

Formalised Cut Admissibility 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@discus.anu.edu.au rpg@discus.anu.edu.au

Abstract. We use a deep embedding of the display calculus for relation algebras $\delta\mathbf{RA}$ in the logical framework Isabelle/HOL to formalise a machine-checked proof of cut-admissibility for $\delta\mathbf{RA}$. Unlike other “implementations”, we explicitly formalise the structural induction in Isabelle/HOL and believe this to be the first full formalisation of cut-admissibility in the presence of explicit structural rules.

1 Introduction

Display Logic [1] is a generalised sequent framework for non-classical logics. Since it is not really a logic, we prefer the term display calculi and use it from now on. Display calculi extend Gentzen’s language of sequents with extra, complex, n -ary structural connectives, in addition to Gentzen’s sole structural connective, the “comma”. Whereas Gentzen’s comma is usually assumed to be associative, commutative and inherently poly-valent, no such implicit assumptions are made about the n -ary structural connectives in display calculi. Properties such as associativity are explicitly stated as structural rules.

Such explicit structural rules make display calculi as modular as Hilbert-style calculi: the logical rules remain constant and different logics are obtained by the addition or deletion of structural rules only. Display calculi therefore provide an extremely elegant sequent framework for “logic engineering”, applicable to many (classical and non-classical) logics in a uniform way [11, 5]. The display calculus $\delta\mathbf{RA}$ [4], for example, captures the logic for relation algebras. 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. Belnap [1] proves that the cut rule is *admissible* in **all** such 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. His proof does not use the standard double induction over the cut rank and degree à la Gentzen.

In [2] we implemented a “shallow” embedding of $\delta\mathbf{RA}$ which enabled us to mimic derivations in $\delta\mathbf{RA}$ using Isabelle/Pure. But it was impossible to reason *about* derivations since they existed only as the trace of the particular Isabelle session. In [9], Pfenning has given a formalisation of cut-admissibility for traditional sequent calculi for various nonclassical logics using the logical framework

* Supported by an Australian Research Council Large Grant

** Supported by an Australian Research Council QEII Fellowship

Elf, which is based upon dependent type theory. Although dependent types allow derivations to be captured as terms, they do not enable us to formalise all aspects of a meta-theoretic proof. As Pfenning admits, the Elf formalisation cannot be used for checking the correct use of the induction principles used in the cut-admissibility proof, since this requires a “deep” embedding [9]. The use of such “deep” embeddings to formalise meta-logical results is rare [8, 7]. To our knowledge, the only full formalisation of a proof of cut-admissibility is that of Schürmann [10], but the calculi used by both Pfenning and Schürmann contain no explicit structural rules, and structural rules like contraction are usually the bane of cut-elimination. Here, we use a deep embedding of the display calculus $\delta\mathbf{RA}$ into Isabelle/HOL to fully formalise the admissibility of the cut rule in the presence of explicit (and arbitrary) structural rules.

The paper is set out as follows. In Section 2 we briefly describe the display calculus $\delta\mathbf{RA}$ for the logic of relation algebras. In Section 3 we describe an encoding of $\delta\mathbf{RA}$ logical constants, logical connectives, formulae, structures, sequents, rules, derivations and derived rules into Isabelle/HOL. In Section 4 we describe the two main transformations required to eliminate cut. In Section 5 we describe how we mechanised these to prove the cut-elimination theorem in Isabelle/HOL. In Section 6 we present conclusions and discuss further work.

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

The following grammar defines the syntax of relation algebras:

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

A display calculus for relation algebras called $\delta\mathbf{RA}$ can be found in [4]. Sequents of $\delta\mathbf{RA}$ are expressions of the form $X \vdash Y$ where X and Y are built from the nullary structural constants E and I and formulae, using a binary comma, a binary semicolon, a unary $*$ or a unary \bullet as structural connectives, according to the grammar below:

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

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

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

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

3 A Deep Embedding of $\delta\mathbf{RA}$ in Isabelle/HOL

In [2], we describe our initial attempts to formalise display calculi in various logical frameworks, and describe why we chose Isabelle/HOL for this work. To make the current paper self-contained, we now describe the Isabelle/HOL data structures used to represent formulae, structures, sequents and derivations. We assume that the reader is familiar with ML and logical frameworks in general.

