



THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-01-02

**Mechanising Cut-Elimination for
Display Logic**

Jeremy E. Dawson and Rajeev Gore

June 2001

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

<http://cs.anu.edu.au/techreports/>

Recent reports in this series:

- TR-CS-01-01 Stephen Roberts Peter Christen, Markus Hegland and Irfan Altas. *A scalable parallel fem surface fitting algorithm for data mining*. October 2001.
- TR-CS-00-02 Peter Strazdins. *A survey of simulation tools for cap project phase iii*. October 2000.
- TR-CS-00-03 Bill Clarke. *Sparc v9 instruction set specification*. October 2000.
- TR-CS-00-01 Jens Gustedt, Ole A. Maehle, and Jan Arne Telle. *Java programs do not have bounded treewidth*. February 2000.
- TR-CS-99-02 Samuel Taylor. *A distributed visualisation tool for digital terrain models*. July 1999.
- TR-CS-99-01 Peter E. Strazdins. *A dense complex symmetric indefinite solver for the Fujitsu AP3000*. May 1999.

Mechanising Cut-Elimination for Display Logic

Jeremy E. Dawson * and Rajeev Goré **

Department of Computer Science and Automated Reasoning Project,
Australian National University, Canberra ACT 0200, Australia
jeremy@arp.anu.edu.au, rpg@arp.anu.edu.au

Abstract. We describe a deep embedding of the display calculus $\delta\mathbf{RA}$, for relation algebras, using Isabelle/HOL. We then describe how the embedding was used to formalise a cut-elimination theorem for $\delta\mathbf{RA}$. Our implementation generalises easily to handle other display calculi. We consider a published proof of strong normalization of cut-elimination for Display Logic, and discuss the problems with it which prevented us from being able to implement it. We then give a different proof and describe its implementation in Isabelle/HOL.

Keywords: proof systems for relation algebra, non-classical logics, automated deduction, display logic, cut elimination, machine-checked proof theory, strong normalization

1 Isabelle Proof of the Cut-Elimination Theorem

1.1 Representation of proofs, etc

Formulae, structures, sequents, rules Formulae, structures, sequents and rules are modelled by the following datatype declarations.

```
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
```

* Supported by an Australian Research Council Large Grant

** Supported by an Australian Research Council QEII Fellowship

```

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

```

The constructors `FV` and `SV` represent formula and structure variables which are amenable to substitution in the statement of a rule or theorem. The constructor `PP` refers to a primitive proposition, 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. The operator `Structform` causes a formula to be “cast” into a structure. We describe a structure expression as being *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.

```

datatype sequent = Sequent structr structr

```

```

datatype rule = Rule (sequent list) sequent
              | Bidi sequent sequent
              | InvBidi sequent sequent

```

In contrast to the 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 statement is true when anything (of the appropriate type) is substituted for it.

The standard formulation of formulae in a display logic involves formula variables (A, B, \dots) and primitive propositions (p, q, \dots), with the identity axiom $p \vdash p$, using primitive propositions only. It is then proved that the sequent $A \vdash A$ is derivable for all formulae 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.

The notations in parentheses describe the syntax used. Some complex manipulation of the syntax, available through Isabelle’s “parse translations” and “print translations”, allows structure variables and constants to be prefixed by

‘\$’, and the notations `FV`, `SV` and `Structform` to be omitted. For technical reasons related to this the syntax for structures and sequents is given separately; note that 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'')`.

`Rule prems concl` means a rule with premises `prems` and conclusion `concl`. `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. There are also functions `premsRule` and `conclRule` which return the list of premises, or the conclusion, of a rule.

Substitutions: 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"
```

consts

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

subPT      :: "fsSubst => pftree => pftree"
ruleSubst  :: "fsSubst => rule => rule"
seqSubst   :: "fsSubst => sequent => sequent"
strSubst   :: "fsSubst => structr => structr"
fmlSubst   :: "fSubst => formula => formula"
```

This means, for example, that `sFind` is a function of two curried arguments, of types `sSubst` and `string`, and result type `structr`, and that `sSubst` is the type of lists of `(string, structr)` pairs. In fact, 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 (see below), rule, sequent, structure or formula.

Derivation trees: We use the term “derivation” to refer to a proof *within* the sequent calculus, reserving the term “proof” for a metatheoretic 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) step. Thus (if the tree represents a real derivation) `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` will be called the *immediate* subtrees of `Pr seq rule pts`.

The leaves of a derivation tree may be axioms or sequents which are left unproved. The derivation tree for a theorem will have no unproved leaves; we will call such a derivation tree *complete*. The derivation tree for a derived rule will have, as its unproved leaf sequents, the premises of the rule. An unproved leaf sequent is modelled by the tree `Unf seq`.

For example, the derivation tree

$$\frac{\frac{A \vdash p \quad A \vdash A}{A, A \vdash p \wedge A} (\vdash \wedge)}{A \vdash p \wedge A} (\text{contraction})$$

is represented as the term

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

(Here `cA` and `ands` are the contraction and $(\vdash \wedge)$ rules, and `idf` is the rule $A \vdash A$).

Properties of derivation trees:

```
allPT      :: "(pftree => bool) => pftree => bool"
allNextPTs :: "(pftree => bool) => pftree => bool"
wfb        :: "pftree => bool"
frb        :: "rule set => pftree => bool"
premsPT    :: "pftree => sequent list"
IsDerivable :: "rule set => rule => bool"
IsDerivableR :: "rule set => sequent set => sequent => bool"
IsDerivableB :: "rule set => sequent => sequent => bool"
cutOnFmls  :: "formula set => pftree => bool"
cutIsLP    :: "formula => pftree => bool"
cutIsLRP   :: "formula => pftree => bool"
```

We call a node of a derivation tree `Pr seq rule pts` *well-formed* if the subterm `rule` can be instantiated so that the instantiated conclusion is `seq` and the instantiated premises are the conclusions of the trees in the list `pts`. This property is expressed as `wfb (Pr seq rule pts)`. We call a derivation tree `pt` *well-formed* if every node in it is well-formed. This is expressed as `allPT wfb pt`,

as `allPT f pt` means that property `f` holds for every sub-tree in the derivation tree `pt`. Also, `allNextPTs f pt` means that `f` holds for every proper sub-tree of `pt`.

The property `allPT (frb rules)` holds when every rule used in a derivation tree belongs to the set `rules`. The function `premsPT` returns a list of all “premises” (unproved assumptions) of the derivation tree, that is, the sequents found in nodes of the form `Unf seq`.

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 premises.

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 1. A derivation tree is *valid* if

- it is well-formed,
- it uses rules in the set of rules of the calculus, and
- it has no premises (ie, unproved leaves).

