Machine-checked Strong Normalisation 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

15 January 2002

Abstract

We use a deep embedding of the display calculus for relation algebras $\boldsymbol{\delta}$**RA** in the logical framework Isabelle/HOL to formalise a machine-checked proof of strong normalisation and cut-elimination for $\boldsymbol{\delta}$**RA**. Our formalisation generalises easily to handle other display calculi and can serve as a basis for formalised proofs of strong normalisation for the classical and intuitionistic versions of a vast range of substructural logic like the Lambek calculus, linear logic, relevant logic, BCK-logic, and their modal extensions. As far as we are aware, this is the first full formalisation of a strong normalisation result using a logical framework.

Keywords: display logic, cut elimination, strong normalisation, machine-checked proof theory

1 Introduction

Display Logic [**?**] is a generalised sequent framework for non-classical logics, based on Gentzen's sequent calculus [**?**]. 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' commas 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.

*Supported by an Australian Research Council Large Grant
†Supported by an Australian Research Council QEII Fellowship

Due to their generality, display calculi provide an extremely elegant sequent framework, applicable to many (classical and non-classical) logics in a uniform way [?, ?]. 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 [?].

In [?], Nuel Belnap gives a 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. Although Belnap uses a double induction, he does not set out the proof in the traditional way involving a cut rank and cut degree a là Gentzen [?]. 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.

To settle the question, Heinrich Wansing gave a proof of strong-normalisation for the cut-elimination procedure for a particular display calculus in [?]. As is traditional in proofs of strong normalisation, Wansing first defines certain transformations on derivations, and 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. The highest cuts, with no cuts above them, are always amenable to these transformations, so strong normalisation implies the admissibility of cut.

We found it difficult to understand Wansing's proof in its entirety so we decided to formalise it for the display calculus $\boldsymbol{\delta}$**RA** for relation algebras from [?], and check it using a logical framework. The choice of the display calculus is not important, we just happen to be very familiar with it. In [?] 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 ...". We were unable to see how the generalisation would proceed.

We therefore developed our own proof of strong normalisation for the display calculus $\boldsymbol{\delta}$**RA**. The work required approximately 9 man-months, required approximately 1600 lines of Isabelle/HOL theory files and 8500 lines of ML proof files, and takes approximately 35 minutes to fully type-check on a ZZZ machine. Here, we describe how we proved the strong normalisation result for $\boldsymbol{\delta}$**RA**. As far as we are aware, this is the first full formalisation of a strong normalisation result using a logical framework.

The paper is set out as follows. In the rest of this section we describe relation algebras, the display calculus $\boldsymbol{\delta}$**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 2 we describe an encoding of $\boldsymbol{\delta}$**RA** logical constants, logical connectives, formulae, structures, sequents, rules, derivations and derived rules into Isabelle/HOL. In Section 3 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 4 we discuss the problems we had in formalising Wansing's proof of strong normalisation. In Section 6 we describe our own proof of strong normalisation for the display calculus $\boldsymbol{\delta}\mathbf{RA}$. Section 7 presents conclusions.

Acknowledgements: This work has been supported by the Australian Research Council.

1.1 Relation Algebras and δRA

Relation algebras [**?**] 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 [**?**] and of the specification and proof of correctness of programs, particularly in the style of Mili [**?**]. The following grammar defines the syntax of relation algebras where the elements of the top line are Boolean and the elements of the bottom line are their relational analogues:

$$\begin{array}{rcl} A\ B & ::= & p_i \mid \top \mid \bot \mid \neg A \mid A \wedge B \mid A \vee B \mid A \supset B \\ & & \mid \mathbf{1} \mid \mathbf{0} \mid\sim A \mid\smile A \mid A + B \mid A \circ B \mid A \rightarrow B \end{array}$$

A display calculus for relation algebras called $\boldsymbol{\delta}\mathbf{RA}$ can be found in [**?**]. Sequents of $\boldsymbol{\delta}\mathbf{RA}$ 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:

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

Thus, whereas Gentzen's sequents $\Gamma \vdash \Delta$ assume that Γ and Δ are comma-separated lists of formulae, $\boldsymbol{\delta}\mathbf{RA}$-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 ($\mathbf{LK}$-$\vdash \vee$), shown below left, is typical of Gentzen's sequent calculi like $\mathbf{LK}$, while the rule ($\boldsymbol{\delta}\mathbf{RA}$-$\vdash \vee$) shown on the right is typical of display calculi:

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

To use this display calculus rule downwards on a sequent $X' \vdash Y'$ containing an occurrence of the structure $(P \mathbin{;} Q)$, everything other than (P, Q) must be moved to the left of $\vdash$ creating the complex structure X, thereby displaying the structure $(P \mathbin{;} Q)$ as the whole of the right-hand side. There are 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 this rule is to "rewrite" some occurrence of $(P \mathbin{;} Q)$ in X' or Y' to $P \vee Q$. Note that the occurrence of $(P \mathbin{;} Q)$ could come from X' or from Y': see [**?**] for a full account.

1.2 Isabelle and its HOL theory

Isabelle is an automated proof assistant [**?**]. Its meta-logic is an intuitionistic typed higher-order logic, sufficient to support the built-in inference steps of higher-order unification and term rewriting. Isabelle accepts inference rules of the form "from $\alpha_1, \alpha_2, \ldots, \alpha_n,$ 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". 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 [**?**], such as Isabelle/HOL.