3.1 Representing Formulae, Structures, Sequents and Rules

An actual derivation in a Display Calculus involves structures containing formulae which are composed of primitive propositions (which we typically represent by p, q, r). It uses rules which are *expressed* using structure and formula variables, typically X, Y, Z and A, B, C respectively, to represent structures and formulae made up from primitive propositions. Nonetheless, in deriving theorems or derived rules we will often use a rule instance where the original variables in the rule are replaced by other variables, rather than actual formulae. We may, for example, have to take the cut rule as shown below left and substitute $B \wedge C$ for A , substitute (Z, D) for X and substitute $C \vee D$ for Y to get the cut rule instance shown below right, and reason about this instance.

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

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

Formulae of $\delta\mathbf{RA}$ are therefore represented by the datatype below:

```
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
```

The constructors `FV` represents formula variables which appear in the statement of a rule or theorem, and which are instantiated to actual formulae of $\delta\mathbf{RA}$ when constructing derivations. The constructor `PP` represents a primitive proposition variable p : once again this lives at the meta-level.

Structures of $\delta\mathbf{RA}$ are represented by the datatype below:

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

The operator **Structform** “casts ” a formula into a structure, since a formula is a special case of a structure (as in the premises of the cut rule given above). The constructor **SV** represents structure variables which appear in the statement of a rule or theorem, and which are instantiated to actual structures of $\delta\mathbf{RA}$ when constructing derivations. Since we must reason about arbitrary derivations, we have to allow derivations to contain structure variables and we must reason about the instantiations explicitly. We therefore cannot use Isabelle’s built-in unification facility for instantiating its “scheme variables” as explained in more detail in [2]. Likewise, formulae of $\delta\mathbf{RA}$ are represented by a datatype which include a constructor **FV** for formula variables which can be instantiated to actual formulae of $\delta\mathbf{RA}$, and a constructor **PP** for a primitive proposition variable p .

The notation in parentheses in the definition of datatype **formula** describe an alternative infix syntax, closer to the actual syntax of $\delta\mathbf{RA}$. 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 omitted.

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

```
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'')**. Functions **premsRule** and **conclRule** return the premise list and the conclusion of a rule respectively. A structure expression is *formula-free* if it does not contain any formula as a sub-structure: that is, if it does not contain any occurrence of the operator **Structform**. A formula-free sequent is defined similarly. The constant **rls** represents the set of rules of $\delta\mathbf{RA}$, encoded using the datatypes just described: we omit the details of this code.

3.2 Handling Substitutions Explicitly

Since a “deep” embedding requires handling substitution explicitly, we now give definitions relating to substitution for structure and formula variables. We first give some type abbreviations, and then the types of a sample of the functions.

```

fSubst = "(string * formula) list"
sSubst = "(string * structr) list"
fsSubst = "fSubst * sSubst"
sFind   :: "sSubst => string => structr"
ruleSubst :: "fsSubst => rule => rule"
seqSubst  :: "fsSubst => sequent => sequent"

```

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 (if any) 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`. There are functions which substitute for every formula or structure variable in a derivation tree (defined below), `rule`, `sequent`, `structure` or `formula`.

3.3 Representing Derivations as Trees

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

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

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

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

Display calculi 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 . We therefore sometimes must treat $A \vdash A$ (where A is a formula variable) as an axiom. Thus the derivation tree `Der ('A' |- 'A') idf []` stands for a finished derivation, which uses the `idfpp` lemma that $A \vdash A$ is derivable for all A , whereas the derivation tree `Unf ('A' |- 'A')` stands for an unfinished derivation with unfinished premise $A \vdash A$.