Finally, `IsDerivable rules rule` denotes that `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`, as defined below (where `set prems` is the set of the members of the list `prems`).

Definition 2 (IsDerivableR). `IsDerivableR rules prems' concl` holds iff there exists a well-formed derivation tree `pftr` whose premises are among `prems'`, whose conclusion is `concl`, and which uses only rules contained in `rules`.

```
"IsDerivableR rules prems' concl == (EX pftr.
  allPT (frb rules) pftr & allPT wfb pftr &
  conclPT pftr = concl & set (premsPT pftr) <= prems')"
```

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

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

Theorems about IsDerivable: Among the results we have proved about the provability 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 3 (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"

```

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

Theorem 4 (*IsDerivableR_deriv*). *If $concl$ is derivable from $prems$ using the set $rules'$, and each rule in $rules'$ is derivable using $rules$, then $concl$ is derivable from $prems$ using $rules$.*

```

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

```

We note that in another reported formalization of the notion of derivations in a logical calculus [5], 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, in the paper cited, on p. 302, a relation *IsDerivable* is defined inductively to be a relation which is transitive in both the senses of the results above (see the second and third clauses of the definition). 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 characterization of derivability. 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).

Theorem 5 (*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 is an instantiated rule $rule$ 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)))"

```

1.2 Transforming a derivation tree to make a cut principal

The method for cut-elimination depends on cuts being made *principal*, that is, so that the cut-formula is introduced immediately above the cut, on both sides. We

define a cut to be *left-principal* [*right-principal*] if the cut-formula is introduced immediately above the cut on the left [right] side. A principal cut on $A \vee B$ is shown in Fig. 1:

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

Fig. 1. Principal cut on formula $A \vee B$

With an arbitrary rule in place of $(\vee \vdash)$ [$(\vdash \vee)$] it would be left- [right-] principal.

The predicates `cutOnFmls`, `cutIsLP` and `cutIsLRP` 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 (see §1.2 and Fig.1)

`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 the predicates `allPT (cutOnFmls ?s)`, `allPT (cutIsLP ?A)` and `allPT (cutIsLRP ?A)` has no premises.

In the case of a cut that is not left-principal, say we have a tree like the one on the left in Fig. 2. 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 Fig. 2 shows this in the case where A is introduced at just one point. The notation $\Pi_L[A? = Y]$ means that all “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 $\Pi_L[A? = Y]$ are the same as the rules used in Π_L ; thus it remains to be proved that $\Pi_L[A? = Y]$ is well-formed. The resulting cut in the diagram on the right of Fig. 2 is left-principal.

It follows from Belnap’s conditions that where A is introduced by an introduction rule, it is necessarily displayed in the succedent position, as above the top of Π_L in the left branch of the left hand derivation in Fig. 2. Other conditions of Belnap (eg, a formula is displayed where it is introduced, and each structure variable appears only once in the conclusion of a rule) ensure that a procedure can be formally defined to accord with the informal description above; the procedure removes a cut on A which is not left-principal and creates (none, one or more) cut(s) on A which are left-principal.

$$\begin{array}{c}
\frac{\Pi'}{X' \vdash A} (\text{intro-}A) \\
\frac{\Pi_L}{X \vdash A} (\rho) \\
\hline
X \vdash A
\end{array}
\quad
\frac{\Pi_R}{A \vdash Y} (\text{cut})
\quad
\frac{\frac{\frac{\Pi'}{X' \vdash A} (\text{intro-}A) \quad \frac{\Pi_R}{A \vdash Y} (\text{cut})}{X' \vdash Y} (\pi) \quad \frac{\Pi_L[A ? = Y]}{X \vdash Y} (\rho)}{X \vdash Y} (\rho)$$

Fig. 2. Making a cut left-principal

Rather more trivial is where the occurrence of A which we want to change to Y is introduced by the identity rule $A \vdash A$. Then the derivation tree is transformed as shown in Fig.3. Again, the diagram is specific to the case where A is introduced at just one point.

This construction generalizes easily to the case where A is introduced (in one of the above ways) at more than one point (eg, arising from use of one of the rules where a structure variable appears twice in the premises), or where A is “introduced” by use of the weakening rule. This description of the procedure is very loose and informal – the formality and completeness of detail is reserved for the machine proof!

$$\begin{array}{c}
\frac{A \vdash A}{\Pi_L} (\pi) \\
\frac{\Pi_L}{X \vdash A} (\rho) \\
\hline
X \vdash A
\end{array}
\quad
\frac{\Pi_R}{A \vdash Y} (\text{cut})
\quad
\frac{\frac{\Pi_R}{A \vdash Y} (\pi) \quad \frac{\Pi_L[A ? = Y]}{X \vdash Y} (\rho)}{X \vdash Y} (\rho)$$

Fig. 3. Making a cut left-principal – A introduced by identity axiom

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

The replacement relations `seqRep` and `seqReps`: 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 if 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, according as b is `True` [`False`]. For two lists seq11 and seq12 , `seqReps b X Y seq11 seq12` holds if each member of seq11 is related to the corresponding member of seq12 by `seqRep b X Y`.

The following are the main theorems used to develop the mechanised proof based on applying this procedure. Several of them use the relation `seqRep pn (Structform A) Y`, since `seqRep pn (Structform A) Y seqa seqy` holds

when seq_a and seq_y are corresponding sequents in the trees Π_L and $\Pi_L[A? = Y]$.

Theorem 6 ($seqExSub'$). *If pat is formula-free and does not contain any structure variable more than once, and can be instantiated to get seq_a , and $seqRep\ pn$ (Structform A) Y $seq_a\ seq_y$ holds, then pat can be instantiated to get seq_y .*

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

The reason pat must be formula-free is that, supposing for example that pat were Structform (FV ‘B‘), it could be instantiated to Structform A , but not to an arbitrary structure Y .

The stronger result $seqExSub''$ is similar to $seqExSub'$, except that the antecedent [succedent] of the sequent pat may contain a formula, provided that the whole of the antecedent [succedent] is that formula.

The result $seqExSub''$ is used in proceeding up the derivation tree Π_L , changing A to Y : if pat is the conclusion of a rule, which, instantiated with $(fs, suba)$, is used in Π_L , then that rule, instantiated with $(fs, suby)$, is used in $\Pi_L[A? = Y]$. This is expressed in the theorem $extSub''$, which is one step in the transformation of Π_L to $\Pi_L[A? = Y]$.