Isabelle/HOL is an Isabelle theory based on the higher-order logic of Church [**?**] and the HOL system of Gordon [**?**]. Thus it includes 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 contains constructs found in functional programming languages, such as `datatype` and `let`, which greatly facilitates re-implementing a program in Isabelle/HOL, and then reasoning about it. 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.

1.3 Related Work on Mechanised Proof Theory

We are aware of only three other attempts to formalise proof theory, cut elimination or strong normalisation. We must confess that we have not done a proper literature survey so we would be grateful for pointers!

The first attempt is 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 [**?**, **?**, **?**]. 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 **δRA** and reported the results in [**?**]. 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 [**?**] 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.*

The second attempt is the work of Matthews *et al* [**?**, **?**, **?**] which implements Feferman's FS_0 in Prolog and Isabelle, and uses it to formalise and extend various meta-theoretical formal systems. It should be possible to formalise cut-elimination in such an implementation, but it does not appear to have been done.

The third attempt is the work of Mikhajlova and von Wright [**?**]. This is the closest to our work since it provides a formalisation of a seqent calculus for first-order logic in the sytem 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.

2 A Deep Embedding of δRA in Isabelle/HOL

In [**?**], 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, we assume that the reader is familiar with ML and logical frameworks in general.

2.1 Representing Formulae, Structures, Sequents and Rules

Formulae of **δRA** are represented by the datatype declaration shown in Figure 1. The constructors `FV` and `SV` represent formula and structure variables which appear in the statement of a rule or theorem, and which are

```
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 $\boldsymbol{\delta}\mathbf{RA}$.

```
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 $\boldsymbol{\delta}\mathbf{RA}$.

instantiated to actual formulae of $\boldsymbol{\delta}\mathbf{RA}$ when constructing derivations. The constructor `PP` represents a primitive proposition p_i, which is not amenable to substitution.

When we reason about substitution of arbitrary structures or formulae for variables (for example, in a rule), we use these variables. Thus we find these variables in the statements of the rules of the object logic. Accordingly, it is an axiom of our meta-logic that a derivation in the object logic may use substitutions (instantiations) of the object logic rules, obtained by substituting for these variables.[1]

Structures of $\boldsymbol{\delta}\mathbf{RA}$ are represented by the datatype declaration shown in Figure 2. The operator `Structform` causes a formula to be "cast" into a structure. 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

[1] Jeremy, I could not understand this paragraph at all. Could you please rewrite it, perhaps with an example?

similarly.

The notation in parentheses in Figures 1 describe an alternative infix syntax which is closer to the actual syntax of $\boldsymbol{\delta}$**RA**. Some complex manipulation of the syntax, available through Isabelle's "parse translations" and "print translations", 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 is used to specify the alternative infix syntax for structures and sequents, details of which we have omitted.

Sequents of $\boldsymbol{\delta}$**RA** are represented by the Isabelle/HOL datatype below:

```
datatype sequent = Sequent structr structr

datatype rule = Rule (sequent list) sequent
              | Bidi sequent sequent
              | InvBidi sequent 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: the two constants `Bidi` and `InvBidi` allow us to cater for these. Thus `Bidi prem concl` means an invertible, or "bi-directional" rule (such as the display postulates) and `InvBidi prem concl` means the rule `Bidi prem concl` used in the inverted sense to derive the conclusion `prem` from the premise `concl`.

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'')`. Special functions `premsRule` and `conclRule` return the list of premises and the conclusion of a rule respectively.

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 statement, it means that the statement is true when anything, of the appropriate type, is substituted for the scheme variable.

The constant `rls` represents the set of rules of $\boldsymbol{\delta}$**RA**, encoded using the datatypes just described: we omit the details of this code.

2.2 Handling Substitutions Explicitly

We now consider the definitions relating to substitution for structure and formula variables. We first define some type abbreviations, and then give the types of some of the functions.

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

For example, `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`, `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''`, X`)`, then `sFind` returns X, and each occurrence of `SV ''X''` in the given object is replaced by X. The functions given substitute for every formula or structure variable in the derivation tree (defined below), rule, sequent, structure or formula.

2.3 Representing Derivations as Trees

We use the term "derivation" to refer to 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 `pftree`) using the following datatype:

```
datatype pftree = Pr sequent rule (pftree list)
                | Unf sequent
```

In a term `Pr seq rule pts` the subterm `seq` is the sequent at the root (bottom) of the tree, and `rule` is the rule used in the last (bottom) inference. Thus, if the tree represents a real derivation, then sequent `seq` will be an instantiation of the conclusion of `rule`, and the corresponding instantiations of the premises of `rule` will be the roots of the trees in the list `pts`. The trees in `pts` are called the *immediate* subtees of `Pr seq rule pts`.

The leaves of a derivation tree are either axioms with no premises, or "`Unf`inished" 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 *complete*. The derivation tree for a derived rule will have the premises of the rule as its `Unf` leaf sequents.

Display calculi typically use the initial sequent $p \vdash p$, using primitive propositions only. It is then proved that the sequent $A \vdash A$ is derivable 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, as the theorem `idfpp`. However we also need to reason about the derivation trees of derived rules; such trees may contain formula and structure variables as well as primitive propositions, and may use the (derived) rule $A \vdash A$, for arbitrary formula A. Therefore we consider formula variables as formulae in their own right, and treat $A \vdash A$ (where A is a formula variable) as an axiom. Thus the derivation tree `Pr (''A'' |- ''A'') idf []` stands for a complete derivation, which uses the lemma that $A \vdash A$ is derivable for all A, whereas the derivation tree `Unf (''A'' |- ''A'')` stands for an incomplete derivation with unfinished premise $A \vdash A$.

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

$$\dfrac{\dfrac{A \vdash p \quad A \vdash A}{A, A \vdash p \wedge A}(\vdash \wedge)}{A \vdash p \wedge A}(ctr)$$