```

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

```

Fig. 1. Functions for reasoning about derivations

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

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

$$\text{Der } (''A'' \text{ } |- \text{ } PP \text{ } p \text{ } \&\& \text{ } ''A'') \text{ } cA$$

$$\text{[Der } (''A'' \text{ } , \text{ } ''A'' \text{ } |- \text{ } PP \text{ } p \text{ } \&\& \text{ } ''A'') \text{ } ands$$

$$\text{[Unf } (''A'' \text{ } |- \text{ } PP \text{ } p),$$

$$\text{Der } (''A'' \text{ } |- \text{ } ''A'') \text{ } idf \text{ } []]$$

3.4 Reasoning About Derivations and Derivability

In this section we describe various functions which allow us to reason about derivations in $\delta\mathbf{RA}$. The types for these functions are shown in Figure 1.

`allDT f dt` holds if property `f` holds for every sub-tree in the tree `dt`.
`allNextDTs f dt` holds if property `f` holds for every proper sub-tree of `dt`.
`wfb (Der concl rule dts)` holds if sequent rule `rule` has an instantiation with conclusion instance `concl` and premise instances which are the conclusions of the derivation trees in the list `dts`. (“`wfb`” stands for *well-formed*).
`allDT wfb dt` holds if every sub-tree of the derivation tree `dt` satisfies `wfb` (ie, if every node in `dt` is well-formed). Such a derivation is said to be *well-formed*.
`frb rules (Der concl rule dts)` holds when the lowest rule `rule` used in a derivation tree `Der concl rule dts` belongs to the set `rules`.
`allDT (frb rules) dt` holds when every rule used in a derivation tree `dt` belongs to the set `rules`.
`premsDT dt` returns a list of all “premises” (unfinished leaves) of the derivation tree `dt`. That is, the sequents found in nodes of `dt` of the form `Unf seq`.
`conclDT dt` returns the end-sequent of the derivation tree `dt`. That is, the conclusion of the bottom-most rule instance.

So `wfb (Der seq rule dts)` means that the bottom node of the derivation tree `Der seq rule dts` is *well-formed*. We say a derivation tree `dt` is *well-formed* if every node in it is well-formed, and express this as `allDT wfb dt`, since `allDT`

f dt means that property f holds for every sub-tree in the derivation tree dt . Also, $\text{allNextDTs } f \text{ } dt$ means that every proper sub-tree of dt satisfies f .

The property allDT (frb rules) holds when every rule used in a derivation tree belongs to the set rules . The function premsDT 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 finished, 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 1. *A derivation tree dt 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 ?dt ==
  allDT wfb ?dt & allDT (frb ?rules) ?dt & premsDT ?dt = []"
```

We have explicitly added question marks in front of rules and dt 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.

Definition 2 (IsDerivableR). *IsDerivableR rules prems' concl holds iff there exists a derivation tree dt 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 dt.
  allDT (frb ?rules) dt & allDT wfb dt &
  conclDT dt = ?concl & set (premsDT dt) <= ?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, $\text{IsDerivable rules rule}$ holds iff rule may be obtained as a derived rule, from the (unordered) set rules . That is, if rule has premise list prems and conclusion concl , then $\text{IsDerivable rules rule}$ is equivalent to $\text{IsDerivableR rules (set prems) concl}$.

3.5 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 1. *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 appellation “: 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 different sort of transitivity result, relating to a derivation using rules which are themselves derived.