To explain the theorem $extSub''$ we first define a property $bprops$ of rules: $bprops\ rule$ holds if the rule $rule$ satisfies three properties, which are related (but do not exactly correspond) to Belnap’s conditions – this is discussed further in §1.6. The three properties are

- 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 (ie the whole of one side)

Theorem 7 ($extSub''$). *Given rule $rule$ and an instantiation $ruleA$ of it, and given a sequent $conclY$, such that*

- (i) $seqRep\ pn$ (Structform A) Y ($conclRule\ ruleA$) $conclY$ 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 $seqRep\ pn$ (Structform A) Y .

```

extSub'' =
  "[| conclRule rule = Sequent pant psuc ;
    conclRule ruleA = Sequent aant asuc ;
    conclY = Sequent yant ysuc ;
    (strIsFml pant & aant = Structform A --> aant = yant) ;
    (strIsFml psuc & asuc = Structform A --> asuc = ysuc) ;
    ruleMatches ruleA rule ; bprops rule ;
    seqRep pn (Structform A) Y (conclRule ruleA) conclY |]
  ==> (EX subY. conclRule (ruleSubst subY rule) = conclY &
    seqReps pn (Structform A) Y (premsRule ruleA)
    (premsRule (ruleSubst subY rule)))"

```

This theorem is used to show that, when a node of the tree Π_L is transformed to the corresponding node of $\Pi_L[A? = Y]$, then the next node(s) above can be so transformed. But this does not hold at the node whose conclusion is $X' \vdash A$ (see Fig.2); condition (iii) above reflects this limitation.

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 `extSub''` to transform Π_L to $\Pi_L[A? = Y]$ up to each point where A is introduced, and then inserting an instance of the (cut) rule (or, in the case where A is introduced by the axiom $A \vdash A$, using the Π_R alone, as in Fig.2). It is to be understood that the derivation trees have no (unfinished) premises.

Theorem 8 (makeCutLP). *Given cut-free derivation tree $ptAY$ deriving $(A \vdash Y)$ and ptA deriving $seqA$, and given $seqY$, where $seqRep \ True \ (Structform \ A) \ Y \ seqA \ seqY$ holds (ie, $seqY$ and $seqA$ are the same except (possibly) that A in a succedent position in $seqA$ is replaced by Y in $seqY$), there is a derivation tree deriving $seqY$ whose cuts are all left-principal on A .*

```

makeCutLP =
  "[| allPT (cutOnFmls {}) ?ptAY; allPT (frb rls) ?ptAY;
    allPT wfb ?ptAY; conclPT ?ptAY = (?A |- $?Y);
    allPT (frb rls) ?ptA; allPT wfb ?ptA;
    allPT (cutOnFmls {}) ?ptA;
    seqRep True (Structform ?A) ?Y (conclPT ?ptA) ?seqY |]
  ==> EX ptY.
    conclPT ptY = ?seqY & allPT (cutIsLP ?A) ptY &
    allPT (frb rls) ptY & allPT wfb ptY"

```

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

These were the most difficult to prove of any of theorems required for this cut-elimination proof. The proofs proceed by structural induction on the initial

derivation tree, where the inductive step involves an application of `extSub''`, except where the formula A is introduced. If A is introduced by an introduction rule, then the inductive step involves inserting an instance of (cut) into the tree, and then applying the inductive hypothesis. If A is introduced by the axiom (*id*), then (and this is the base case of the induction) the tree `ptAY` is substituted.

Definition 9. Two derivation trees are *equivalent* if they have the same conclusion (root) sequent.

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

Theorem 10 (allLP). *Given a valid derivation tree pt containing just one cut, which is on formula A and is at the root of pt , there is an equivalent valid tree, all of whose cuts are left-principal and are on A .*

```
allLP =
  "[| cutOnFmls {?A} ?pt; allPT wfb ?pt; allPT (frb rls) ?pt;
    allNextPTs (cutOnFmls {}) ?pt |]
  ==> EX ptn.
      conclPT ptn = conclPT ?pt & allPT (cutIsLP ?A) ptn &
        allPT (frb rls) ptn & allPT wfb ptn"
```

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

1.3 Definition of the transformation

In §1.2 we described the proofs about transforming a derivation tree to make a cut principal. These proofs were couched in terms of existence – the relevant results `makeCutLP` and `makeCutRP`, state the existence of a new derivation tree whose cuts are left-principal, or principal. Likewise, the result `extSub''` merely states the existence of the substitution needed to construct one node of the new derivation tree.

As part of an attempt to implement some of the proofs of Wansing [8] (discussed later, §1.7), we gave (partially) a functional definition of the transformation used to obtain the new derivation tree.

We define the following transformations:

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

The function `mLP` produces the derivation tree whose existence is asserted in the theorem `makeCutLP`. Thus the arguments to `mLP` are

- (a) a derivation tree with root $A \vdash Y$, such as the right subtree of the first tree given in Fig. 2

- (b) the sequent X_Y ,
- (c) a tree with root X_A , such as the left subtree of the first tree given in Fig. 2,

where X_A will usually contain occurrences of A in one or more succedent position(s), and X_Y is X_A with zero, one or more of these occurrences of A changed to Y . The result is a new derivation tree, with root sequent X_Y whose new cuts (on A) are left-principal, such as the second subtree given in Fig. 2. `mLPs` takes for its third argument a list of derivation trees instead of one, and returns a corresponding list.

Likewise `mRP`, 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 , returns a derivation trees whose root sequent is X_Y , whose new cuts are right-principal.

The definition of `mLP` is presented using a function `mLP'`, which is just `mLP` when applied to a derivation tree in which, if the root sequent is $X \vdash A$, the displayed occurrence of A is not principal. In turn, these functions are defined in terms of a function `newSub`, defined by

```
newSub_Pr "newSub (Pr seqA rule ptlA) seqY =
  stepSub (ruleSubOf (Pr seqA rule ptlA)) (conclRule rule) seqY"
```

Here, `ruleSubOf` gives a substitution which takes `rule` to the instance actually used at the bottom of `Pr seqA rule ptlA`. 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.

```
mLP'_Pr = "mLP' ptAY seqY (Pr seqA rule ptlA) =
  (let sub = newSub (Pr seqA rule ptlA) seqY;
    premsinst = premsRule (ruleSubst sub rule)
    in Pr seqY rule (mLPs ptAY premsinst ptlA))" : thm
```

```
mLP_Pr' = "mLP ptAY seqY (Pr seqA rule ptlA) =
  (if strIsFml (succ (conclRule rule)) & succ seqA ~= succ seqY
   then let seqYA = $(antec seqY) |- $(succ seqA)
        in Pr seqY cut [mLP' ptAY seqYA (Pr seqA rule ptlA), ptAY]
   else mLP' ptAY seqY (Pr seqA rule ptlA))" : thm
```

The definition of `mRP` is similar. Proving the 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.