```
Pr (''A'' |- PP p && ''A'') cA
 [Pr (''A'', ''A'' |- PP p && ''A'') ands
 [Unf (''A'' |- PP p),
  Pr (''A'' |- ''A'') idf []]]
```

2.4 Reasoning About Derivation Trees

In this section we describe various functions which allow us to reason about derivations in $\boldsymbol{\delta}$**RA**.

`allPT f pt` holds if property `f` holds for every sub-tree in the derivation tree `pt`.

`allNextPTs f pt` holds if property `f` holds for every proper sub-tree of `pt`.

`wfb (Pr concl rule pts)` holds if sequent rule `rule` has an instantiation with conclusion instance `concl` and premise instances which are the conclusions of the derivation trees in the list `pts`. Such a node is said to be *well-formed*.

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

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

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

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

`conclPT pt` returns the end-sequent of the derivation tree `pt`. 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 complete, that is, it has no unfinished leaves.

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

Definition 2.1 *A derivation tree* `pt` *is* valid *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 ?pt ==
   allPT wfb ?pt & allPT (frb ?rules) ?pt & premsPT ?pt = []"
```

We have explicitly added question marks in front of `rules` and `pt` to flag that they are free variables, even though the question mark would be absent in the Isabelle/HOL theory file itself: we follow this practice throughout this paper from now on.

Definition 2.2 (`IsDerivableR`) *`IsDerivableR rules prems' concl` holds iff there exists a derivation tree `pftr` 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 pftr.
      allPT (frb ?rules) pftr & allPT wfb pftr &
      conclPT pftr = ?concl & set (premsPT pftr) <= ?prems')"
```

Here, `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, for example, if `rule` is of the form `Rule prems concl`, then `IsDerivable rules rule` is equivalent to `IsDerivableR rules (set prems) concl`.

If `rule` is a bi-directional rule, say `Bidi prem concl`, then `IsDerivable rules rule` is equivalent to `IsDerivableB rules prem concl` defined below:

```
"IsDerivableB ?rules ?prem ?concl ==
     IsDerivableR ?rules {?prem} ?concl &
     IsDerivableR ?rules {?concl} ?prem"
```

2.5 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 2.1 (`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
```

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

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

Theorem 2.2 (`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 [**?**], these two properties were, in effect, stated rather than proved. The disadvantage of proceeding that way is the possiblity of stating them incorrectly. For example, [**?**] 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 [**?**, 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 $P \to Q$ could be used as a derived rule on the grounds that one instance of it, say $True \to True$, is provable.

Finally we have a recursive characterization of derivability.

Theorem 2.3 (`IsDerivableR_recur`) *A conclusion `concl` is derivable from premises `prems` using rules `rules` if and only if either `concl` is one of `prems`,*

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

Figure 3: Principal cut on formula $A \vee B$

or there exists an instantiated `rule` *obtained from* `rules` *and 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. ruleFromSet rule ?rules &
 conclRule rule = ?concl & (ALL p : set (premsRule rule).
 IsDerivableR ?rules ?prems p)) )" : thm
```

Here, `ruleFromSet rule rules` means that `rule` is an instantiation of a rule in `rules` (which may be the inverse sense of a bi-directional rule), and `premsRule rule` is a list containing the premisses of rule `rule`.

3 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 là Belnap [**?**]. We assume familiarity with notions like "parametric ancestors" of a cut formula from [**?**].

3.1 Principal Cuts and Belnap's Condition C8

Definition 3.1 *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.*

Belnap's condition C8 guarantees that all principal cuts can be replaced by cuts on strictly smaller formulae. For example, the principal cut on $A \vee B$ shown in Figure 3 can be replaced by the derivation shown in Figure 4.

The replacement derivation contains cuts only on A and B, which are smaller formulae than $A \vee B$. Where the cut-formula is a single formula variable, it is necessarily introduced by the identity axiom: such cuts can be removed trivially.

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

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

Figure 4: Transformed principal cut on formula $A \vee B$

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

Figure 5: Making a cut left-principal

3.2 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 5. 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 5 shows this in the case where A is introduced at just one point.

In Figure 5, the notation $\Pi_L[A]$ and $Z[A]$ 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 $Z[Y]$ means that all such "appropriate" instances of A are changed to Y: that is, instances of A which can be traced to the instance displayed on the right in $X \vdash A$. The rules contained in the new sub-derivation $\Pi_L[Y]$ are the same as the rules used in Π_L; thus it remains to be proved that $\Pi_L[Y]$ is well-formed. The resulting cut in the diagram on the right of Figure 5 is left-principal. Notice that the original sub-derivation Π may be transformed