Theorem 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

```

In another reported formalisation of the notion of derivations in a logical calculus [8], 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, [8] defines `IsDerivable` inductively as a relation which is transitive in both the senses of the results above; see the second and third clauses of the definition on [8, 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 incorrect 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.

4 An Operational View of Cut-Elimination

We now give an operational view of cut-elimination, to explain the steps involved in the overall cut-elimination procedure à la Belnap [1]. We assume familiarity with notions like “parametric ancestors” of a cut formula [1].

4.1 Principal Cuts and Belnap’s Condition C8

Definition 3. *An application of (cut) is left-principal [right-principal] if the cut-formula is the principal formula of the left [right] premise of the cut rule.*

Given a derivation (tree) with one principal cut, such as in Figure 2, Belnap’s condition (C8) on the rules of a Display Calculus ensures that the given derivation can be transformed into one whose cuts are on smaller formulae. For example, the principal cut on $A \vee B$ shown in Figure 2 can be replaced by the derivation shown in Figure 3, where $(cs1)$, $(cs2)$, $(\overline{cs1})$ and $(\overline{cs1})$ are two of the display postulates and their inverses respectively. The replacement derivation contains cuts only on A and B , which are smaller formulae than $A \vee B$.

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

Fig. 2. Principal cut on formula $A \vee B$

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

Fig. 3. Transformed principal cut on formula $A \vee B$

There is one such transformation for every connective and this is the basis for a step of the cut-elimination proof which depends on induction on the structure or size of the cut-formula. The base case of this induction is where the cut-formula is introduced by the identity axiom. Such a cut, and its removal, are shown in Figure 4. We return to the actual mechanisation in Section 5.1.

The transformation of a principal 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.

4.2 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

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

Fig. 4. Principal cut where cut-formula is introduced by identity axiom

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

Fig. 5. Making a cut left-principal

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 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 .

Belnap’s conditions guarantee 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 Figure 5. Other conditions of Belnap (*e.g.* 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.

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

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”.

5 Functions for Reasoning About Cuts

We therefore need functions, with the following types, for reasoning about derivations which end with a cut:

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

Each require the bottom node of the derivation tree to be of the form `Der seq rule dts`, and that if `rule` is `(cut)`, then: for `cutOnFmls s` the cut is on a formula in the set `s`; for `cutIsLP A` the cut is on formula `A` and is left-principal; and for `cutIsLRP A` the cut is on formula `A` and is (left- and right-)principal.

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

5.1 Dealing With Principal Cuts

For each logical connective 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. 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 3 (orC8). *Assume we are given a valid derivation tree `dt` whose only instance of cut (if any) is at the bottom, and that this cut is principal with cut-formula $A \vee B$. Then there is a valid derivation tree `dtn` with the same conclusion as `dt`, such that each cut (if any) in `dtn` has A or B as cut-formula.*

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

5.2 Making A Cut (Left) Principal

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, when `b` is `True` [`False`], such differences occur only in succedent [antecedent] positions. For two lists `seq11` and `seq12` of sequents,

`seqReps b X Y seq11 seq12` holds if each n th member of `seq11` is related to the n th member of `seq12` by `seqRep b X Y`.

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

Theorem 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 in the conclusion of a rule is one of Belnap’s conditions [1].

The stronger result `seqExSub2` is similar to `seqExSub1`, except that the antecedent [succedent] of the sequent `pat` may contain a formula, provided that the whole of the antecedent [succedent] is that formula.

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

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

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

Theorem 5 (extSub2). *Suppose we are given a 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`.*

```

extSub2 = "[| 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)))" : thm

```

This theorem is used to show that, when a node of the tree $\Pi_L[A]$ is transformed to the corresponding node of $\Pi_L[Y]$, 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.5); condition (iii) above reflects this limitation.

5.3 Turning One Cut Into Several Left-Principal Cuts

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

Theorem 6 (makeCutLP). *Given cut-free derivation tree $dtAY$ deriving $(A \vdash Y)$ and dtA 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 = "[| allDT (cutOnFmls {}) ?dtAY; allDT (frb rls) ?dtAY;
allDT wfb ?dtAY; conclDT ?dtAY = (?A |- $?Y);
allDT (frb rls) ?dtA; allDT wfb ?dtA; allDT (cutOnFmls {}) ?dtA;
seqRep True (Structform ?A) ?Y (conclDT ?dtA) ?seqY |]
==> EX dtY. conclDT dtY = ?seqY & allDT (cutIsLP ?A) dtY &
allDT (frb rls) dtY & allDT wfb dtY" : thm

```

Function `makeCutRP` is basically the symmetric variant of `makeCutLP`; so $dtAY$ is a cut-free derivation tree deriving $(Y \vdash A)$. But with the extra hypothesis that A is introduced at the bottom of $dtAY$, 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 in this cut-elimination proof. The proofs proceed by structural induction on the initial derivation tree, where the inductive step involves an application of `extSub2`, except where the formula A is introduced. If A is introduced by an introduction rule, then the inductive step involves inserting an instance of (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 $dtAY$ is substituted for $A \vdash A$.