The result `makeCutmLP`, and a similar result `makeCutmRP`, correspond to `makeCutLP` and `makeCutRP`.

```
makeCutmLP =
  "[| allPT (cutOnFmls { }) ptAY; allPT (frb rls) ptAY;
```


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

Fig. 5. Cut-formula introduced by identity axiom

Here is the resulting theorem for \vee :

Theorem 11 (orC8). *If a valid derivation tree pt satisfies*

- *if the bottom rule of pt is (cut), then it is principal and its cut-formula is $A \vee B$, and*
- *any cuts above the root of pt have either A or B as cut-formula,*

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

1.5 Putting it all together

A monolithic proof of the cut-elimination theorem would be very complex, involving several nested inductions. For the transformations above replace one 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, whereas we need, ultimately, to replace many cuts in a given derivation tree. We manage this complexity by defining two predicates, `canElim` and `canElimAll`, whose meanings (assuming valid trees) are given below.

```
canElim      :: "(pftree => bool) => bool"
canElimAll   :: "(pftree => bool) => bool"
```

`canElim f` means that if a valid derivation tree `pt` has no cuts other than (possible) one at the bottom, and satisfies `f`, then there is an equivalent valid cut-free derivation tree.

`canElimAll f` means that if every subtree of a given valid tree `pt` satisfies `f`, then there is a valid cut-free derivation tree `pt'` equivalent to `pt` (we also include as part of this definition the requirement that if the bottom rule of `pt` is not (cut), then the bottom rule of `pt'` is the same).

We can now restate the theorems `allLP` and `allLRP`, in terms of `canElim` and `canElimAll`.

- Theorem 12** (`allLP'`, `allLRP'`). (a) *If you can eliminate any number of left-principal cuts on A from a valid tree pt , then you can eliminate a single cut on A from the bottom of pt .*
- (b) *If you can eliminate any number of principal cuts on A from a valid tree pt , then you can eliminate a single left-principal cut on A from the bottom of pt .*

Proof. (a) The theorem `allLP` says “you can turn a tree pt which has one cut, which is at the bottom and is on A , into a tree with several cuts, which are all left-principal and are on A ”, which is equivalent to the theorem statement.

(b) This has a corresponding proof. \square

Now if we can eliminate one 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 proof tree above the cut. (There is just a slight complication: we need to show that eliminating a cut must not change a lower cut from being principal to not principal).

This gives the following three theorems:

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

We also have two results which will be combined with `elimFmls`

```
twoElim = "[| canElim (cutOnFmls {?A}); canElim (cutOnFmls {?B}) |]
==> canElim (cutOnFmls {?B, ?A})"
```

(“if you can eliminate a cut (at the bottom) on A and if you can eliminate one on B then you can eliminate one which may be either on B or A ”).

```
trivElim = "canElim (cutOnFmls {})"
```

(“you can eliminate cuts (if any) from a tree which has no cuts”).

We also have the theorems such as `orC8` (see §1.4) showing that a tree with a single (left- and right-) principal cut on a particular formula can be replaced by a tree with cuts on the subformulae of that formula; these theorems are converted to a list of theorems `thC8Es'`, of which one is

```
"canElimAll (cutOnFmls {?B, ?A}) ==> canElim (cutIsLRP (?A v ?B))"
```

Combining this with the previous results `elimLRP`, `allLRP'`, `elimLP` and `allLP'` we get the list of theorems `thC8Es`, including

```
"canElimAll (cutOnFmls {?B, ?A}) ==> canElim (cutOnFmls {?A v ?B})"
```

We then combine these with the result `elimFmls` and (in the case of a binary logical connective) `twoElim` to get the list of theorems `elimFmlRecs`, including

```
"[| canElim (cutOnFmls {?A}); canElim (cutOnFmls {?B}) |]
  ==> canElim (cutOnFmls {?A v ?B})"
```

The theorems `elimFmlRecs` are then used in a proof by structural induction on the structure of a formula to prove

```
canElimFml = "canElim (cutOnFmls {?fml})"
```

(“you can eliminate one cut on any given formula `fml`”),

```
canElimAny = "canElim (cutOnFmls UNIV)",
```

(“you can eliminate one cut whatever the cut-formula”), and, using `elimFmls`,

```
canElimAll = "canElimAll (cutOnFmls UNIV)"
```

Finally we convert `canElimAll` into

```
cutElim =
  "IsDerivableR rls {} ?concl ==> IsDerivableR (rls - {cut}) {} ?concl"
```

1.6 Use of properties of the Display Calculus

Belnap’s cut-elimination theorem 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 the proof for the specific case of $\delta\mathbf{RA}$. We refer to the conditions in the form given by Kracht [4].

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 $\delta\mathbf{RA}$.

`ruleInOrInv rule rules` : either `rule` or its inverse is a member of `rules`

`seqNoSSF seq` : if a formula appears on one side of the \vdash in `seq` then the whole of that side is a formula.

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

`noDups list` : `list` contains no member more than once.

`seqSVs' bool seq` : a list of the structure variables in antecedent or succedent (depending on `bool`) positions in `seq`.

The following three results are used in the proofs of `makeCutLP` and `makeCutRP`.

In the conclusion of a rule, no structure has a sub-structure which is a formula

```
seqNoSSF_rls =
  "ruleInOrInv ?rule rls ==> seqNoSSF (conclRule ?rule)" : thm
```

No structure variable appears more than once in the conclusion of a rule.

```
noDupSVs_rls =
  "ruleInOrInv ?rule rls ==> noDups (seqSVs (conclRule ?rule))" : thm
```

If a structure variable appears in the conclusion of a rule in antecedent [succedent] position, then it does not appear in a premise of that rule in succedent [antecedent] position.

```
noSVsAS_rls =
  "[| ruleInOrInv ?rule rls; ?prem : set (premsRule ?rule);
    ?s : set (seqSVs' ?b (conclRule ?rule)) |]
  ==> ?s ~: set (seqSVs' (~ ?b) ?prem)"
```

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.

```
noNewSVs_rls =
  "[| ruleInOrInv ?rule rls; ?prem : set (premsRule ?rule);
    ?s : set (seqSVs' ?b ?prem) |]
  ==> ?s : set (seqSVs' ?b (conclRule ?rule))"
```

The result `orC8`, and the corresponding results for each logical connective, form Belnap's condition (C8).

Belnap's condition (C3) is equivalent to `noDupSVs_rls`.

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 the proof is `noSVsAS_rls`, which is weaker than (C4), since `noSVsAS_rls` permits a structure variable to appear in both antecedent and succedent positions in the premises, provided that it does not appear in the conclusion.

