a problem with his proof. His remedy is to define an additional reduction which is used in certain special cases previously handled by “parametric” moves. This additional reduction is a combination of “parametric” and “principal” moves, and is more restrictive than the reductions allowed by our proof. We can mimic his new reduction by appropriately restricting the sequence in which we carry out our moves in these special cases. Note, however, that the two proofs still proceed using completely different methods.

We use the same definition of reduction as does Wansing [40, §4.2, §4.3]. Given a derivation tree with (*cut*) at its root, changes can be made to the tree to deal with that particular cut; we call these “cut-reductions”. Following Wansing [40, §4.2], we can classify these cut-reductions as primitive or parametric. We define and discuss these later.

More generally, we can change a tree by performing a cut-reduction on any subtree; we call such changes *reductions*.

10.2 Defining Strongly Normalising Derivations

Assuming an irreflexive binary relation `cutReduces` over derivations, to be defined later, we define the reduction relation mentioned above using Isabelle’s inductive definition feature [28, § 2.10], which defines the minimal set(s) closed under some given rules. For example, the rules $\{0 \in \mathcal{S}; n \in \mathcal{S} \implies n + 2 \in \mathcal{S}\}$ define \mathcal{S} to be the set of even natural numbers, although the set of all naturals also satisfies the rules. For more details, see [28, § 2.10].

The actual definition of the relation `cutReduces` is given later, meaning that all proofs until then are modulo this definition. But we explain now that when a derivation Π_0 `cutReduces` to another derivation Π_1 (we say $(\Pi_1, \Pi_0) \in \text{cutReduces}$), the derivation Π_0 has an instance of (*cut*) at the bottom, and the change from Π_0 to Π_1 involves changing the tree at that point. When we apply a single such change anywhere in a tree Π_0 to get Π_1 , then we say $(\Pi_1, \Pi_0) \in \text{reduction}$.

Definition 10.1 (reduces, reduction) *A derivation tree Π_0 reduces to derivation tree Π_1 ($(\Pi_1, \Pi_0) \in \text{reduction}$) if either (a) Π_0 `cutReduces` to Π_1 , or (b) Π_0 and Π_1 are identical except that exactly one immediate subtree Π_0 reduces to the corresponding immediate subtree of Π_1 .*

Wansing [40, p. 52] defines a strongly normalising derivation tree as a tree from which every sequence of reductions is finite. We define inductively the set of strongly normalising derivation trees.

Definition 10.2 (sn_set, strongly normalising) *The set `sn_set` is the smallest set of derivation trees such that:*

- (a) *if Π_0 cannot be reduced then $\Pi_0 \in \text{sn_set}$*
- (b) *if every tree Π_1 to which Π_0 reduces is in `sn_set` then $\Pi_0 \in \text{sn_set}$.*

In the code this is defined in terms of the well-founded part of an arbitrary relation. Our terminology relies on the fact that in classical logic (and so in higher-order logic, $x \in \text{wfp}(r)$ is equivalent to the non-existence of an infinite sequence of r -reductions from x .

```

inductive "wfp r"
  intrs
    wfpI "(ALL dtn.
      (dtn, ?dt) : r --> dtn : wfp r) ==> ?dt : wfp r"

sn_set_def = "sn_set == wfp reduction"

```

Note that Definition 10.2.(b) subsumes Definition 10.2.(a), which is why there is only one clause in the corresponding Isabelle code.

Definition 10.3 (strongly normalising) *A derivation tree is strongly normalising iff it is a member of `sn_set`.*

Note that `cutReduces` is irreflexive, so if Π_0 cannot be reduced at all, then it follows that $\Pi_0 \in \text{sn_set}$.

10.3 Various Well-Founded Orderings

To prove that every derivation tree is strongly normalising, we use a binary relation `dtorder` on derivation trees, and show that it is well-founded. First we define an auxiliary relation, `sn1red`, which is also an inductively defined set.

Definition 10.4 (sn1red) *For two lists `dtl1` and `dtl0` of derivation trees, `sn1red dtl1 dtl0` holds iff the lists `dtl1` and `dtl0` are non-empty and differ at only one position where `dtl0` contains tree `dt0` and `dtl1` contains tree `dt1`, and `dt0` reduces to `dt1`, and `dt0` is strongly normalising.*

```

sn1I1 = "(?h', ?h) : reduction & ?h : sn_set
  ==> (?h' # ?t, ?h # ?t) : sn1red"
sn1I2 = "(?t', ?t) : sn1red ==> (?h # ?t', ?h # ?t) : sn1red"

```

We define four binary relations, `LRPorder`, `cutorder`, `sn1order` and `dtorder` on derivation trees as sets of ordered pairs as follows. These definitions, except `cutorder`, use Isabelle's inductive definition feature.

Definition 10.5 (LRPorder) $\Pi_1 <_{LRP} \Pi_0$ if the bottom inferences of derivations Π_1 (*Der seq1 cutr dtl1*) and Π_0 (*Der seq0 cutr dtl0*) are both (cut), and either

- (a) the cut in Π_0 is neither left- nor right- principal, and the cut in Π_1 is left-principal or
- (b) the cut in Π_0 is neither left- nor right- principal, and the cut in Π_1 is right-principal, or

(c) the cut in Π_0 is not (left- and right-)principal, and the cut in Π_1 is (left- and right-)principal.

```

inductive "LRPorder"
  intrs
    LRPoL "[| ~ cutLPcf (Der ?seq0 cutr ?dtl0) ;
            ~ cutRPcf (Der ?seq0 cutr ?dtl0) ;
            cutLPcf (Der ?seq1 cutr ?dtl1) |] ==>
          (Der ?seq1 cutr ?dtl1, Der ?seq0 cutr ?dtl0) : LRPorder"
    LRPoR "[| ~ cutLPcf (Der ?seq0 cutr ?dtl0) ;
            ~ cutRPcf (Der ?seq0 cutr ?dtl0) ;
            cutRPcf (Der ?seq1 cutr ?dtl1) |] ==>
          (Der ?seq1 cutr ?dtl1, Der ?seq0 cutr ?dtl0) : LRPorder"
    LRPoLR "[| ~ cutLRPcf (Der ?seq0 cutr ?dtl0) ;
            cutLRPcf (Der ?seq1 cutr ?dtl1) |] ==>
          (Der ?seq1 cutr ?dtl1, Der ?seq0 cutr ?dtl0) : LRPorder"

```

Definition 10.6 (cutorder) $\Pi_1 <_{cut} \Pi_0$ if the bottom inferences of derivations Π_1 and Π_0 are both (cut), and if either

- (a) the size of the cut-formula of Π_1 is smaller than that of Π_0 , or
- (b) each derivation has the same cut-formula, and $\Pi_1 <_{LRP} \Pi_0$.

```

cutorder_def =
  "cutorder == inv_image (measure size <*lex*> LRPorder)
  (%dt. (cutForm dt, dt)) Int {(dt1, dt0). isCut dt1 & isCut dt0}"

```

The definition `cutorder_def` uses Isabelle's notation `%` for lambda abstraction, `Int` for set intersection, and a feature `{...}` for defining sets, and the following Isabelle/HOL functions, which help to construct well-founded relations; see [28, §2.9.2]:

$$\begin{aligned}
(t_1, t_2) \in \text{measure size} & \text{ iff } \text{size } t_1 < \text{size } t_2 \\
(t_1, t_2) \in R_1 <*lex*> R_2 & \text{ iff } (t_1, t_2) \in R_1 \text{ or } (t_1 = t_2 \text{ and } (t_1, t_2) \in R_2) \\
(t_1, t_2) \in \text{inv_image } R \ f & \text{ iff } (f \ t_1, f \ t_2) \in R.
\end{aligned}$$