```
cutOnFmls     :: "formula set => pftree => bool"
cutIsLP       :: "formula => pftree => bool"
cutIsLRP      :: "formula => pftree => bool"
```

Figure 6: Functions for reasoning about cuts

into a *different* sub-derivation Π' during this process since the parametric ancestors of A occurring in $\Pi[A]$ will in turn need to be "cut away" below where they are introduced, and replaced by Y.

Subsequently, the "mirror-image" procedure is followed, to convert a left-principal cut into one or more (left- and right-)principal cuts.

The process of making a cut left-principal, or of making a left-principal cut (left and right) principal is called a "parametric move".

3.3 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 6. Each require the bottom node of the derivation tree to be of the form `Pr seq rule pts`, 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.

Note that it follows from the definitions that a derivation tree satisfying any of `allPT (cutOnFmls s)`, `allPT (cutIsLP A)` and `allPT (cutIsLRP A)` has no unfinished leaves. We omit details.

4 Wansing's Strong Normalization Proof

The ideas inherent in principal and parametric moves are well-known, but they were first applied to prove a strong normalisation result for a display calculus by Wansing in [**?**] (reproduced in [**?**]). In Wansing's book [**?**], he gives a proof of a strong normalization result, that any (sufficiently long) sequence of steps in the process of cut-elimination terminates.

We have implemented large parts of Wansing's strong normalization 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 5 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. Thus our assertion failed, and so we could not prove Wansing's result. We attempted to repair this defect by reference to the relevant part of Wansing's proof [**?**, page 54]. Unfortunately it is based on a diagram much like Figure 5 and appears to rely on the fact that the subtree Π in Figure 5 appears unchanged in the transformed tree. Wansing states that the extension to situations more complex than that of Figure 5 is to be shown "analogously", and he makes no distinction between the situations of Figures **??** and **??**.

5 Discussion

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. For example, Gentzen [**?**] himself first replaced the cut rule with the mix rule, showed that cut was derivable from the mix rule, and then eliminated the mix rule. Recent work of Borisavljevic *et al* [**?**] has shown that the direct elimination of the cut rule is by no means trivial, requiring a further detour through a special normal form for proofs. Thus it is natural to ask if the direct elimination of cut in display logic suffers from similar problems.

The problems caused by contraction can also be avoided by proving a lemma that contraction itself is an admissible rule, as done by Pfenning [**?**]. Then, all sequents obtained by an application of contraction can be replaced by their "uncontracted" counterparts, by appeal to this lemma. We cannot follow this strategy since explicit structural rules like contraction are what give display calculi their generality.

6 A Proof of Strong Normalization

We now provide a proof of strong normalization which overcomes the problems described in Section 4.

We use the same definition of reduction as does Wansing [**?**, §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 [**?**, §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*.

6.1 Defining Strongly Normalisable Derivations

Definition 6.1 (reduces) *Assuming a relation* ***cutReduces*** *(to be defined later) a derivation tree* Π_0 ***reduces*** *to a derivation tree* Π_1 *if either*

(a) Π_0 ***cutReduces*** *to* Π_1*, or*

(b) Π_0 *and* Π_1 *are the same, except that exactly one of the immediate subtrees of* Π_0 *reduces to the corresponding immediate subtree of* Π_1*.*

```
reduces_Unf = "reduces (Unf ?seq) ?ptn = False"
reduces_Pr  = "reduces (Pr ?seq ?rule ?ptl) ?ptn =
 ( (EX ptl'. onereduces ?ptl ptl' & ?ptn = Pr ?seq ?rule ptl')
   | cutReduces (Pr ?seq ?rule ?ptl) ?ptn )"

onereduces_Nil  = "onereduces [] ?ptl' = False"
onereduces_Cons = "onereduces (?h # ?t) ?ptl' = ( ?ptl' ~= []
  & (reduces ?h (hd ?ptl') & ?t = (tl ?ptl')
     | ?h = (hd ?ptl') & onereduces ?t (tl ?ptl')) )"
```

The actual definition of the relation `cutReduces` is given later, meaning that all proofs until then are modulo this definition.

Wansing [**?**, p. 52] defines a *strongly normalizable* derivation tree as a tree from which every sequence of reductions is finite. We define inductively the set of strongly normalizable derivation trees using Isabelle's inductive definition feature, which defines the minimal set closed under the given rules. For example, the rules $\{0 \in \mathcal{S}, n \in \mathcal{S} \Longrightarrow n+2 \in \mathcal{S}\}$ define $\mathcal{S}$ to be the set of even naturals, although the set of all naturals also satisfies the rules. For more details, see [**?**, § 2.10].

Definition 6.2 (sn_set) *The set* ***sn_set*** *is the smallest set of derivation trees such that:*

(a) if Π_0 *cannot be reduced then* $\Pi_0 \in$ ***sn_set***

(b) if every tree Π_1 *to which* Π_0 *reduces is in* ***sn_set*** *then* $\Pi_0 \in$ ***sn_set****.*

```
inductive "sn_set"
  intrs
    snI "(ALL ptn.
    reduces ?pt ptn --> ptn : sn_set) ==> ?pt : sn_set"
```

Note that Definition 6.2.(b) subsumes Definition 6.2.(a), which is why there is only one clause in the corresponding Isabelle code.

Definition 6.3 (strongly normalisable) *A derivation tree is* strongly normalizable *iff it is a member of* ***sn_set****.*

6.2 Various Well-Founded Orderings

To prove that every derivation tree is strongly normalizable, we use a binary relation `dtorder` on derivation trees, and show that it is well-founded. First we need an auxiliary definition.

Definition 6.4 (`sn1red`) *For two lists `ptl1` and `ptl2` of derivation trees, `sn1red ptl1 ptl2` holds iff the lists `ptl1` and `ptl2` are non-empty and differ at only one position where `ptl1` contains tree `pt1` and `ptl2` contains tree `pt2`, and `pt1` reduces to `pt2`, and `pt1` is strongly normalizable.*