(C4) is implied by `noDupSVs_rls` and `noNewSVs_rls` (which we had used originally). However `noNewSVs_rls` is stronger than it needs to be for this result, since (C4) permits structure variables to appear in the premise(s) but not in the conclusion. Note that Display Calculi generally do satisfy `noNewSVs_rls`.

Our theorem `seqNoSSF_rls` is equivalent to Belnap's (C5), since, in Kracht's formulation, formula variables can never be parameters.

Under Kracht's formulation, Belnap's (C2), (C6) and (C7) are trivially satisfied.

1.7 Wansing's Strong Normalization Proof

In Wansing's book [8], he gives a proof of a strong normalization 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 Fig. 1 to Fig. 4, see §1.4

parametric moves such as the transformation in Fig. 2

Our approach so far has been to do these transformations only on a topmost cut in a derivation tree; this is implicit in the statement of `makeCutLP` and `makeCutRP`, and in the proofs of the theorems in §1.5. However, Wansing permits these moves to occur for a cut which is *not* a topmost cut. His 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 Fig. 2 shown below:

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

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 Π' in the right tree of Fig. 2 actually is contained in the left tree (as indeed it is in that diagram). That is, the transformation process does not alter Π' but leaves it intact. Some more complicated diagrams are needed. We said in §1.2 “This construction generalizes easily to the case where A is introduced . . . at more than one point . . .”, and we now elaborate on this. Fig. 6(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 Fig. 2.

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

(i) before the transformation

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

(ii) after the transformation

Fig. 6. Branch in left subtree

In this situation our assertion implies, in effect, that the subtrees Π' and Π'' in the transformed derivation tree actually appear in the original one – again, as indeed they do.

Consider now the situation where a second occurrence of A is introduced by contraction. Here we find that A is introduced twice, but “in succession”, not “in parallel”. It seems clearer if we give a more concrete example, as shown in Fig. 7, where A is $\neg B$.

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

(i) before the transformation

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

(ii) after the transformation

Fig. 7. Two successive introductions of $\neg B$

Now, if we try to match the left tree in Fig. 2 with the tree in Fig. 7(i), we get that A of Fig. 2 is $\neg B$ in Fig. 7 (as stated), X' is $(* \neg B, X)$ and Π' is the subtree of Fig. 7(i) whose root sequent is $* \neg B, X \vdash *B$. Now, in this case, this subtree Π' does *not* appear unchanged in the transformed tree Fig. 7(ii). Rather, Fig. 7(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.

We attempted to repair this defect by reference to the relevant part of Wansing’s proof [8, p. 54]. Unfortunately it is based on a diagram much like Fig. 2 and appears to rely on the fact that the subtree II' in Fig. 2 appears unchanged in the transformed tree. He states that the extension to situations more complex than that of Fig. 2 is to be shown “analogously”, and he makes no distinction between the situations of Figs 6 and 7.

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 [7] 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 [2] 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, and to answer why Belnap’s proof does not require such detours.

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 Fig. 2 and Fig. 6, 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 Fig. 7 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. In a Gentzen-style sequent calculus, the procedures used to get around this problem can be quite complex (eg, [2]).

The occurrence of a cut, and above it a contraction involving the cut-formula causes no great problem for the proofs of cut-elimination in Display Logic, such as that of Belnap [1, § 4], and the more detailed proof described in this paper. If we described our proof in terms of the rank of a cut, the comparison between the ranks of two cuts would depend on, in order,

- the size of the cut-formula, and then
- whether the cut is (left- and right-)principal (lowest rank), left-principal, or neither (highest rank)

With this definition of rank, each step of the proof transforms a cut into cut(s) of lower rank (that is, regarding a transformation such as those shown in Figs 6 and 7 as a single step of the proof). However we should note that neither Belnap’s proof or ours specifically uses a notion of rank.

The computer-based proof that these transformations are possible is done by structural induction on the structure of a derivation tree *before* transformation, so a given subtree may be transformed into a subtree of any size. We may describe the procedure informally as follows. Given a tree such as that of Figs 2, 6 or 7, take the left sub-tree and consider all the occurrences of A which are “parametric ancestors” of the occurrence of A on the right in the bottom $X \vdash A$ [1]. Change

all these occurrences of A to Y , except at sequents where A is introduced by a logical introduction rule. These sequents are all of the form $X'[A] \vdash A$, a fact that can be proved from the properties of Display Logic. At these sequents, we would want the occurrence on the right to be changed to Y to fit in with the (changed) derivation tree below, but we would want it to remain as A to fit in with the derivation tree above. So at such sequents we replace the tree on the left of Fig. 8 with that on the right, noting that $\Pi'[A]$ does not contain any occurrence of A traceable to the succedent occurrence of A in $X'[A] \vdash A$.

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

Fig. 8. Changing parametric ancestor occurrences of A to Y

The case of A introduced by the identity axiom is simpler, and is dealt with as in Fig. 3.

It can be seen that when the relevant part of the cut-elimination procedure is stated in this way, we can accommodate the case of Fig. 7 by describing the procedure in terms of $X'[A]$ and $X'[Y]$, and $\Pi'[A]$ and $\Pi'[Y]$ instead of just X' .

2 A Proof of Strong Normalization

We provide a proof of strong normalization which overcomes the problems described in §1.7.

We use the same definition of reduction as does Wansing [8, §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 [8, §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*.

Definition 13 (reduces). Assuming a relation `cutReduces` (to be defined later) a derivation tree Π **reduces** to a derivation tree Π' if either

- (a) Π `cutReduces` to Π' , or
- (b) Π and Π' are the same, except that exactly one of the immediate subtrees of Π reduces to the corresponding immediate subtree of Π'

`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')))"

```

Wansing [8, 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.

Definition 14 (`sn_set`). A derivation tree Π is *strongly normalizable* if either

- (a) Π cannot be reduced to another tree Π' , or
- (b) for every tree Π' to which Π can be reduced, this definition determines that Π' is strongly normalizable.

We let `sn_set` denote the set of strongly normalizable derivation trees.

Definition 15 (`sn_set`). A derivation tree Π is *strongly normalizable* if it is a member of `sn_set`, which we define to be the smallest set of trees such that

- (a) If Π cannot be reduced to another tree Π' , then $\Pi \in \text{sn_set}$.
- (b) If, for every tree Π' to which Π can be reduced, $\Pi' \in \text{sn_set}$, then $\Pi \in \text{sn_set}$.

This definition uses Isabelle's inductive definition feature. This defines the minimal set closed under the given rules. For example, defining a set \mathcal{S} of natural numbers using the rules $\{0 \in \mathcal{S}, n \in \mathcal{S} \implies n + 2 \in \mathcal{S}\}$ defines the set of even naturals (although the set of all naturals also satisfies the rules). For more details, see [6, §2.10]

```

snI "(ALL ptn. reduces pt ptn --> ptn : sn_set) ==> pt : sn_set"

```

To prove that every derivation tree is strongly normalizable, we first define a binary relation `dtorder` on derivation trees, and show that it is well-founded.

First we need an auxiliary definition.

Definition 16 (`sn1red`). For two lists `ptl1` and `ptl2` of derivation trees, `sn1red ptl1 ptl2` means that the lists `ptl1` and `ptl2` are non-empty and are the same except at one position, where `ptl1` contains tree `pt1` and `ptl2` contains tree `pt2`, and

- `pt1` reduces to `pt2`, and
- `pt1` is strongly normalizable.