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

Theorem 7 (allLP). *Given a valid derivation tree dt containing just one cut, which is on formula A and is at the root of dt , there is a valid tree with the same conclusion (root) sequent, all of whose cuts are left-principal and are on A .*

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

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

5.4 Putting It All Together

A monolithic proof of the cut-admissibility theorem would be very complex, involving either several nested inductions or a complex measure function. For the transformations above replace one arbitrary cut by many left-principal cuts, one left-principal cut by many principal cuts and one principal cut by one or two cuts on subformulae, whereas we need, ultimately, to replace many arbitrary cuts in a given derivation tree. We can manage this complexity by considering how we would write a program to perform the elimination of cuts from a derivation tree. One way would be to use a number of mutually recursive routines, as follows:

- `elim` eliminates a single arbitrary cut, by turning it into several left-principal cuts and using `elimAllLP` to eliminate them ...
- `elimAllLP` eliminates several left-principal cuts, by repeatedly using `elimLP` to eliminate the top-most remaining one ...
- `elimLP` eliminates a left-principal cut, by turning it into several principal cuts and using `elimAllLRP` to eliminate them ...
- `elimAllLRP` eliminates several principal cuts, by repeatedly using `elimLRP` to eliminate the top-most remaining one ...
- `elimLRP` eliminates a principal cut, by turning it into several cuts on smaller cut-formulae, and using `elimAll` to eliminate them ...
- `elimAll` eliminates several arbitrary cuts, by repeatedly using `elim` to eliminate the top-most remaining one ...

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

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

We define two predicates, `canElim` and `canElimAll`, whose types and meanings (assuming valid trees) are given below. When we use them, the argument `f` will be one of the functions `cutOnFmls`, `cutIsLP` and `cutIsLRP`.

Definition 4. *canElim f* holds for property *f* if, for any valid derivation tree *dt* satisfying *f* and containing at most one cut at the bottom, there is a valid cut-free derivation tree which is equivalent to (has the same conclusion as) *dt*.

canElimAll f means that if every subtree of a given valid tree *dt* satisfies *f*, then there is a valid cut-free derivation tree *dt'* equivalent to *dt* such that if the bottom rule of *dt* is not (cut), then the same rule is at the bottom of *dt'*.

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

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

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

We restate allLP and allLRP using canElim and canElimAll.

Theorem 8 (allLP', allLRP').

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

```
allLP' = "canElimAll (cutIsLP ?A) ==> canElim (cutOnFmls {?A})":thm
allLRP' = "canElimAll (cutIsLRP ?A) ==> canElim (cutIsLP ?A)":thm
```

Now if we can eliminate one arbitrary cut (or left-principal cut, or principal cut) then we can eliminate any number, by eliminating them one at a time starting from the top-most cut. This is easy because eliminating a cut affects only the proof tree above the cut. (There is just a slight complication: we need to show that eliminating a cut does not change a lower cut from being principal to not principal, but this is not difficult). This gives the following three results:

Theorem 9. *If we can eliminate one arbitrary cut (or left-principal cut, or principal cut) from any given derivation tree, then we can eliminate any number of such cuts from any given derivation tree.*

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

We also have the theorems such as `orC8` (see §4.1) dealing with a tree with a single (left- and right-) principal cut on a given formula.

Theorem 10. *A tree with a single (left- and right-) principal cut on a given formula can be replaced by a tree with arbitrary cuts on the immediate subformulae (if any) of that formula.*

These theorems (one for each constructor for the type `formula`) are converted to a list of theorems `thC8Es'`, of which an example is

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

We have one such theorem for each logical connective or constant, and one for the special formulae `FV str` and `PP str`. These latter two arise in the trivial instance of cut-elimination when Y is A and Π_{AY} is empty in Figure 4.