```
sn1red_Nil  = "sn1red [] ?ptl2 = False"
sn1red_Cons = "sn1red (?h # ?t) ?ptl2 = (?ptl2 ~= [] &
       (reduces ?h (hd ?ptl2) & ?h : sn_set & ?t = (tl ?ptl2)
       | ?h = (hd ?ptl2) & sn1red ?t (tl ?ptl2)) )"
```

Definition 6.5 (`LRPorder`, `cutorder`, `sn1order`, `dtorder`) *We define four binary relations, `LRPorder`, `cutorder`, `sn1order` and `dtorder` on derivation trees as sets of ordered pairs:*

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

  (i) the cut in Π_1 *is left-principal or right-principal, and the cut in* Π_0 *is neither, or*

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

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

  (i) the size of the cut-formula of Π_1 *is smaller than that of* Π_0*, or*

  (ii) each derivation has the same cut-formula, and $\Pi_1 <_{LRP} \Pi_0$*.*

(c) $\Pi_1 <_{sn1} \Pi_0$ *if* Π_0 *and* Π_1 *are the same except that one of the immediate subtrees of* Π_0 *is strongly normalizable and reduces to the corresponding immediate subtree of* Π_1*.*

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

  (i) the bottom inference of Π_1 *is not (*cut*), but that of* Π_0 *is*

  (ii) $\Pi_1 <_{cut} \Pi_0$

  (iii) $\Pi_1 <_{sn1} \Pi_0$*.*

These definitions, except `cutorder`, use Isabelle's inductive definition feature.

```
inductive "LRPorder"
 intrs
  LRPoL "[| ~ cutLPcf (Pr ?seq1 cutr ?ptl1) ;
            ~ cutRPcf (Pr ?seq1 cutr ?ptl1) ;
              cutLPcf (Pr ?seq2 cutr ?ptl2) |] ==>
         (Pr ?seq2 cutr ?ptl2, Pr ?seq1 cutr ?ptl1) : LRPorder"
  LRPoR "[| ~ cutLPcf (Pr ?seq1 cutr ?ptl1) ;
            ~ cutRPcf (Pr ?seq1 cutr ?ptl1) ;
              cutRPcf (Pr ?seq2 cutr ?ptl2) |] ==>
         (Pr ?seq2 cutr ?ptl2, Pr ?seq1 cutr ?ptl1) : LRPorder"
  LRPoLR "[| ~ cutLRPcf (Pr ?seq1 cutr ?ptl1) ;
               cutLRPcf (Pr ?seq2 cutr ?ptl2) |] ==>
          (Pr ?seq2 cutr ?ptl2, Pr ?seq1 cutr ?ptl1) : LRPorder"

cutorder_def =
 "cutorder == inv_image (measure size <*lex*> LRPorder)
 (%pt. (cutForm pt, pt)) Int {(pt1, pt2). isCut pt1 & isCut pt2}"

inductive "sn1order"
 intrs
  sn1I "sn1red ?ptl2 ?ptl1 ==>
       (Pr ?seq ?rule ?ptl1, Pr ?seq ?rule ?ptl2) : sn1order"

inductive "dtorder"
 intrs
  cnc "?rule ~= cutr ==>
       (Pr ?seq1 ?rule ?ptl1, Pr ?seq2 cutr ?ptl2) : dtorder"
  dtc "?pts : cutorder ==> ?pts : dtorder"
  snr "?pts : sn1order ==> ?pts : dtorder"
```

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

$(t_1, t_2) \in \texttt{measure size}$ iff $\texttt{size}\ t_1 < \texttt{size}\ t_2$

$(t_1, t_2) \in R_1\texttt{<*lex*>}R_2$ iff $(t_1, t_2) \in R_1$ or ($t_1 = t_2$ and $(t_1, t_2) \in R_2$)

$(t_1, t_2) \in \texttt{inv\_image}\ R\ f$ iff $(f\ t_1, f\ t_2) \in R$.

We can then prove the following theorem with relative ease.

Theorem 6.1 (`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
```

Note that, 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 6.1 (`sntr'`) *If* $\Pi_2 <_{cut} \Pi_1$ *and* $\Pi_1 <_{sn1} \Pi_0$ *then* $\Pi_2 <_{cut} \Pi_0$*.*

```
sntr' = "[| (?pty, ?ptza) : cutorder; (?ptza, ?ptz) : sn1order |]
        ==> (?pty, ?ptz) : cutorder" : thm
```

Proof: If $\Pi_2 <_{\mathtt{cut}} \Pi_1$ by virtue of the sizes of their cut-formulae, then $\Pi_2 <_{\mathtt{cut}} \Pi_0$ for the same reason.

Suppose, on the other hand, that $\Pi_2 <_{\mathtt{LRP}} \Pi_1$. Since $\Pi_1 <_{\mathtt{sn1}} \Pi_0$, 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 $\Pi_2 <_{\mathtt{LRP}} \Pi_0$.

(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 $\Pi_2 <_{\mathtt{LRP}} \Pi_1$, it follows that $\Pi_2 <_{\mathtt{LRP}} \Pi_0$.

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

Theorem 6.2 (`wf_dtorder`) *`dtorder` is well-founded.*

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

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

As `sn1order` is well-founded, there is no infinite `sn1order`-decreasing chain; therefore the chain contains infinitely many pairs $\Pi_{i+1} <_{\mathtt{cut}} \Pi_i$. But intermediate pairs in `sn1order` may be "absorbed" – for example if also $\Pi_i <_{\mathtt{sn1}} \Pi_{i-1}$, then by Lemma 6.1, we have $\Pi_{i+1} <_{\mathtt{cut}} \Pi_{i-1}$. Thus there is an infinite `cutorder`-decreasing chain, contradicting that `cutorder` is well-founded. **Q.E.D.**

6.3 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 6.6 (`snHered`) *A derivation tree* Π *satisfies* `snHered` *iff:*

> *if all the immediate subtrees of* Π *are strongly normalizable then* Π *is strongly normalizable.*

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

The next lemma follows from this definition.