```

"sn1red [] pt12 = False"
"sn1red (h # t) pt12 = (pt12 ~= [] &
  (reduces h (hd pt12) & h : sn_set & t = tl pt12 |
  h = hd pt12 & sn1red t (tl pt12)))"

```

Definition 17 (LRPorder, cutorder, sn1order, dtorder). We define four binary relations, LRPorder, cutorder, sn1order and dtorder on derivation trees. We define them, as sets of ordered pairs, by:

- (a) $\Pi_1 <_{\text{LRP}} \Pi_0$ 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.
- (b) $\Pi_1 <_{\text{cut}} \Pi_0$ 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 $\Pi_1 <_{\text{LRP}} \Pi_0$.
- (c) $\Pi_1 <_{\text{sn1}} \Pi_0$ 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) $\Pi_1 <_{\text{dt}} \Pi_0$ iff any of the following hold:
 - (i) the bottom inference of Π_1 is not (*cut*), but that of Π_0 is
 - (ii) $\Pi_1 <_{\text{cut}} \Pi_0$
 - (iii) $\Pi_1 <_{\text{sn1}} \Pi_0$.

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

```

inductive "LRPorder"
  intrs
    LRPoL "[| ~ cutLPcf (Pr seq1 cutr pt11) ;
~ cutRPcf (Pr seq1 cutr pt11) ;
  cutLPcf (Pr seq2 cutr pt12) |] ==>
  (Pr seq2 cutr pt12, Pr seq1 cutr pt11) : LRPorder"
    LRPoR "[| ~ cutLPcf (Pr seq1 cutr pt11) ;
~ cutRPcf (Pr seq1 cutr pt11) ;
  cutRPcf (Pr seq2 cutr pt12) |] ==>
  (Pr seq2 cutr pt12, Pr seq1 cutr pt11) : LRPorder"
    LRPoLR "[| ~ cutLRPcf (Pr seq1 cutr pt11) ;
cutLRPcf (Pr seq2 cutr pt12) |]
  ==> (Pr seq2 cutr pt12, Pr seq1 cutr pt11) : LRPorder"

defs
  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` depends on a number of functions defined in Isabelle/HOL to assist in constructing well-founded relations, as follows:

- $(t_1, t_2) \in \text{measure size}$ iff $\text{size } t_1 < \text{size } t_2$
- $(t_1, t_2) \in R_1 \langle * \text{lex} * \rangle R_2$ iff $(t_1, t_2) \in R_1$ or $t_1 = t_2$ and $(t_1, t_2) \in R_2$
- $(t_1, t_2) \in \text{inv_image } R \ f$ iff $(f \ t_1, f \ t_2) \in R$

For further details, see [6, §2.9.2].

We can then prove in turn the following theorems, which are relatively straightforward. Note that, despite the terminology, these relations are not reflexive, and not all transitive. But the intuition is that $(\Pi_1, \Pi_0) \in \text{dtorder}$ means that Π_1 is closer to being cut-free (in some sense) than Π_0 .

Theorem 18 (`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

```

To prove that `dtorder` is well-founded, we first need a result on the interaction between `cutorder` and `sn1order`:

Lemma 19 (`sntr'`). *If $\Pi_2 <_{\text{cut}} \Pi_1$ and $\Pi_1 <_{\text{sn1}} \Pi_0$ then $\Pi_2 <_{\text{cut}} \Pi_0$.*

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

- 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 $\Pi_2 <_{\text{LRP}} \Pi_0$.

- 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, $\Pi_2 <_{\text{cut}} \Pi_0$.

□

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

Theorem 20 (*wf_dtorder*). *dtorder is well-founded.*

Proof. It is easy to see that an infinite *dtorder*-decreasing chain $\Pi_0 >_{\text{dt}} \Pi_1 >_{\text{dt}} \Pi_2 >_{\text{dt}} \dots$ 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 each pair (Π_{i+1}, Π_i) is in either *sn1order* or *cutorder*.

As *sn1order* is well-founded, there is no infinite *sn1order*-decreasing chain; therefore the chain contains infinitely many pairs $(\Pi_{i+1}, \Pi_i) \in \text{cutorder}$. But intermediate pairs in *sn1order* may be “absorbed” – for example if $(\Pi_i, \Pi_{i-1}) \in \text{sn1order}$, by Lemma 19, we have $(\Pi_{i+1}, \Pi_{i-1}) \in \text{cutorder}$. Thus there is an infinite *cutorder*-decreasing chain, contradicting that *cutorder* is well-founded.

□

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

We next define *snHered*, a property of derivation trees, so named for the idea that such a tree inherits strong normalization from its immediate subtrees.

Definition 21 (*snHered*). A derivation tree *pt* satisfies the property *snHered* iff the following holds:

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

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

The next lemma follows from this definition.

Lemma 22 (*hereds_sn*). *Every subtree of a derivation tree pt has the property snHered iff pt is strongly normalizable.*

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

We intend to define cut-reduction in such a way as to enable 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 23 (`nparRedP`). A derivation tree reduction from Π to Π' satisfies `nparRedP` if, for each subtree Π'_s of Π' whose bottom inference is (*cut*), either

- (a) Π'_s is a proper subtree of Π , or
- (b) $(\Pi'_s, \Pi) \in \text{LRPorder}$, and they have the same cut-formula.

It can be seen that, in the case of the reductions shown in Figs. 6 and 7, the cuts visible in those figures satisfy Definition 23(b).

Definition 24 (`c8redP`). Let the bottom inference of Π be (*cut*). A reduction from Π to Π' satisfies `c8redP` if, for each subtree Π'_s of Π' whose bottom inference is (*cut*), either

- (a) Π'_s is a proper subtree of Π , or
- (b) the cut-formula of Π'_s is smaller than that of Π .

Note that both Definition 24(b) and Definition 23(b) imply $(\Pi'_s, \Pi) \in \text{dtorder}$.

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

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

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

Definition 25 (`cutReduces`). The derivation tree Π *cut-reduces* to Π' if the following hold:

- (a) Π and Π' satisfy either `nparRedP` (for a parametric reduction) or `c8redP` (for a primitive reduction)
- (b) Π and Π' have the same conclusion
- (c) the bottom rule of Π is (*cut*)
- (d) $\Pi \neq \Pi'$
- (e) Π' does not consist solely of a non-axiomatic leaf

(f) Π has at least one immediate subtree