Definition 10.7 (sn1order)

$\Pi_1 <_{sn1} \Pi_0$ if Π_0 (Der seq rule `dtl0`), and Π_1 (Der seq rule `dtl1`) are the same except that the ordered pair $(dtl1, dtl0)$ formed from their respective lists of immediate subtrees is in `sn1red`: that is, if Π_0 and Π_1 are the same except that one of the immediate subtrees of Π_0 is strongly normalising and reduces to the corresponding immediate subtree of Π_1 .

```

inductive "sn1order"
  intrs
    sn1I "(?dtl1, ?dtl0) : sn1red ==>
          (Der ?seq ?rule ?dtl1, Der ?seq ?rule ?dtl0) : sn1order"

```

Definition 10.8 (*dtorder*)

$\Pi_1 <_{dt} \Pi_0$ iff any of the following hold:

- (a) the bottom inference of Π_1 is not (*cut*), but that of Π_0 is
- (b) $\Pi_1 <_{cut} \Pi_0$
- (c) $\Pi_1 <_{sn1} \Pi_0$.

```

inductive "dtorder"
  intrs
    cnc "?rule ~ = cutr ==>
          (Der ?seq1 ?rule ?dtl1, Der ?seq0 cutr ?dtl0) : dtorder"
    dtc "?dts : cutorder ==> ?dts : dtorder"
    snr "?dts : sn1order ==> ?dts : dtorder"

```

The expression `?rule = cutr` stipulates that the bottom-most rule of the derivation tree `dtl1` is not *cut*.

We can then prove the following theorem with relative ease, so we omit details.

Theorem 10.1 (*wf_LRPorder, wf_cutorder, wf_sn1order*) *The relations LRPorder, cutorder and sn1order are well-founded.*

```

wf_LRPorder = "wf LRPorder" : thm
wf_cutorder = "wf cutorder" : thm
wf_sn1order = "wf sn1order" : thm

```

Despite the notation, these relations are *not* reflexive, and some are *not* transitive. Intuitively, $(\Pi_1, \Pi_0) \in dtorder$ means that Π_1 is closer to being cut-free (in some sense) than is Π_0 . To prove that *dtorder* is well-founded, we first need a result on the interaction between *cutorder* and *sn1order*.

Lemma 10.1 (*sntr'*) *If $\Pi_2 <_{cut} \Pi_1$ and $\Pi_1 <_{sn1} \Pi_0$ then $\Pi_2 <_{cut} \Pi_0$.*

```

sntr' = "[| (?dty, ?dtza) : cutorder; (?dtza, ?dtz) : sn1order |]
         ==> (?dty, ?dtz) : cutorder" : thm

```

Proof: First note that the bottom inference of Π_1 is (*cut*), and $\Pi_1 <_{sn1} \Pi_0$, so that the bottom inference of Π_0 is (*cut*) and has the same cut-formula. Secondly, $\Pi_2 <_{cut} \Pi_1$ means either that Π_2 has a smaller cut-formula than

does Π_1 , or that $\Pi_2 <_{\text{LRP}} \Pi_1$. Therefore, if $\Pi_2 <_{\text{cut}} \Pi_1$ by virtue of the sizes of their cut-formulae, then $\Pi_2 <_{\text{cut}} \Pi_0$ for the same reason.

Suppose, on the other hand, that $\Pi_2 <_{\text{LRP}} \Pi_1$. Since $\Pi_1 <_{\text{sn1}} \Pi_0$, one immediate subtree Π_0^s of Π_0 reduces to a corresponding subtree of Π_1 . We consider cases for where this reduction took place:

- (a) If the reduction is in a proper subtree of Π_0^s , then the reduction does not affect the lowest inference of Π_0 , which we know to be a (*cut*). Since Π_1 and Π_0 are identical except for this reduction, Π_1 is left-(right-)principal iff Π_0 is so. Hence $\Pi_2 <_{\text{LRP}} \Pi_0$.
- (b) If, on the other hand, Π_0^s is the left (right) immediate subtree of Π_0 and therefore has (*cut*) as its lowest inference, then Π_0 is not left-(right-) principal. Therefore if Π_0^s is reduced to turn Π_0 into Π_1 , and $\Pi_2 <_{\text{LRP}} \Pi_1$, it follows that $\Pi_2 <_{\text{LRP}} \Pi_0$.

In both cases, $\Pi_2 <_{\text{cut}} \Pi_0$. **Q.E.D.**

Theorem 10.2 (`wf_dtorder`) *dtorder* is well-founded.

```
wf_dtorder = "wf dtorder" : thm
```

Proof: It is easy to see that an infinite `dtorder`-decreasing chain $\Pi_0 >_{\text{at}} \Pi_1 >_{\text{at}} \Pi_2 >_{\text{at}} \dots$ must contain either an infinite `sn1order`-decreasing chain of trees whose bottom inference is not (*cut*), or an infinite chain $\Pi_0, \Pi_1, \Pi_2, \dots$ of trees whose bottom inference is (*cut*) and for each pair (Π_{i+1}, Π_i) , either $\Pi_{i+1} <_{\text{sn1}} \Pi_i$ or $\Pi_{i+1} <_{\text{cut}} \Pi_i$.

As `sn1order` is well-founded, there is no infinite `sn1order`-decreasing chain; therefore the chain contains infinitely many pairs $\Pi_{i+1} <_{\text{cut}} \Pi_i$. But intermediate pairs in `sn1order` may be “absorbed” – for example if also $\Pi_i <_{\text{sn1}} \Pi_{i-1}$, then by Lemma 10.1, we have $\Pi_{i+1} <_{\text{cut}} \Pi_{i-1}$. Thus there is an infinite `cutorder`-decreasing chain, contradicting that `cutorder` is well-founded. **Q.E.D.**

10.4 Inheriting Strong Normalisation from Immediate Subtrees

We next define `snHered`, a property of derivation trees which indicates that a tree inherits strong normalization from its immediate subtrees.

Definition 10.9 (`snHered`) *A derivation tree Π satisfies `snHered` iff:*

if all the immediate subtrees of Π are strongly normalising then Π is strongly normalising.

```
snHered_def = "snHered ?dt ==
  set (nextUp ?dt) <= sn_set --> ?dt : sn_set"
```

Here, `nextUp dt` returns the immediate subtrees of `dt`, and `set` allows us to treat these as a set, while `<=` is Isabelle's notation for the subset relation.

The next lemma follows from this definition.

Lemma 10.2 (`hereds_sn`) *A derivation tree Π is strongly normalising iff every subtree of Π has the property `snHered`.*

```
hereds_sn = "(ALL dts.
  isSubt ?dt dts --> snHered dts) = (?dt : sn_set)" : thm
```

Proof:

Part 1: We assume that Π is strongly normalising and show that the immediate subtrees of Π are strongly normalising. By applying this result repeatedly, we deduce that every subtree of a strongly normalising derivation tree is strongly normalising. Thus, if Π is strongly normalising, then all subtrees of Π are strongly normalising, which automatically implies that all subtrees of Π satisfy `snHered`.

The main step is thus to prove that all immediate subtrees of a strongly normalising tree are themselves strongly normalising. For this we use the theorem below which is generated automatically by Isabelle from the inductive definition of `wfp` (recall that `sn_set` is `wfp reduction`):

```
wfp.induct = "[| ?xa : wfp ?r ; (!!dt. (ALL dtn.
  (dtn, dt) : ?r --> dtn : wfp ?r & ?P dtn) ==> ?P dt ) |]
  ==> ?P ?xa" : thm
```

By letting `?xa` be Π , the first conjunct `?xa : sn_set` in the antecedent of `wfp.induct` is true by the assumption that Π is strongly normalising.