Lemma 6.2 (`hereds_sn`) *A derivation tree* Π *is strongly normalizable iff every subtree of* Π *has the property* `snHered`*.*

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

Proof:

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

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

```
sn_set.induct = "[| ?xa : sn_set ; (!!pt. (ALL ptn.
    reduces pt ptn --> ptn : sn_set & ?P ptn) ==> ?P pt ) |]
 ==> ?P ?xa" : thm
```

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

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 `sn_set.induct` says that the immediate subtrees of Π are strongly normalisable. A proof by contradiction shows that the second conjunct of the antecedent of `sn_set.induct` is also true.

Part 2: By induction on the length of the derivation tree Π using the assumption that every subtree of Π, including itself, has the property `snHered`. **Q.E.D.**

6.4 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 6.7 (`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* $\Pi_1^s <_{LRP} \Pi_0$*.*

```
nparRedP_Unf = nparRedP "(Unf ?seq) ?ptn = False"

nparRedP_Pr  = "nparRedP (Pr ?seq ?rule ?ptl) ?ptn = (ALL pts.
 isSubt ?ptn pts & isCut pts -->
    isSubt (Pr ?seq ?rule ?ptl) pts & Pr ?seq ?rule ?ptl ~= pts
  | cutForm (Pr ?seq ?rule ?ptl) = cutForm pts &
    (pts, Pr ?seq ?rule ?ptl) : LRPorder)"
```

In each of Figures **??** and **??**, the cuts in the original and transformed derivations satisfy Definition 6.7(b).

Definition 6.8 (`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 smaller than the cut-formula of the bottom inference of* Π_0*.*

```
c8redP = "c8redP ?pt ?ptn == ALL pts.
  isSubt ?ptn pts & botRule pts = cutr -->
    isSubt ?pt pts & ?pt ~= pts |
    size (cutForm pts) < size (cutForm ?pt)"
```

Note that both Definition 6.8(b) and Definition 6.7(b) imply $(\Pi_1^s, \Pi_0) \in$ `dtorder`.

We now define a *cut-reduction* (being either parametric or primitive[2]) as satisfying one of the properties `nparRedP` and `c8redP`, as well as some further simple conditions which help the proof.

Definition 6.9 (`cutReduces`) *The derivation tree Π_0* cut-reduces *to Π_1 if the following hold simultaneously: Π_0 and Π_1 satisfy either `nparRedP` (for a parametric reduction) or `c8redP` (for a primitive reduction); Π_0 and Π_1 have the same conclusion; the bottom rule of Π_0 is (*cut*); Π_0 and Π_1 are not identical; Π_1 does not consist solely of an unfinished leaf; and Π_0 has at least one immediate subtree.*

```
cutReduces_Pr = "cutReduces (Pr ?seq ?rule ?ptl) ?ptn = (
 (nparRedP (Pr ?seq ?rule ?ptl) ?ptn
   | c8redP (Pr ?seq ?rule ?ptl) ?ptn)
  & conclPT ?ptn = ?seq & ?rule = cutr
  & (Pr ?seq ?rule ?ptl) ~= ?ptn & ~ isUnf ?ptn & ?ptl ~= [] )"
```

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 `ptn` be well-formed (via `allPT wfb ptn`) or require that it use rules which belong to the calculus (via `allPT (frb rls) ptn`). However the definition is also strong in that it requires that either `nparRedP` or `c8redP` holds. Later we will show that the reductions in which we are interested do satisfy `nparRedP` or `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.

6.5 Strong-Normalisation

Lemma 6.3 (`dth`) *For a given derivation Π_0, if all derivation trees $\Pi' <_{dt} \Pi_0$ have the property `snHered`, then so does Π_0.*

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

Proof: Given Π_0, assume that

(a) all derivation trees $\Pi' <_{\mathtt{dt}} \Pi_0$ satisfy `snHered`, and

(b) all immediate subtrees of Π_0 are strongly normalizable.

[2] We have used principal for this term as well, or did I do that ?

We have to prove that Π_0 is strongly normalizable, whence, by Definition 6.6, it satisfies `snHered`. The proof then proceeds by considering possible cases for a reduction of Π_0, giving a tree Π_1.

Q.E.D.

Theorem 6.3 (`all_sn`) *Every derivation tree is strongly normalizable.*

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

Proof: By well-founded induction, it follows from Lemma 6.3 that every derivation tree satisfies `snHered`; the result follows from Lemma 6.2. **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 we are interested in.

6.6 Making A Cut (Left) Principal

We define a relation between two sequents. For boolean `b`, structures `X`, `Y` and sequents `seq1` and `seq2`, the expression `seqRep b X Y seq1 seq2` is true 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 such differences occur only in succedent [antecedent] positions when `b` is `True` [`False`]. For two lists `seql1` and `seql2` of sequents, `seqReps b X Y seql1 seql2` holds if each member of `seql1` is related to the corresponding member of `seql2` by `seqRep b X Y`.

The following are the main theorems used to develop the mechanised proof based on making cuts (left and right) principal. Several of them use the relation `seqRep pn (Structform A) Y`, since `seqRep pn (Structform A) Y seqa seqy` holds when `seqa` and `seqy` are correpsonding sequents in the trees $\Pi_L[A]$ and $\Pi_L[Y]$ from Figure 5.