```
cutReduces_Pr "cutReduces (Pr seq rule pt1) ptn =
  (rule = cutr & pt1 ~ = [] & ~ isUnf ptn &
   (c8redP (Pr seq rule pt1) ptn | nparRedP (Pr seq rule pt1) ptn) &
   conclPT ptn = seq & (Pr seq rule pt1) ~ = ptn)"
```

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 (`allPT wfb ptn`) or use rules which belong to the calculus (`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. (Clearly strong normalization for a larger class of reductions implies strong normalization for a smaller class).

Lemma 26 (dth). *For a given derivation Π_0 , if all derivation trees $\Pi' <_{\text{at}} \Pi_0$ have the property `snHered`, then so does Π_0 .*

Proof. Given Π_0 , assume that

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

We have to prove that Π_0 is strongly normalizable, whence, by Definition 21, it satisfies `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 normalizable) subtree of Π_0 , and so $\Pi_1 <_{\text{sn1}} \Pi_0$ and hence $\Pi_1 <_{\text{at}} \Pi_0$ by definition. Since all immediate subtrees of Π_1 are strongly normalizable (they are equal to or are a reduction of immediate subtrees of Π_0), and `snHered` holds for Π_1 , the derivation Π_1 is strongly normalizable.

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 normalizable. If it is a reduction of the bottom cut, we get the new tree Π_1 whose cuts are either in copies of (strongly normalizable) proper subtrees of Π_0 or are cuts which are smaller (in `dtorder`) than the bottom cut of Π_0 (see the remark following Definitions 23 and 24). Thus every subtree Π_1^s of Π_1 (including Π_1 itself) satisfies one of the following:

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

Now a subtree Π_1^s satisfying (a) is strongly normalizable, and a subtree Π_1^s satisfying (b) satisfies `snHered`. Thus every Π_1^s satisfies `snHered`. Since Π_1^s is an arbitrary subtree of Π_1 , it follows from Lemma 22 that Π_1 is strongly normalizable.

Thus in either case Π_1 is strongly normalizable. Since Π_1 was obtained via an arbitrary reduction from Π_0 , it follows that Π_0 is strongly normalizable. Thus we have that `snHered` holds for Π_0 . \square

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

Theorem 27 (`all_sn`). *Every derivation tree is strongly normalizable.*

Proof. By well-founded induction, it follows from Lemma 26 that every derivation tree satisfies `snHered`; the result follows from Lemma 22. \square

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

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.

Recall that the parametric reduction involves tracing up the derivation tree, from a premise of the cut being reduced, to all points where the cut-formula A is introduced. As Wansing describes [8, p. 49] this portion of the derivation tree cannot contain another cut (if this were allowed, an infinite sequence of reductions *would* be possible).

Definition 28 (`ncLP`). The exact definition of `ncLP` is given below, but the “intention” of the definition is that `ncLP seqY ptA` holds iff calculating the tree `mLP ptAY seqY ptA` does not involve traversing another cut; `ncRP` has analogous meaning.

```
ncLP  :: "sequent => pftree => bool"
ncLPs :: "sequent list => pftree list => bool"
ncRP  :: "sequent => pftree => bool"
ncRPs :: "sequent list => pftree list => bool"

ncLP_Pr "ncLP seqY (Pr seqA rule pt1A) = (seqA = seqY | rule ~= cutr &
  (if strIsFml (succ (conclRule rule)) & succ seqA ~= succ seqY then
    let seqYA = Sequent (antec seqY) (succ seqA) ;
        sub = newSub (Pr seqA rule pt1A) seqYA ;
        premsinst = premsRule (ruleSubst sub rule)
    in (ncLPs premsinst pt1A)
  else
    let sub = newSub (Pr seqA rule pt1A) seqY ;
        premsinst = premsRule (ruleSubst sub rule)
    in (ncLPs premsinst pt1A)))"
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)"
```

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

This follows from the following lemma.

Lemma 29 (sameConcLP). *If $seqY$ is equal to the conclusion of ptA , then $mLP\ ptAY\ seqY\ ptA = ptA$.*

Where this holds, clearly calculating $mLP\ ptAY\ seqY\ ptA$ does not require proceeding further up the tree.

```
sameConcLP = "allPT wfb ?ptA ==>
  mLP ?ptAY (conclPT ?ptA) ?ptA = ?ptA" : thm
```

In the following we will assume a cut as in Fig. 2, ie, with conclusion $X \vdash Y$ and cut-formula A .

Theorem 30 (pRedLP). *Consider a parametric reduction of a cut which proceeds by transforming the left subtree (to change its conclusion from $X \vdash A$ to $X \vdash Y$), using the function mLP . Assume that the subtree satisfies the condition $ncLP$ (in effect, that the transformation can be performed without traversing another cut), and assume the cut is not already left-principal. Then the reduction satisfies the condition $nparRedP$.*

```
pRedLP = "[| ?pt = Pr ?seqY cutr [?ptA, ?ptAY]; ~ rootIsSucP ?ptA;
  allPT wfb ?pt; allPT (frb rls) ?pt; ncLP ?seqY ?ptA |]
  ==> nparRedP ?pt (mLP ?ptAY ?seqY ?ptA)" : thm
```

The condition $\sim\ rootIsSucP\ ?ptA$ means in effect that the cut is not already left-principal. It is required as a condition of the theorem because if the cut is already 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, ie, it satisfies the predicate `cutReduces`. The results requires stronger conditions, but these are satisfied in the circumstances when we perform this reduction.

Theorem 31 (cutRedLP). *If a cut, at the root of a tree Π (pt), is not left-principal (and also the left branch is not an unproved leaf), and*

- (a) Π is well-formed,
- (b) Π uses rule set rls , and
- (c) obtaining a new tree Π' by calculating mLP of the left branch of Π (to change its conclusion from $X \vdash A$ to $X \vdash Y$) does not traverse another cut

then the reduction from Π to Π' is a cut-reduction.