By letting `(nextUp .)` be a function that returns a list containing the immediate subtrees of its argument, and letting `(?P .)` be `(set (nextUp .) <= sn_set)`, the conclusion `(set (nextUp ?xa) <= sn_set)` of theorem `wfp.induct` says that the immediate subtrees of Π are strongly normalising. It remains to prove that the second premise of `wfp.induct` is true.

The second premise is an implication whose main connective is the inner `==>`. For a contradiction, we let `dt` be an arbitrary tree and assume that the antecedent of this inner implication is true, and that its conclusion `(set (nextUp dt) <= sn_set)` is false.

The latter tells us that `dt` has an immediate subtree `dts` which is not strongly normalising: meaning that `dts` reduces to some `dts'` which is **not strongly normalising**. Since the reduction happens to an immediate subtree of `dt`, we know that `dt` itself reduces to some `dt'` and that `dts'` is an immediate subtree of `dt'`.

Letting dtn be this dt' in the antecedent of --> tells us that dt' is strongly normalising and that $(\text{set } (\text{nextUp } \text{dtn}) \leq \text{sn_set})$ holds. The latter says that the immediate subtrees of dt' are strongly normalising. Since dts' is one such immediate subtree, we have that dts' is **strongly normalising**: giving the contradiction we seek.

Thus the second conjunct of the antecedent of wfp.induct is also true. Note that the “fact” that dt' is strongly normalising is not used: such “facts” seem to appear frequently in proofs that use theorems like wfp.induct which are generated automatically from Isabelle’s inductive definitions, and are useful only occasionally.²

Since the two conjuncts of the antecedent of wfp.induct are true, its conclusion is true: the immediate subtrees of Π are strongly normalising.

Part 2: We use induction on the structure of the derivation tree Π . So we assume, as an inductive hypothesis, that the forward implication of the theorem holds for every Π_s in $\text{nextUp } \Pi$. We assume that every subtree of Π , including itself, has the property snHered : therefore, this also holds for every Π_s and so, by the inductive hypothesis, each Π_s is in sn_set . Then, as $\text{snHered } \Pi$, Π is in sn_set . **Q.E.D.**

10.5 Reasoning About Cut-Reductions

We intend to define cut-reduction in a way that enables us to make some assertions about cut-reductions. We first define properties nparRedP and c8redP of reductions (which in fact apply to parametric and primitive reductions respectively). The definitions do not require that the bottom inference of the derivation be (*cut*), but they are used only where this is so.

Definition 10.10 (nparRedP) *The relation nparRedP holds between two derivation trees Π_0 and Π_1 if every subtree Π_1^s of Π_1 with a bottom inference (*cut*) satisfies either*

- (a) Π_1^s is a proper subtree of Π_0 , or
- (b) Π_1^s and Π_0 have the same cut-formula and $\Pi_1^s <_{LRP} \Pi_0$.

$\text{nparRedP_Unf} = \text{nparRedP } "(Unf \text{ ?seq} \text{ ?dtn} = \text{False})"$

$\text{nparRedP_Der} = \text{"nparRedP (Der ?seq ?rule ?dtl) ?dtn} = (\text{ALL } \text{dts} . \text{isSubt ?dtn } \text{dts} \ \& \ \text{isCut } \text{dts} \ \text{-->} \\ \text{isSubt (Der ?seq ?rule ?dtl) } \text{dts} \ \& \ \text{Der ?seq ?rule ?dtl} \ \sim = \ \text{dts} \\ | \ \text{cutForm (Der ?seq ?rule ?dtl)} = \ \text{cutForm } \text{dts} \ \& \\ (\text{dts}, \ \text{Der ?seq ?rule ?dtl}) : \ \text{LRPorder})"$

²This is common: it is well-known in program verification that we sometimes need a stronger induction hypothesis than the obvious one in order to prove that a particular property is a loop-invariant of a while-loop.

In each of Figures 10 and 11, the cuts in the original and transformed derivations satisfy Definition 10.10(b).

Definition 10.11 (c8redP) *A reduction from Π_0 to Π_1 satisfies c8redP if the bottom inference of Π_0 is (cut), and for each subtree Π_1^s of Π_1 whose bottom inference is (cut), either*

- (a) Π_1^s is a proper subtree of Π_0 , or
- (b) the cut-formula of Π_1^s is a proper subformula of the cut-formula of the bottom inference of Π_0 .

```
"c8redP ?dt ?dtn == ALL dts.
  isSubt ?dtn dts & botRule dts = cutr -->
  isSubt ?dt dts & ?dt ~= dts |
  (cutForm dts, cutForm ?dt) : ipsubfml"
```

Note that both Definition 10.11(b) and Definition 10.10(b) imply $(\Pi_1^s, \Pi_0) \in \text{dtorder}$.

We now define a *cut-reduction* (being either parametric or principal) as satisfying one of the properties `nparRedP` and `c8redP`, as well as some further simple conditions which help the proof.

Definition 10.12 (cutReduces) *The derivation tree Π_0 cut-reduces to Π_1 if the following hold simultaneously:*

- (a) Π_0 and Π_1 satisfy either *nparRedP* (for a parametric reduction) or *c8redP* (for a primitive reduction)
- (b) Π_0 and Π_1 have the same conclusion
- (c) the bottom rule of Π_0 is (cut)
- (d) Π_0 and Π_1 are not identical
- (e) Π_1 does not consist solely of an unfinished leaf
- (f) Π_0 has at least one immediate subtree

```
cutReduces_Der = "cutReduces (Der ?seq ?rule ?dtn) ?dtn = (
  (nparRedP (Der ?seq ?rule ?dtn) ?dtn
  | c8redP (Der ?seq ?rule ?dtn) ?dtn)
  & conclDT ?dtn = ?seq & ?rule = cutr
  & (Der ?seq ?rule ?dtn) ~= ?dtn & ~ isUnf ?dtn & ?dtn ~= [] )"

```

Note that for the purposes of the proof of strong normalization, we have defined cut-reduction weakly in that, for example, we do not require that the new derivation tree \mathbf{dtn} be well-formed (via $\mathbf{allDT\ wfb\ dtn}$) or require that it use rules which belong to the calculus (via $\mathbf{allDT\ (frb\ rls)\ dtn}$). However the definition is also strong in that it requires that either $\mathbf{nparRedP}$ or $\mathbf{c8redP}$ holds. Later we will show that the reductions in which we are interested do satisfy $\mathbf{nparRedP}$ or $\mathbf{c8redP}$. The result of this is that we prove strong normalization for a larger class of reductions than we are really interested in. The requirement that Π_0 and Π_1 be distinct is necessary to ensure that null reductions are excluded.

Recall that Definition 10.1 of “reduction” made a forward reference to the notion of “cut-reduction”. Definition 10.12 of “cut-reduction” therefore completes the definition of “reduction”. Given Definitions 10.1 and 10.12(b), we can now assert that any two derivations $(\Pi_1, \Pi_0) \in \mathbf{reduction}$ must have the same conclusion. This is useful in Definition 10.14 later.

10.6 Strong-Normalisation

Lemma 10.3 (dth) *For a given derivation Π_0 , if all derivation trees $\Pi' <_{\mathbf{dt}} \Pi_0$ have the property $\mathbf{snHered}$, then so does Π_0 .*

```
dth = "ALL dt'. (dt', ?dt) : dtorder --> snHered dt'
      ==> snHered ?dt" : thm
```

Proof: Given Π_0 , assume that

- (a) all derivation trees $\Pi' <_{\mathbf{dt}} \Pi_0$ satisfy $\mathbf{snHered}$, and
- (b) all immediate subtrees of Π_0 are strongly normalising.