Together with the theorems `elimLRP`, `allLRP'`, `elimLP`, `allLP'` and `elimFmls`, we now have theorems corresponding to the six routines described above. As noted already, a call to `elim` would indirectly call `elim` with a smaller cut-formula as argument, and so the program would terminate, the base case of the recursion being the trivial instance of Figure 4 mentioned above. Correspondingly, the theorems we now have can be combined to give the following.

Theorem 11. *We can eliminate a cut on a formula if we can eliminate a cut on each of the formula's immediate subformulae.*

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

The proof of Theorem 11 is by cases on the formation of the cut-formula, and here we have shown the case for \vee only. There is one such case for each constructor for the type `formula`. We can therefore use structural induction on the structure of a formula to prove that we can eliminate a cut on any given formula `fml`: that is, we can eliminate any cut.

Theorem 12. *A sequent derivable using (cut) is derivable without using (cut).*

Proof. Using `elimFmls` from Theorem 9, it follows that we can eliminate any number of cuts, as reflected by the following sequence of theorems.

```
canElimFml = "canElim (cutOnFmls {?fml})" : thm
canElimAny = "canElim (cutOnFmls UNIV)" : thm
canElimAll = "canElimAll (cutOnFmls UNIV)" : thm
cutElim    = "IsDerivableR rls {} ?concl ==>
              IsDerivableR (rls - {cut}) {} ?concl" : thm
```

Corollary 1. *The rule (cut) is admissible in δRA .*

6 Conclusion and Further Work

We have formulated the Display Calculus for Relation Algebra, $\delta\mathbf{RA}$, in Isabelle/HOL as a “deep embedding”, allowing us to model and reason about derivations 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 were omitted – “due to their difficulty” – from another reported mechanised formalisation of provability [8, p. 302].

We have proved Belnap’s cut-admissibility theorem for $\delta\mathbf{RA}$. This was a considerable effort, and could not have been achieved without the complementary features (found in Isabelle) of the extensive provision of powerful tactics, and the powerful programming language interface available to the user.

The most important disadvantage of our approach is the inability to easily produce a program for cut-elimination from our Isabelle/HOL proofs, even when our proofs mimic a programming style (see §5.4).

Results like `IsDerivableR_trans` and `IsDerivableR_deriv` are general results about the structure of derivations, closely resembling facilities available in Isabelle: namely, successive refinement of subgoals, and use of a previously proved lemma. A higher order framework allows us to reason about higher and higher meta-levels, like the `IsDerivable` relation itself, without invoking explicit “reflection principles” [6]. This is the topic of future work.

References

1. N D Belnap. Display logic. *Journal of Philosophical Logic*, 11:375–417, 1982.
2. Jeremy E Dawson and R Goré. Embedding display calculi into logical frameworks: Comparing Twelf and Isabelle. In C Fidge (Ed), *Proc. CATS 2001: The Australian Theory Symposium*, ENTCS, 42: 89–103, 2001, Elsevier.
3. J E. Dawson and R Gore. A new machine-checked proof of strong normalisation for display logic. Submitted 2001.
4. R Goré. Cut-free display calculi for relation algebras. In D van Dalen and M Bezem (Eds), *CSL96: Selected Papers of the Annual Conference of the European Association for Computer Science Logic*, LNCS 1258:198–210. Springer, 1997.
5. R Goré. Substructural logics on display. *Logic Journal of the Interest Group in Pure and Applied Logic*, 6(3):451–504, 1998.
6. J Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre, 1995.
7. S Matthews. Implementing FS_0 in Isabelle: adding structure at the metalevel. In J Calmet and C Limongelli, editors, *Proc. Disco’96*. Springer, 1996.
8. A Mikhajlova and J 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 1479:295–314. Springer, 1998.
9. F Pfenning. Structural cut elimination. In *Proc. LICS 94*, 1994.
10. C Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Dept. of Comp. Sci., Carnegie Mellon University, USA, CMU-CS-00-146, 2000.
11. H Wansing. *Displaying Modal Logic*, volume 3 of *Trends in Logic*. Kluwer Academic Publishers, Dordrecht, August 1998.