```
cutRedLP = "[| ?pt = Pr ?seqY cutr [?ptA, ?ptAY]; ~ rootIsSucP ?ptA;
  ~ isUnf ?ptA; allPT wfb ?pt; allPT (frb rls) ?pt;
  ncLP ?seqY ?ptA |]
  ==> cutReduces ?pt (mLP ?ptAY ?seqY ?ptA)" : thm
```

The following results, `allLPm` and `allLRPm`, are similar to the corresponding results `allLP` and `allLRP`, proved earlier, the difference being that these results additionally prove that the new derivation tree either is equal to the old one or is a cut-reduction of it.

Notice that these results operate on a tree with a single cut, at the bottom – that is, their purpose is for dealing with a topmost cut of a tree with multiple cuts. The proofs of these results use Theorem 31 (`cutRedLP`) and `cutRedRP` (which is a “mirror-image” analogue of `cutRedLP`, using `mRP`). They therefore use the parametric reductions defined in terms of `mLP` and `mRP`.

Theorem 32 (`allLPm`). *Given a tree Π with one cut, at the bottom, with cut-formula A , such that*

- (a) Π is well-formed,
- (b) Π uses rule set `rls`, and
- (c) Π has no premises (unproved leaves)

then Π cut-reduces to a tree Π' such that

- Π' has the same conclusion as Π ,
- Π' satisfies (a) to (c) above, and
- all cuts in Π' are left-principal, with cut-formula A

Note that the statement in Isabelle also allows `pt` (Π) to be cut-free, and `ptn` (Π') to be equal to it. Recall that the conditions `allPT cutOnFmls fmls pt`, `allPT cutIsLP A pt`, etc, imply that `pt` has no premises.

Theorem 33 (`allLRPm`). *As for Thm. 32, except that the cut at the root of Π is left-principal, and all cuts in Π' are (left- and right-) principal.*

```
allLPm = "[| cutOnFmls {?A} ?pt; allPT wfb ?pt; allPT (frb rls) ?pt;
  allNextPTs (cutOnFmls {}) ?pt |]
  ==> EX ptn. (ptn = ?pt | cutReduces ?pt ptn) &
  conclPT ptn = conclPT ?pt & allPT (cutIsLP ?A) ptn &
  allPT (frb rls) ptn & allPT wfb ptn" : thm

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

We have also shown that the primitive reductions described in §1.4 satisfy `c8redP`, and are in fact cut-reductions. For each logical connective we considered the reduction analogous to that in Fig. 5 (which was used to prove `orC8` and analogous results), and showed that it satisfies `c8redP`. These results (one for each logical connective) were combined to give Theorem 34.

Theorem 34 (formC8W). *Given a valid tree Π with one cut, which is principal and is at the bottom of Π , there exists a valid tree Π' such that the relation $c8redP$ holds between Π and Π' , and Π' has the same conclusion as Π .*

```
formC8W = "[| cutIsLRP ?form ?pt; allPT wfb ?pt; allPT (frb rls) ?pt |]
  ==> EX ptn. conclPT ptn = conclPT ?pt & allPT wfb ptn &
    allPT (frb rls) ptn & c8redP ?pt ptn &
    set (premsPT ptn) <= set (premsPT ?pt)" : thm
```

We have now shown that our earlier proof of cut-elimination used, or could have used) only cut-reductions. This is enough to demonstrate a cut-elimination procedure using cut-reductions. However 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 we are interested in. Recall that a *valid* derivation tree is one which is well-formed, uses rules from `rls`, and has no premises.

Definition 35 (validRed). A *valid reduction* is a reduction of tree Π to tree Π' , where Π' is a valid tree which has the same conclusion as Π .

```
validRed_def "validRed == {(ptn, pt).
  conclPT ptn = conclPT pt & valid rls ptn & reduces pt ptn}"
```

Theorem 36 (cutEXred). *For any valid tree which has exactly one cut, at the bottom, there is available a cut-reduction to another valid tree with the same conclusion.*

Proof. The proof of theorem uses Theorems 32 and 33 for the cases where parametric reductions are required, and Theorem 34 where primitive reductions are required. \square

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

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

Theorem 37 (ExRed). *For any valid tree which contains a cut, there is available a valid reduction.*

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

The outline of the proof from here is clear: since every tree with a cut admits a reduction, and there is no infinite sequence of reductions, so an effective cut-elimination procedure is to repeatedly perform any reduction until it is no longer possible.

Theorem 38 (`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.*

Note that the notation ^* denotes transitive closure.

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

Theorem 39 (`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 &
    conclPT ptn = conclPT ?pt & valid rls ptn &
    (ptn, ?pt) : validRed^*" : thm
```

That is to say, given a derivation tree, there is a cut-free equivalent obtained by repeated reductions. Therefore we have the cut-elimination result, 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 40 (`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
```

3 Conclusion and Further Work

We have formulated the Display Calculus for Relation Algebra, $\delta\mathbf{RA}$, in Isabelle/HOL. This formulation is a “deep embedding” which allows us to model, and reason about, derivations in the logic (rather than just performing derivations, as in a shallow embedding). We have proved, from the definitions, “transitivity” results about the composition of proofs. These are results which have been omitted – due to their difficulty – from another reported mechanized formalization of provability.

We have proved Belnap’s cut-elimination theorem for Display Calculi for the Display Calculus $\delta\mathbf{RA}$. This has been a considerable effort, and could not have been achieved without the complementary features (found in Isabelle) of the

- extensive provision of powerful tactics, and
- 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 arbitrary sets of structural and logical connectives (as is done by Goré in [3]), and of rules. In a sense, this would require a “deeper” embedding still. For, in our first implementation, 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 proofs. To prove the generic Belnap theorem, we need to set up the necessary structures to model arbitrary sets of connectives and rules.

References

1. N D Belnap. Display logic. *Journal of Philosophical Logic*, 11:375–417, 1982.
2. Mirjana Borisavljevic, Kosta Dosen, and Zoran Petric. On permuting cut with contraction. *Mathematical Structures in Computer Science*, 10:99–136, 2000.
3. R Goré. Substructural logics on display. *Logic Journal of the Interest Group in Pure and Applied Logic*, 6(3):451–504, 1998.
4. M Kracht. Power and weakness of the modal display calculus. In H Wansing, editor, *Proof Theory of Modal Logics*, pages 92–121. Kluwer, 1996.
5. 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.
6. 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.
7. M. E. Szabo, editor. *The collected papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.
8. Heinrich Wansing. *Displaying Modal Logic*, volume 3 of *TRENDS IN LOGIC*. Kluwer Academic Publishers, Dordrecht, August 1998. Hardbound, ISBN 0-7923-5205-X.