We have to prove that Π_0 is strongly normalising, whence, by Definition 10.9, it satisfies $\mathbf{snHered}$. Consider possible cases for a reduction of Π_0 , giving a tree Π_1 .

Firstly, suppose that the bottom inference of Π_0 is not cut. Then any reduction is in an immediate (strongly normalising) subtree of Π_0 . So $\Pi_1 <_{\mathbf{sn1}} \Pi_0$ and hence $\Pi_1 <_{\mathbf{dt}} \Pi_0$ by Definition 10.8. All immediate subtrees of Π_1 are equal to or are a reduction of immediate subtrees of Π_0 , which themselves are strongly normalising by assumption (b). All immediate subtrees of Π_1 are therefore strongly normalising. The property $\mathbf{snHered}$ holds for Π_1 by assumption (a), hence the derivation Π_1 is strongly normalising.

Secondly, suppose that the bottom inference of Π_0 is cut, and consider a reduction. This reduction is either parametric or primitive. If it is in an immediate subtree, by the previous argument, Π_1 is strongly normalising. If it is a reduction of the bottom cut, we get the new tree Π_1 whose cuts are either in copies of (strongly normalising) proper subtrees of Π_0 or are cuts

which are smaller in `dtorder` than the bottom cut of Π_0 , by the remark following Definitions 10.10 and 10.11. Thus every subtree Π_1^s of Π_1 (including Π_1 itself) satisfies one of the following:

- (c) Π_1^s is a subtree of a proper subtree of Π_0
- (d) $\Pi_1^s <_{\text{at}} \Pi_0$.

Now a subtree Π_1^s satisfying (c) is strongly normalising by assumption (b), and a subtree Π_1^s satisfying (d) satisfies `snHered`. Thus every Π_1^s satisfies `snHered`. Since Π_1^s is an arbitrary subtree of Π_1 , it follows from Lemma 10.2 that Π_1 is strongly normalising.

In either case Π_1 is strongly normalising. Since Π_1 was obtained via an arbitrary reduction from Π_0 , it follows that Π_0 is strongly normalising. Thus we have that `snHered` holds for Π_0 . **Q.E.D.**

The machine-proof of this last lemma is quite complicated, and a careful examination of it highlights why the definition of `dtorder` needs to be so complex.

Theorem 10.3 (`all_sn`) *Every derivation tree is strongly normalising.*

```
all_snH = "snHered ?dt"      : thm
all_sn  = "strongNorm ?dt"  : thm
```

Proof: By well-founded induction, it follows from Lemma 10.3 that every derivation tree satisfies `snHered`; the result follows from Lemma 10.2. **Q.E.D.**

At this point we have actually shown using Isabelle that a class of reductions which we have defined is strongly normalizing. We need to show that this class of reductions contains the ones in which we are interested.

Given a derivation tree with (*cut*) at its root, the tree can be changed to deal with that particular cut; we call these “cut-reductions”. Following Wansing [40, §4.2], we classify these cut-reductions as principal or parametric. The aim is to show that each such cut-reduction is a reduction. Since we know reductions are strongly-normalising, our cut-reductions would then also be strongly-normalising.

10.7 Making A Cut (Left) Principal

In §7.3 we described how to make a cut (left-) principal. We now define this transformation as a function `mLP`, which will constitute a functional definition of the transformation used to obtain the new derivation tree when making a cut left-principal. We define the following transformations:

```

mLP, mLP' :: "dertree => sequent => dertree => dertree"
mLPs :: "dertree => sequent list => dertree list => dertree list"
mRP, mRP' :: "dertree => sequent => dertree => dertree"
mRPs :: "dertree => sequent list => dertree list => dertree list"

mLP_Der' = "mLP ?dtAY ?seqY (Der ?seqA ?rule ?dtlA) =
  (if strIsFml (succ (conclRule ?rule))
    & succ ?seqA ~= succ ?seqY
  then let ?seqYA = $(antec ?seqY) |- $(succ ?seqA)
        in Der ?seqY cutr
        [mLP' ?dtAY ?seqYA (Der ?seqA ?rule ?dtlA), ?dtAY]
  else mLP' ?dtAY ?seqY (Der ?seqA ?rule ?dtlA))"

mLP'_Der = "mLP' ?dtAY ?seqY (Der ?seqA ?rule ?dtlA) =
  (let sub = newSub (Der ?seqA ?rule ?dtlA) ?seqY;
      premsinst = premsRule (?ruleSubst sub ?rule)
  in Der ?seqY ?rule (mLPs ?dtAY premsinst ?dtlA))"

```

The arguments to `mLP` are

dtAY: a derivation tree with root $A \vdash Y$, such as the right subtree Π_R of the left tree given in Figure 7

seqY: a sequent such as the sequent $X \vdash Y$ at the bottom of the trees shown in Figure 7

Der seqA rule dtlA: a tree with root `seqA`, such as the tree with root $X \vdash A$ which forms the left subtree of the left tree given in Figure 7

where sequent `seqA` will usually contain occurrences of A in one or more succedent position(s), and sequent `seqY` is `seqA` with zero, one or more of these occurrences of A changed to Y . The result of `mLP` is a new derivation tree, with root sequent `seqY` whose new cuts (on A) are left-principal, such as the right tree given in Figure 7. Thus, the function `mLP` can be used to transform the left derivation tree from Figure 7 into the right derivation tree in Figure 7.

The function `mLPs` takes a list of sequents instead of just one sequent for its second argument, a list of derivation trees instead of just one tree for its third argument, and returns a corresponding list of appropriately transformed derivation trees.

The definition of `mLP` consists of two parts: an “if then” part and an “else” part. We use the derivations shown in Figure 7 as a running example.

The “if then” part of `mLP` handles the case where the displayed occurrence of A in the succedent of the root `seqA` of `Der seqA rule dtlA` is the principal formula of rule `rule`: for example where `seqA` is $Z[A] \vdash A$ and `rule` is (intro- A) in the left hand derivation shown in Figure 7. This part

makes `seqY` the conclusion of a new (*cut*) rule application: `seqY` is $Z[Y] \vdash Y$ in the right derivation in Figure 7. The right premise of the new cut is the tree `dtAY`: the tree Π_R deriving sequent $A \vdash Y$ in Figure 7. The left subtree of this new cut is a derivation tree with end-sequent `seqYA`, where `seqYA` is built from the antecedent of `seqY` and the succedent of `seqA`: the sequent `seqYA` is $Z[Y] \vdash A$ in Figure 7. The immediate subtrees of this left subtree are obtained via the sister function `mLP'`: the derivation $\Pi'[Y]$ from Figure 7.

The function `mLP'` computes the substitution `sub` required to instantiate the conclusion of `rule` to its second argument `seqY`: in our running example `rule` is (*intro-A*) and `seqY` is $Z[Y] \vdash A$. It then computes the new premises of this instance of `rule`: the premises of (*intro-A*) in the right hand derivation from Figure 7. It constructs a new derivation with end-sequent `seqY` and recursively calls function `mLPs` to appropriately transform the subtrees `dtlA` of `Der seqA rule dtlA` so they now derive these new premises of `rule`: the derivation $\Pi'[Y]$ from Figure 7.

The function `mLP'` is also used in the “else” part of `mLP` since, in this case, `mLP` is being applied to a derivation tree whose end-sequent `seqA` contains parametric ancestor occurrences of A which are not principal formulae of `rule`. For example, when transforming $\Pi_L[A]$ into $\Pi_L[Y]$ from Figure 7. So all we need is a function like `mLP'` that merely computes and applies appropriate substitutions, rather than one like `mLP` that inserts new cuts.