Theorem 6.4 (`seqExSub1`) *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 `seqRep pn (Structform A) Y seqa seqy` holds, then `pat` can be instantiated to obtain sequent `seqy`.*

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

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 is one of Belnap's conditions [**?**].[3]

We use a functional definition `mLP` of the transformation used to obtain the new derivation tree when making a cut left-principal.

The arguments to `mLP` are

`ptAY`: a derivation tree with root $A \vdash Y$, such as the right subtree Π_R of the left tree given in Figure 5

`seqY`: the sequent X_Y

`Pr seqA rule ptlA`: a tree with root X_A, such as the tree with root $X \vdash A$ which forms the left subtree of the left tree given in Figure 5

where sequent X_A will usually contain occurrences of A in one or more succedent position(s), and sequent X_Y is X_A 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 X_Y whose new cuts (on A) are left-principal, such as the right tree given in Figure 5.

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

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 [**?**, p. 49] this portion of the derivation tree cannot contain another cut: if it did, an infinite sequence of reductions *would* be possible. We therefore defined a function `ncLP seqY ptA` which returns true iff calculating the tree `mLP ptAY seqY ptA` does not involve traversing another cut; `ncRP` has analogous meaning.

We now have the ingredients necessary to prove that we can transform an arbitrary cut on A into one or more left-principal cuts on A as described in Figure 5.

Theorem 6.5 (`makeCutmLP2`) *Suppose we are given a valid derivation tree `ptAY` deriving $A \vdash Y$, and a valid derivation tree `ptA`, and a sequent `seqY` obtained from the conclusion of `ptA` by (possibly) replacing some succedent occurrences of A by Y, and a tree `ptY` obtained by transforming `ptA` using `mLP` without crossing any cuts. Then, for every subtree `pts` of `ptY` which ends in a cut, one of the following holds:*

- *`pts` is a subtree of the original tree `ptA`*

[3] Jeremy, is this really the reason for the condition about no duplications?

- *`pts` is a subtree of the original tree `ptAY`*

- *`pts` ends with a left principal cut on A, and the right subtree of this cut is the original `ptAY`, and the conclusion of `ptY` is the given `seqY`, and `ptY` is well-formed and uses rules from `rls`.*

```
makeCutmLP2 = "[| allPT wfb ?ptAY; allPT (frb rls) ?ptAY;
      conclPT ?ptAY = (?A |- $?Y);
      allPT wfb ?ptA; allPT (frb rls) ?ptA;
      seqRep True (Structform ?A) ?Y (conclPT ?ptA) ?seqY;
      ?ptY = mLP ?ptAY ?seqY ?ptA;
      ncLP ?seqY ?ptA   |]
   ==> (ALL pts.
       isSubt ?ptY pts & isCut pts -->
       isSubt ?ptA pts  |
       isSubt ?ptAY pts |
       cutIsLP ?A pts & hd (tl (nextUp pts)) = ?ptAY) &
       conclPT ?ptY = ?seqY & allPT wfb ?ptY &
       allPT (frb rls) ?ptY" : thm
```

Theorem 6.5 essentially says the following.

Corollary 6.1 *Given a valid derivation tree `ptAY` deriving $A \vdash Y$, and a valid derivation tree `ptA` 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 `ptY`, 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 provided the predicate `ncLP` is satisfied, a parametric reduction which uses `mLP` satisfies the predicate `nparRedP`. We assume an initial cut as in Figure 5: that is, with conclusion $X \vdash Y$ and cut-formula A.

Theorem 6.6 (pRedLP) *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 = "[| ?ptY = mLP ?ptAY ?seqY ?ptA;
      ?pt = Pr ?seqY cutr [?ptA, ?ptAY];
```

```
      ~ rootIsSucP ?ptA; valid rls ?pt;
      ncLP ?seqY ?ptA |]
   ==> nparRedP ?pt ?ptY & valid rls ?ptY &
       conclPT ?ptY = conclPT ?pt" : thm
```

The condition `~ rootIsSucP?ptA` 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 `?pt` and `mLP ?ptAY ?seqY ?ptA` are equal.

The following result is similar, but says that the parametric reduction is a cut-reduction: that is, it satisfies the predicate `cutReduces`. The results requires stronger conditions, but these are satisfied in the circumstances when we perform this reduction.

Theorem 6.7 (`pRedLP3`) *If we are given a valid tree* `pt` *whose bottom-most rule is a cut with conclusion* `seqY`*, and the cut is not left-principal, and we obtain a new tree* `ptY` *by calculating* `mLP` *of the left subtree* `ptA` *of* `pt` *without crossing any cuts, then the reduction from* `pt` *to* `ptY` *is a cut-reduction, and* `ptY` *is also a valid tree.*

```
pRedLP3 = "[| ?pt = Pr ?seqY cutr [?ptA, ?ptAY];
      valid rls ?pt;  ~ rootIsSucP ?ptA;
      ?ptY = mLP ?ptAY ?seqY ?ptA; ncLP ?seqY ?ptA
      |]
   ==> cutReduces ?pt ?ptY & valid rls ?ptY" : thm
```

Once again, the requirement `~ rootIsSucP?ptA` 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.

6.7 Dealing With Principal Cuts

For each logical connnective and constant in the calculus, we prove that a derivation ending in a (left and right) principal cut, where the main connective of the cut formula is that connective, can be transformed into another derivation of the same end-sequent, using only cuts (if any) on formulae which are strict subformulae of the original cut-formula: see Figure 3 and Figure 3 for the case of $\vee$.

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 using the theorem prover.

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

Theorem 6.8 (`orC8`) *Assume we are given a valid derivation tree* `pt`*. Assume that if the bottom rule of* `pt` *is (*cut*), then the cut is principal and its cut-formula is $A \vee B$. Also assume that any cuts above the root of* `pt` *have either A or B as cut-formula. Then then there is a valid derivation tree* `ptn` *with the same conclusion as* `pt`*, such that* all *cuts in* `ptn` *have either A or B as cut-formula.*