The function `mLP'` uses the function `newSub`, defined as below:

```
newSub_Der = "newSub (Der ?seqA ?rule ?dtlA) ?seqY =
  stepSub (ruleSubOf (Der ?seqA ?rule ?dtlA))
    (conclRule ?rule) ?seqY"
```

Here, `ruleSubOf` returns a substitution which takes `rule` to the instance actually used at the bottom of `Der seqA rule dtlA`. To explain `stepSub (fs, suba) seq seqY`, let `(fs, suby)` be a substitution which takes `seq` to `seqY`, and let `sub` be a structure variable substitution which acts as `suby` on the structure variables in `seq`, and as `suba` on other structure variables. Then `stepSub (fs, suba) seq seqY` is `(fs, sub)`. Consequently, in `mLP'`, the list `premsinst` consists of the premises of `rule` instantiated with the substitution returned by `ruleSubOf` but modified so that the conclusion of `rule` is the argument `seqY` of `mLP'`.

The definition of `mRP` is similar: given derivation trees whose root sequents are `seqA` (A in antecedent positions) and $Y \vdash A$, and given `seqY`, which is `seqA` with some occurrences of A in antecedent positions changed to Y , `mRP` returns a derivation tree whose root sequent is `seqY`, and whose new cuts are right-principal.

Proving these necessary results is complicated by the fact that it is necessary to show that the substitution `suby` in the definition of `stepSub` exists, and that although such a substitution is not unique, the choice does not matter.

Recall that the parametric reduction involves tracing up the derivation tree, from a premise of the cut being reduced, to all points where the parametric ancestors of the cut-formula A are introduced. As Wansing describes [40, p. 49] this portion of the derivation tree cannot contain another cut: if it did, an infinite sequence of reductions *would* be possible.

Definition 10.13 (ncLP) *The exact definition of ncLP is given below, but the “intention” of the definition is that ncLP seqY dtA holds iff calculating the tree mLP dtAY seqY dtA does not involve traversing another cut; ncRP has an analogous meaning.*

```

ncLP  :: "sequent => dertree => bool"
ncLPs :: "sequent list => dertree list => bool"
ncRP  :: "sequent => dertree => bool"
ncRPs :: "sequent list => dertree list => bool"

ncLP_Der = "ncLP ?seqY (Der ?seqA ?rule ?dtlA) = (
  ?seqA = ?seqY | ?rule ~= cutr &
  (if strIsFml (succ (conclRule ?rule)) &
    succ ?seqA ~= succ ?seqY
  then
    let seqYA = Sequent (antec ?seqY) (succ ?seqA) ;
        sub = newSub (Der ?seqA ?rule ?dtlA) seqYA ;
        premsinst = premsRule (ruleSubst sub ?rule)
    in (ncLPs premsinst ?dtlA)
  else
    let sub = newSub (Der ?seqA ?rule ?dtlA) ?seqY ;
        premsinst = premsRule (ruleSubst sub ?rule)
    in (ncLPs premsinst ?dtlA) ))"
ncLP_Unf "ncLP ?seqY (Unf ?seqA) = True"

ncLPs_Nil = "ncLPs ?prems [] = (?prems = [])"
ncLPs_Cons = "ncLPs ?prems (?h # ?t) = (?prems ~= [] &
  ncLP (hd ?prems) ?h & ncLPs (tl ?prems) ?t)"

```

Both ncLP and mLP use the same “if then else” construct and ncLP mimics the substitutions that are computed in mLP, as long as the rule is not the cut rule: thus ncLP returns “False” when it reaches a cut rule since this indicates that we would have to cross a cut rule application in tracing some parametric ancestor of A to where it becomes principal.

The other difference between ncLP and mLP is that ncLP will return “True” if $?seqA = ?seqY$: that is, if the end-sequent seqA is already equal to the intended target of mLP. At such sequents, calculating mLP dtAY seqY dtA clearly does not require proceeding further up the tree. But mLP continues to traverse the whole tree in this case since it is required to return a

new derivation rather than just a boolean value. The fact that `mLP` does not actually effect any change in this case is guaranteed by the following lemma.

Lemma 10.4 (sameConcLP) *If $seqY$ is equal to the conclusion of a well-formed derivation tree dtA , then $mLP\ dtAY\ seqY\ dtA = dtA$.*

```
sameConcLP = "allDT wfb ?dtA ==>
  mLP ?dtAY (conclDT ?dtA) ?dtA = ?dtA" : thm
```

Thus the actual and “intended” definitions are equivalent for a well-formed derivation tree, which are the only ones we are really interested in.

We now have the ingredients to prove that we can transform an arbitrary cut on A into one or more left-principal cuts on A as described in Figure 7.

Theorem 10.4 (makeCutmLP2) *Suppose we are given a valid derivation tree $dtAY$ deriving $A \vdash Y$, and a valid derivation tree dtA , and a sequent $seqY$ obtained from the conclusion of dtA by (possibly) replacing some succedent occurrences of A by Y , and a tree dtY obtained by transforming dtA using mLP without crossing any cuts. Then, for every subtree dts of dtY which ends in a cut, one of the following holds:*

- dts is a subtree of the original tree dtA
- dts is a subtree of the original tree $dtAY$
- dts ends with a left principal cut on A , and the right subtree of this cut is the original $dtAY$, and the conclusion of dtY is the given $seqY$, and dtY is well-formed and uses rules from rls .

```
makeCutmLP2 = "[| C345 ?rules; cutr : ?rules;
  allDT wfb ?dtAY; allDT (frb ?rules) ?dtAY;
  conclDT ?dtAY = (?A |- $?Y);
  allDT wfb ?dtA; allDT (frb ?rules) ?dtA;
  (conclDT ?dtA, ?seqY) : seqrep True (Structform ?A) ?Y;
  ?dtY = mLP ?dtAY ?seqY ?dtA;
  ncLP ?seqY ?dtA   |]
==> (ALL dts.
  isSubt ?dtY dts & isCut dts -->
  isSubt ?dtA dts |
  isSubt ?dtAY dts |
  cutIsLP ?A dts & hd (tl (nextUp dts)) = ?dtAY) &
  conclDT ?dtY = ?seqY & allDT wfb ?dtY &
  allDT (frb ?rules) ?dtY" : thm
```

Theorem 10.4 essentially says the following.

Corollary 10.1 *Given a valid derivation tree $dtAY$ deriving $A \vdash Y$, and a valid derivation tree dtA deriving $seqA$, and given $seqY$ where $seqY$ and $seqA$ are the same except (possibly) that A in a succedent position in $seqA$ is replaced by Y in $seqY$, there is a valid derivation tree dtY , obtained from $seqA$ via mLP without crossing any cuts, which has end-sequent $seqY$, and whose new cuts are all left-principal on A .*

We now need to show that the transformation performed by mLP are in the class of reductions that are strongly normalising.

The next result says that if predicate $ncLP$ is satisfied, then a parametric reduction which uses mLP satisfies the predicate $nparRedP$. We assume an initial cut as in Figure 7 with conclusion $X \vdash Y$ and cut-formula A .

Theorem 10.5 (pRedLP2) *Consider a parametric reduction of a cut which proceeds by transforming the left subtree (to change its conclusion from $X[A] \vdash A$ to $X[Y] \vdash Y$), using the function mLP . Assume that the subtree satisfies the condition $ncLP$ (in effect, that the transformation can be performed without traversing another cut). Also assume the cut is not already left-principal. Then the reduction satisfies the condition $nparRedP$.*

```
pRedLP2 = "[| C345 ?rules;
  ?dtY = mLP ?dtAY ?seqY ?dtA;
  ?dt = Der ?seqY cutr [?dtA, ?dtAY];
  ~ rootIsSucP ?dtA; valid ?rules ?dt;
  ncLP ?seqY ?dtA |]
==> nparRedP ?dt ?dtY & valid ?rules ?dtY &
  conclDT ?dtY = conclDT ?dt" : thm
```

The condition $\sim \text{rootIsSucP?dtA}$ means in effect that the cut is not already left-principal. It is required as a condition of the theorem to avoid null reductions, because if the cut is already left-principal, then $?dt$ and $mLP ?dtAY ?seqY ?dtA$ are equal.

The following result is similar, but says that the parametric reduction is a cut-reduction: that is, it satisfies the predicate $cutReduces$.

Theorem 10.6 (pRedLP3) *If we are given a valid tree dt whose bottom-most rule is a cut with conclusion $seqY$, and the cut is not left-principal, and we obtain a new tree dtY by calculating mLP of the left subtree dtA of dt without crossing any cuts, then the reduction from dt to dtY is a cut-reduction, and dtY is also a valid tree.*

```
pRedLP3 = "[| C345 ?rules;
  ?dt = Der ?seqY cutr [?dtA, ?dtAY];
  valid ?rules ?dt; ~ rootIsSucP ?dtA;
  ?dtY = mLP ?dtAY ?seqY ?dtA; ncLP ?seqY ?dtA |]
==> cutReduces ?dt ?dtY & valid ?rules ?dtY" : thm
```

Once again, the requirement $\sim \text{rootIsSucP } ?dtA$ that the cut be not already left-principal is simply to avoid null reductions.

There are analogous theorems pRedRP2 , pRedRP3 , makeCutmRP , which guarantees that we can make a cut right-principal, and consequently another theorem makeCutmLRP which guarantees that we can make a cut (left and right) principal. The associated functions also produce derivations that satisfy the properties required of them.

Thus every parametric move corresponds to a reduction which is in the class of strongly normalising reductions.

10.8 Dealing With Principal Cuts

Given a derivation (tree) with one principal cut, such as in Figure 5(a), Belnap’s condition (C8) on the rules of a Display Calculus ensures that the given derivation can be transformed into one whose cuts are on smaller formulae. For example, the derivation in Figure 5(a) is transformed into the one in Figure 5(b) where $(cs1)$, $(cs2)$, $(\overline{cs1})$ and $(\overline{cs1})$ are two of the display postulates and their inverses respectively.

As discussed in §7.2, a (left and right) principal cut can be transformed into another derivation of the same end-sequent, using “new” cuts (if any) on formulae which are strict subformulae of the original cut-formula. Here we prove theorems for the situation where the original derivation tree may contain other cuts, and we use the property c8redP to express that the “new” cuts are on subformulae.

Here is the resulting theorem for \forall : there is an analogous theorem for every logical connective and logical constant.

Theorem 10.7 (orvC8W) *Assume we are given a valid derivation tree dt . Assume that if the bottom rule of dt is (cut), then the cut is principal and its cut-formula is $A \vee B$. Then there is a valid derivation tree dtn with the same conclusion as dt , such that $\text{c8redP } dt \ dtn$ holds.*

```
orvC8W = "[| cutIsLRP (?A v ?B) ?dt; valid rls ?dt |]
  ==> EX dtn. valid rls dtn & conclDT dtn = conclDT ?dt &
    c8redP ?dt dtn" : thm
```

Here, $(\text{sfml}, \text{fml}) : \text{ipsubfml}$ means that sfml is an immediate subformula of fml . The definition of $\text{c8redP } dt \ dtn$ (see Definition 10.11) ensures that every subtree dt_s of dtn which has a cut at its root is either a proper subtree of the original dt , or is a cut on an immediate subformula of the cut at the bottom of dt . Thus $\text{c8redP } dt \ dtn$ guarantees that all *new* cuts in the transformed tree dtn are on immediate subformulae of the cut-formula of dt .

We have also shown that the primitive reductions described above satisfy c8redP , and are in fact cut-reductions. For each logical connective we

considered the reduction analogous to that in Figures 5(a) and 5(b) (which was used to prove `orc8w` and analogous results), and showed that it satisfies `c8redP`. These results (one for each logical connective) were combined to give Theorem 10.8.

Theorem 10.8 (`c8sn_rls`) *If the bottom-most rule of a valid derivation tree Π_0 is a (left and right) principal cut, then there exists a valid derivation tree Π_1 with the same conclusion as Π_0 and such that `textttc8redP` holds for Π_0 and Π_1 .*

```
"c8sn ?rules == ALL form dt.
  cutIsLRP form dt & valid ?rules dt --> (EX dtn.
    conclDT dtn = conclDT dt & c8redP dt dtn & valid ?rules dtn)"
```

```
c8sn_rls = "c8sn_rls" : thm
```

Thus every principal move corresponds to a reduction which is in the class of strongly normalising reductions.

10.9 Cut-Elimination Via Strong Normalisation

We now proceed to complete a proof of cut-elimination using our proof of strong normalization. We use convenient abbreviations to describe the sorts of trees and reductions in which we are interested. Recall that a *valid* derivation tree is one which is well-formed, uses rules from `rls`, and has no unfinished leaves.

Definition 10.14 (`validRed`) *A valid reduction is a reduction of tree Π_0 to tree Π_1 , where Π_1 is a valid tree (which has the same conclusion as Π_0).*

```
validRed_def = "validRed ?rules ==
  {(dtn, dt). valid ?rules dtn & (dtn, dt) : reduction}" : thm
```

Lemma 10.5 *A valid reduction of a subtree is a valid reduction of the whole tree.*

Proof: A reduction of a subtree is a reduction of the whole tree, by Definition 10.1 of reduction. Since a reduction of a subtree has the same conclusion as the original sub-tree, a valid reduction of the subtree cannot destroy the validity of the original tree (nor change the end-sequent of the original tree).

Q.E.D.

Theorem 10.9 *Parametric moves give valid reductions.*

Proof: By Theorem 10.6 we know that applying a parametric move to a tree with a non-left-principal cut at the bottom gives a cut-reduction to a valid tree with the same end-sequent, and hence a valid reduction. The theorem analogous to Theorem 10.6 which uses `pRedRP3` handles any non-right-principal cuts. If the original tree is actually a sub-tree of a larger tree, then such a cut-reduction is a valid reduction of the larger tree by Lemma 10.5. **Q.E.D.**

Theorem 10.10 (cutExRed) *For any valid tree with a single cut at the bottom there exists a cut-reduction to another valid tree with the same conclusion, using only parametric and principal moves.*

```
cutExRed = "[| C345 ?rules; c8sn ?rules;
  isCut ?dt; valid ?rules ?dt; allNextDTs (cutOnFmls { }) ?dt |]
  ==> EX dtn. cutReduces ?dt dtn & valid ?rules dtn" : thm
```

Proof: The proof uses only `pRedLP3` and `pRedRP3` and the definition of `c8sn`, guaranteeing that the reductions are restricted to parametric and principal moves only. Note that `conclDT dtn = conclDT dt` is implied by `cutReduces dt dtn`. **Q.E.D.**

Given any derivation tree containing a cut, we can apply the theorem above to a top-most cut.

Theorem 10.11 (ExRed) *For any valid tree which contains a cut, there is available at least one valid reduction.*

```
ExRed = "[| "[| C345 ?rules; c8sn ?rules;
  ~ allDT (Not o isCut) ?dt; valid ?rules ?dt |]
  ==> EX dtn. (dtn, ?dt) : validRed ?rules" : thm
```

Proof: The proof uses Theorem 10.10. **Q.E.D.**

The outline of the proof from here is clear: since every tree with a cut admits at least one valid reduction, and parametric and principal moves give valid reductions, and there is no infinite sequence of valid reductions, an effective cut-elimination procedure is to repeatedly perform any sequence of principal and parametric reductions until no reduction is possible. Below, the notation `^*` denotes transitive closure.