```
orC8 = "[| allPT wfb ?pt; allPT (frb rls) ?pt;
  cutIsLRP (?A v ?B) ?pt; allNextPTs (cutOnFmls {?B, ?A}) ?pt |]
 ==> EX ptn. conclPT ptn = conclPT ?pt & allPT wfb ptn &
    allPT (frb rls) ptn & allPT (cutOnFmls {?B, ?A}) ptn" : thm
```

We have also shown that the primitive reductions described above satisfy `c8redP`, and are in fact cut-reductions: for each logical connnective. These results (one for each logical connnective) were combined to give Theorem 6.9.

Theorem 6.9 (`vformC8W'`) *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 Π_0* `cutReduces` *to Π_1.*

```
vformC8W' = "[| ?pt = Pr ?seq cutr ?ptl;  valid rls ?pt ;
       cutIsLRP ?form ?pt |]
   ==> EX ptn. cutReduces ?pt ptn & valid rls ptn" : thm
```

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

6.8 Cut-Elimination Via Strong Normalisation

We now proceed to complete a proof of cut-elimination using the concept 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 6.10 (`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 == {(ptn, pt).
    conclPT ptn = conclPT pt & valid rls ptn & reduces pt ptn}"
```

Lemma 6.4 *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 of reduction. Since a valid 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 its end-sequent. **Q.E.D.**

Theorem 6.10 *Parametric and principal moves give valid reductions.*

Proof: By Theorem 6.7 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. 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 6.4. The argument for principal moves is similar, except that it uses Theorem 6.9 instead of Theorem 6.7. **Q.E.D.**

Theorem 6.11 (`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 = "[| isCut ?pt; valid rls ?pt;
       allNextPTs (cutOnFmls {}) ?pt |]
   ==> EX ptn. cutReduces ?pt ptn & valid rls ptn" : thm
```

Proof: The proof uses only `pRedLP3` and `pRedRP3` and `vformC8W'`, guaranteeing that the reductions are restricted to parametric and principal moves only. Note that `conclPT ptn = conclPT pt` is implied by `cutReduces pt ptn`. **Q.E.D.**

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

Theorem 6.12 (`ExRed`) *For any valid tree which contains a cut, there is available at least one valid reduction.*

```
ExRed = "[| valid rls ?pt ; ~ allPT (Not o isCut) ?pt |]
   ==> EX ptn. (ptn, ?pt) : validRed" : thm
```

Proof: The proof uses Theorem cutExRed. **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.

Theorem 6.13 (`validRed_min`) *For any derivation tree* Π *there exists a tree* Π^r *obtained from* Π *such that* Π^r *cannot be further validly reduced.*

```
validRed_min = "EX ptn. (ptn, ?pt1) : validRed^* &
                 ~ (EX pts. (pts, ptn) : validRed)" : thm
```

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

Theorem 6.14 (`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 = "valid rls ?pt ==> EX ptn.
   allPT (Not o isCut) ptn & valid rls ptn &
   (ptn, ?pt) : validRed^* & conclPT ptn = conclPT ?pt" : thm
```

Corollary 6.2 *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 6.14, 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 6.10 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 6.3, 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 6.15 (`cutElim_SN`) *If a sequent can be derived using rules* `rls`*, then it can be derived from those rules omitting (*cut*).*

```
cutElim_SN = "IsDerivableR rls {} ?concl ==>
             IsDerivableR (rls - {cutr}) {} ?concl" : thm
```

7 Conclusion and Further Work

We have formulated a "deep" embedding of the Display Calculus for Relation Algebra $\boldsymbol{\delta}$**RA** in Isabelle/HOL and used it for reasoning about derivability in $\boldsymbol{\delta}$**RA** rather just mimicking derivations in $\boldsymbol{\delta}$**RA**, as in a previous "shallow" embedding [**?**]. In Section 2.5 we have proved "transitivity" results like `IsDerivableR_trans` and `IsDerivableR_deriv` about the composition of $\boldsymbol{\delta}$**RA** derivations. Similar results were simply asserted (incorrectly) in another reported mechanized formalization of provability for first-order logic, due to their difficulty [**?**].

We have tried to formalise Wansing's proof of strong normalisation for display calculi, but found that it contained an anomaly which we could not resolve. We have therefore proved strong normalisation for the display calculus $\boldsymbol{\delta}$**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 [**?**]. In a sense, this would require a "deeper" embedding still. For, in our first implementation [**?**], we set up the specific connectives and rules of $\boldsymbol{\delta}$**RA** in Isabelle, and used Isabelle proofs as the $\boldsymbol{\delta}$**RA**-derivations. In the present implementation, we again set up the specific connectives and rules of $\boldsymbol{\delta}$**RA**, although we set up data structures to model arbitrary derivations. To prove the generic Belnap theorem, we need to set up the necessary structures to model arbitrary sets of connectives and rules.[4]

[4] Jeremy, why are there two references for "Isabelle's Logics"?