From here, we repeatedly perform valid reductions. The notation `^*` denotes transitive closure, and `allDT (Not o isCut) dtn` means that `dtn` is cut-free since `o` denotes composition of functions in Isabelle/HOL. Finally, Theorem 10.14 states our result in terms of cut-admissibility.

Theorem 10.12 (`validRed_min`) *For any derivation tree Π there exists a tree Π^r , obtained from Π by repeated valid reductions, such that Π^r cannot be further validly reduced.*

```
validRed_min = "EX dtn. (dtn, ?dt) : (validRed ?rules)^* &
  ~ (EX dts. (dts, dtn) : validRed ?rules)" : thm
```

Proof: Every tree with a cut admits at least one valid reduction, and parametric and principal moves give valid reductions, and there is no infinite sequence of valid reductions. So repeatedly perform any sequence of (principal and parametric) reductions until no reduction is possible. **Q.E.D.**

Below, `allDT (Not o isCut) dtn` means that `dtn` is cut-free since `o` stands for composition of functions in Isabelle/HOL.

Theorem 10.13 (`redNoCut`) *For any valid tree Π , there exists a cut-free valid tree Π^r , obtained from Π by repeated valid reductions, such that Π^r has the same conclusion as Π .*

```
redNoCut = "[| C345 ?rules;
  c8sn ?rules; valid ?rules ?dt |]
  ==> EX dtn. allDT (Not o isCut) dtn &
  conclDT dtn = conclDT ?dt & valid ?rules dtn &
  (dtn, ?dt) : (validRed ?rules)^*" : thm
```

Corollary 10.2 *Any sequence of principal and parametric moves starting from some derivation Π containing cuts, eventually terminates with a cut-free derivation Π^r that has the same conclusion as Π .*

Proof: In Theorem 10.13, starting from Π , choose a sequence of parametric and principal reductions only. We know that at least one such is applicable to any Π containing a cut, to its topmost cut. Each reduction in the sequence is a valid reduction by Theorem 10.9 so the sequence produces only valid trees with the same end-sequent as Π . Since valid reductions are reductions, they are strongly normalising by Theorem 10.3, hence this sequence must terminate with some valid derivation Π^r , to which no reduction is applicable. This is only possible if Π^r is cut-free. **Q.E.D.**

We therefore have cut-elimination, proved using a strong normalization approach.

Although an Isabelle proof cannot guarantee that we have encoded Wansing's definition of reduction correctly, this result shows that we have defined a class of reductions which is large enough to permit cut-elimination, while being small enough to be strongly normalizing.

Theorem 10.14 (`cutElim_SN`) *If a sequent can be derived using rules `rls`, then it can be derived from those rules omitting `(cut)`.*

```

cutElim_SN = "[| C345 ?rules; c8sn ?rules;
  IsDerivableR ?rules {} ?concl |]
  ==> IsDerivableR (?rules - {cutr}) {} ?concl" : thm

```

As usual in sequent calculi, the normal form is not unique.

11 Conclusion and Further Work

We have formulated a “deep” embedding of the Display Calculus for Relation Algebra $\delta\mathbf{RA}$ in Isabelle/HOL and used it for reasoning about derivability in $\delta\mathbf{RA}$ rather just mimicking derivations in $\delta\mathbf{RA}$, as in a previous “shallow” embedding [8]. In Section 5.5 we have proved “transitivity” results like `IsDerivableR_trans` and `IsDerivableR_deriv` about the composition of $\delta\mathbf{RA}$ derivations. Similar results were simply asserted (incorrectly) in another reported mechanized formalization of provability for first-order logic, due to their difficulty [26].

We tried to formalise Wansing’s proof of strong normalisation for display calculi, but found that it contained an error. We have therefore proved strong normalisation for the display calculus $\delta\mathbf{RA}$ using a different method, based on a rather complex ordering which we show to be well-founded. This has been a considerable effort, and could not have been achieved without the complementary features (found in Isabelle) of extensive and powerful tactics, and a powerful programming language interface available to the user.

Belnap’s theorem is expressed to apply to any Display Calculus satisfying his conditions. To prove his theorem in that form would require modelling an arbitrary Display Calculus, with generalised rules for arbitrary sets of structural and logical connectives. A theoretical framework for such generalised rules can be found in [17]. In a sense, this would require a “deeper” embedding still. For, in our first implementation [8], we set up the specific connectives and rules of $\delta\mathbf{RA}$ in Isabelle, and used Isabelle proofs as the $\delta\mathbf{RA}$ -derivations. In the present implementation, we again set up the specific connectives and rules of $\delta\mathbf{RA}$, although we set up data structures to model arbitrary derivations. However much of the work in the present implementation is generic with respect to the set of rules, subject to the rules satisfying the properties (C3) to (C5) and (C8). To prove the generic Belnap theorem, we would need to set up the necessary structures to model arbitrary sets of connectives and rules from [16].

The most important disadvantage of our approach is the inability to easily produce a program for cut-elimination from our Isabelle/HOL proofs, even when our proofs mimic a programming style (see §7.5). We now discuss possible further work to address this issue.

In [9] we described a shallow embedding of $\delta\mathbf{RA}$ into Isabelle/Pure. We also described an implementation of $\delta\mathbf{RA}$ in Twelf, which had the feature that deriving a sequent produced a term containing the derivation tree for

that sequent. If we think of that Twelf implementation as a “shallow embedding”, we may then ask whether a “deep embedding” in Twelf, and a similar proof of cut-elimination using it, would produce a term representing that proof. That term might then provide a function to perform cut-elimination. Note, however, that proof by structural induction on the types `dertree` and `formula` (types defined using Isabelle/HOL’s `datatype` facility) were crucial. To perform the proof in another system we would still need to reproduce these types.

Results like `IsDerivableR_trans` and `IsDerivableR_deriv` are general results about the structure of derivations, closely resembling facilities available in Isabelle: namely, successive refinement of subgoals, and use of a previously proved lemma. A higher order framework allows us to reason about higher and higher meta-levels, like the `IsDerivable` relation itself, without invoking explicit “reflection principles” [18]. This is the topic of future work.

12 Appendix: Kracht’s Formulation of Belnap’s Conditions

For every sequent rule Belnap [5, page 388] first defines the following notions: in an application Inf of a sequent rule (ρ) , “constituents occurring as part of occurrences of structures assigned to structure-variables are defined to be **parameters** of Inf ; all other constituents are defined as **nonparametric**, including those assigned to formula-variables. Constituents occupying similar positions in occurrences of structures assigned to the same structure-variable are defined as **congruent** in Inf ”. The eight (actually seven) conditions shown below are from [21]:

- (C1) Each formula which is a constituent of some premiss of a rule ρ is a subformula of some formula in the conclusion of ρ .
- (C2) Congruent parameters are occurrences of the same structure.
- (C3) Each parameter is congruent to at most one constituent in the conclusion. Equivalently, no two constituents of the conclusion are congruent to each other.
- (C4) Congruent parameters are either all antecedent parts or all succedent parts of their respective sequent.
- (C5) If a formula is non-parametric in the conclusion of a rule ρ , it is either the entire antecedent, or the entire succedent. Such a formula is called a **principal** formula.
- (C6/7) Each rule is closed under simultaneous substitution of arbitrary structures for congruent parameters.
- (C8) If there are inference rules ρ_1 and ρ_2 with respective conclusions $X \vdash P$ and $P \vdash Y$ with P principal in both inferences (in the sense of C5), and if (cut) is applied to yield $X \vdash Y$ then, either $X \vdash Y$ is identical to $X \vdash P$ or to $P \vdash Y$; or it is possible to pass from the premisses of ρ_1 and ρ_2 to $X \vdash Y$ by means of inferences which are instances of (cut) where the cut-formula is always a proper subformula of P . If P satisfies the “if” part of this condition it is known as a “matching principal constituent”.

References

- [1] A A Adams. A formalisation of weak normalisation (with respect to permutations) of sequent calculus proofs. *LMS Journal of Computational Mathematics*, 3:1–26, 2000.
- [2] Thorsten Altenkirch. A formalization of the strong normalization proof for system F in LEGO. In Marc Bezem and Jan Frisco Groote, editors, *Typed Lambda Calculi and Applications*, LNCS 664, pages 13–28. Springer, 1993.
- [3] B Barras B and B Werner. Coq in coq. <http://pauillac.inria.fr/~barras/download/coqincoq.ps.gz>.
- [4] David A. Basin and Seán Matthews. Structuring metatheory on inductive definitions. *Information and Computation*, 162(1-2):80–95, 2000. <http://www.informatik.uni-freiburg.de/~basin/>.
- [5] N D Belnap. Display logic. *Journal of Philosophical Logic*, 11:375–417, 1982.
- [6] Mirjana Borisavljevic, Kosta Dosen, and Zoran Petric. On permuting cut with contraction. *Mathematical Structures in Computer Science*, 10:99–136, 2000.
- [7] A Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [8] J Dawson and R Goré. A mechanised proof system for relation algebra using display logic. In *JELIA98: Proceedings of the European Workshop on Logic in Artificial Intelligence*, LNAI 1489:264-278. Springer, 1998. <http://discus.anu.edu.au/~jeremy/dra/dra-files/>.
- [9] Jeremy E Dawson and Rajeev Goré. Embedding display calculi into logical frameworks: Comparing Twelf and Isabelle. In Colin Fidge, editor, *Proc. CATS 2001: The Australian Theory Symposium*, volume 42 of *Electronic Notes in Theoretical Computer Science*, pages 89–103, <http://www.elsevier.nl/gej-ng/31/29/23/68/show/Products/notes/cover.htm>, 2001. Elsevier.
- [10] Jeremy E. Dawson and Rajeev Goré. Machine-checked strong normalisation for display logic. Technical report, Computer Sciences Laboratory and Dept of Computer Science, Australian National University, December 2001. <http://arp.anu.edu.au/~rpg/CutElim/>.
- [11] Jeremy E. Dawson and Rajeev Goré. Formalised cut admissibility for display logic. In V. A Carreno, C. A. Munoz, and S. Tahar, editors,

- TPHOLs02: Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*, volume LNCS, pages 131–147. Springer, 2002.
- [12] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3), September 1992.
- [13] R A Elmasri and S B Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, 2nd edition, 1994.
- [14] M J C Gordon and T F Melham. *Introduction to HOL: a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
- [15] R Goré. Cut-free display calculi for relation algebras. In D van Dalen and M Bezem, editors, *CSL96: Selected Papers of the Annual Conference of the European Association for Computer Science Logic*, volume LNCS 1258, pages 198–210. Springer, 1997. <http://arp.anu.edu.au/~rpg>.
- [16] R Goré. Gaggles, Gentzen and Galois: How to display your favourite substructural logic. *Logic Journal of the Interest Group in Pure and Applied Logic*, 6(5):669–694, 1998. <http://arp.anu.edu.au/~rpg>.
- [17] R Goré. Substructural logics on display. *Logic Journal of the Interest Group in Pure and Applied Logic*, 6(3):451–504, 1998. <http://arp.anu.edu.au/~rpg>.
- [18] J Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre, 1995. <http://www.abo.fi/~jharriso/>.
- [19] Alain Heuerding. *Sequent Calculi for Proof Search in some Modal Logics*. PhD thesis, Institute for Applied Mathematics and Computer Science, University of Berne, Switzerland, 1998.
- [20] Paul Taylor Jean-Yves Girard and Yves Lafont. *Proofs and Types*. Cambridge Tracts In Theoretical Computer Science. Cambridge University Press, 1989.
- [21] M Kracht. Power and weakness of the modal display calculus. In H Wansing, editor, *Proof Theory of Modal Logics*, pages 92–121. Kluwer, 1996.
- [22] R D Maddux. Introductory course on relation algebras, finite-dimensional cylindric algebras, and their interconnections. In Monk J. D. Andreka, H. and I. Nemeti, editors, *Algebraic Logic*, volume 54 of

- Colloquia Mathematica Societatis Janos Bolyai*, pages 361–392. North-Holland, Amsterdam, 1991.
- [23] S Matthews. A theory and its metatheory in FS_0 . In Dov M Gabbay, editor, *What is a logic system?*, pages 329–354. Oxford University Press, 1994.
- [24] S. Matthews, A. Smaill, and D. Basin. Experience with FS_0 as a framework theory. In G Huet and G Plotkin, editors, *Logical Environments*, pages 61–82. Cambridge University Press, 1993. <http://www.informatik.uni-freiburg.de/~basin/>.
- [25] Sean Matthews. Implementing FS_0 in Isabelle: adding structure at the metalevel. In J Calmet and C Limongelli, editors, *Proc. Disco'96*. Springer, 1996. <http://www.informatik.uni-freiburg.de/~basin/>.
- [26] Anna Mikhajlova and Joakim von Wright. Proving isomorphism of first-order proof systems in HOL. In J Grundy and M Newey, editors, *Theorem Proving in Higher-Order Logics*, LNCS 1479, pages 295–314. Springer, 1998.
- [27] A Mili. A relational approach to the design of deterministic programs. *Acta Informatica*, 20:315–328, 1983.
- [28] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle's logics: Hol. Technical report. 15 February 2001, `doc/logics-HOL.dvi` in the Isabelle distribution.
- [29] Lawrence C. Paulson. The isabelle reference manual. Technical report. 15 February 2001, `doc/ref.dvi` in the Isabelle distribution.
- [30] Lawrence C. Paulson. Isabelle's logics. Technical report. 15 February 2001, `doc/logics.dvi` in the Isabelle distribution.
- [31] Frank Pfenning. Structural cut elimination. In *Proc. LICS 94*, 1994. <http://www-2.cs.cmu.edu/~fp/publications.html>.
- [32] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, School of Computer Science, Carnegie Mellon University, USA, 1994. <http://www-2.cs.cmu.edu/~fp/publications.html>.
- [33] Frank Pfenning. A structural proof of cut elimination and its representation in a logical framework. Technical Report CMU-CS-94-218, School of Computer Science, Carnegie Mellon University, USA, 1994. <http://www-2.cs.cmu.edu/~fp/publications.html>.

- [34] Frank Pfenning. Logical frameworks. In A Robinson and A Voronkov, editors, *Handbook of Automated Reasoning Volume*, volume 1, chapter 17, pages 1063–1148. The MIT Press, North Holland, 2001.
- [35] D Pym and J Harland. A uniform proof-theoretic investigation of linear logic programming. *J. of Logic and Computation*, 4:175–207, 1994.
- [36] C Schürmann. *Automating the Meta Theory of Deductive Systems*. CMU-CS-00-146, Dept. of Comp. Sci. , Carnegie Mellon University, USA, 2000.
- [37] M. E. Szabo, editor. *The collected papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.
- [38] A Troelstra and H Schwichtenberg. *Basic Proof Theory*. Number 43 in Cambridge Tracts In Theoretical Computer Science. Cambridge University Press, 1996.
- [39] H Wansing. Strong cut-elimination in display logic. *Reports on Mathematical Logic*, 29:117–131, 1995 (published 1996).
- [40] Heinrich Wansing. *Displaying Modal Logic*, volume 3 of *Trends in Logic*. Kluwer Academic Publishers, Dordrecht, August 1998.