Embedding Display Calculi into Logical Frameworks : Comparing Twelf and Isabelle

Jeremy E. Dawson * and Rajeev Goré **

Formal Methods Group
Dept. of Computer Science
Fac. of Eng. and Inf. Tech.

Automated Reasoning Group
Computer Sciences Laboratory
Res. Sch. of Inf. Sci. and Eng.

Australian National University

jeremy@csl.anu.edu.au

rpg@csl.anu.edu.au

Abstract. We compare several methods of implementing the display (sequent) calculus $\boldsymbol{\delta}$**RA** for relation algebra in the logical frameworks Isabelle and Twelf. We aim for an implementation enabling us to formalise within the logical framework proof-theoretic results such as the cut-elimination theorem for $\boldsymbol{\delta}$**RA** and any associated increase in proof length. We discuss issues arising from this requirement.

Keywords: logical frameworks, higher-order logics, proof systems for relation algebra, non-classical logics, automated deduction, display logic.

1 Introduction and Motivation

Relation algebras [13] are extensions of Boolean algebras; whereas Boolean algebras model subsets of a given set, relation algebras model binary relations on a given set. Thus relation algebras have operations such as relational composition and relational converse. As each relation is itself a set (of ordered pairs), relation algebras also have the Boolean operations such as intersection (conjunction) and complement (negation). Relation algebras form the basis of relational databases [6] and of the specification and proof of correctness of programs.

Display Logic [1] is a generalised sequent framework for non-classical logics, based on the Gentzen sequent calculus [7]. Its advantages include a generic cut-elimination theorem, which applies whenever the rules for the display calculus satisfy certain, easily checked, conditions. It is an extremely general logical formalism, applicable to many (classical and non-classical) logics in a uniform way [10], [21]. The generality of the display framework means that essentially the same meta-level proofs work for many different logics. A rigorous mechanical formalisation of such

* Supported by an Australian Research Council Small Research Grant.
** Supported by an Australian Research Council Queen Elizabeth II Fellowship.

proofs is then widely applicable and worth pursuing. In this paper we discuss the implementation of $\boldsymbol{\delta}\mathbf{RA}$, a display calculus for relation algebras, as a case study for exploring various methods for such a mechanical formalisation of display calculi in general.

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 assumed the comma to be associative, commutative and inherently poly-valent, display calculi make no such implicit assumptions. Properties such as these are explicitly stated as structural rules. For example, $\boldsymbol{\delta}\mathbf{RA}$-sequents are built using a binary comma, a binary semicolon, and unary $*$ and $\bullet$ structural connectives. Thus, whereas Gentzen's sequents $\Gamma \vdash \Delta$ assume that Γ and Δ are comma-separated lists of formulae, $\boldsymbol{\delta}\mathbf{RA}$-sequents $X \vdash Y$ assume that X and Y are complex tree-like structures built from formulae together with comma, semicolon, $*$ and $\bullet$.

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

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

Intuitively, to use this display calculus rule downwards on a sequent $X' \vdash Y'$, everything other than (P, Q) must be moved into the complex structure X on the left of $\vdash$, thereby displaying the structure (P, Q) as the whole of the right-hand side. There are rules which enable any given structure to be displayed. After the rule application we can "undisplay" the moved material back to its original position (reversing the display steps used), so that the sole purpose of this rule is to "rewrite" some (P, Q) to $P \vee Q$ somewhere inside Y'. See [9] for a full account.

Isabelle is an automated proof assistant [17]. Its meta-logic is an intuitionistic typed higher-order logic, sufficient to support the built-in inference steps of higher-order unification and term rewriting. Isabelle accepts inference rules of the form "from $\alpha_1, \alpha_2, \ldots, \alpha_n$, infer β" where the α_i and β are expressions of the Isabelle meta-logic, or are expressions using a new syntax, defined by the user, for some "object logic". An Isabelle user can encode a particular calculus $\mathcal{C}_L$ for some logic L as an "object logic" by using these rule templates to encode the set of inference rules for $\mathcal{C}_L$. For example, if $\mathcal{C}_L$ is a natural deduction calculus, then the α_i and

β will be formulae of L, whereas if $\mathcal{C}_L$ is a sequent calculus, then the α_i and β will be sequents of $\mathcal{C}_L$. Such an encoding is called an "object logic", even though it is a (typically natural deduction or sequent) *calculus* for some particular logic L. In practice most users would build on one of the comprehensive "object logics" already supplied [18].

Twelf [20] is an implementation of the Edinburgh Logical Framework (LF) [11], which is based on a typed λ-calculus with dependent types. Logics are represented using a *judgements as types* principle, where each judgement of the form $\Gamma, x : Q \vdash y : P$ is identified with the types of its proofs. Twelf also accepts inference rules of the form "from $\alpha_1, \alpha_2, \ldots, \alpha_n,$ infer β", But now, the rule is expressed as the declaration of a λ-calculus term of type $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_n \to \beta$. Once again, if the calculus we are trying to capture is a natural deduction calculus, then the α_i and β will be formulae (types), but if the calculus is a sequent calculus, then the α_i and β will be sequents (types).

In an earlier paper [4], we described the Isabelle implementation of $\boldsymbol{\delta}$**RA**, a display calculus for relation algebra [9], as an object logic of Isabelle. In that paper we described how we had used the implementation to prove results comparing alternative formalisations of relation algebra from a proof-theoretic perspective. However we had not proved those results themselves *in* Isabelle, in the sense we now explain.

Suppose we have a logic L (in the sense of a set of formulae which we regard as valid with respect to some semantics) and two calculi $\mathcal{P}$ and $\mathcal{Q}$ for L (each consisting of axioms and inference rules). Further suppose that we have mechanical implementations of $\mathcal{P}$ and $\mathcal{Q}$. Then one can use the implementation of $\mathcal{P}$ to derive in $\mathcal{P}$ every axiom and rule of $\mathcal{Q}$. One could then go outside the mechanical system and argue (by induction on the size, or on the structure, of a $\mathcal{Q}$-derivation) that therefore every $\mathcal{Q}$-derivable object (typically a formula or sequent) is also $\mathcal{P}$-derivable.

The results referred to in [4] were proved in this manner. An alternative and, when one's aim is the mechanical proof of proof-theoretical results, better, approach would be one which enabled the inductive argument above to be carried out in the mechanical theorem-prover. This would require a different style of implementation of the calculi $\mathcal{P}$ and $\mathcal{Q}$ in the prover, so as to enable reasoning about the shape and form of $\mathcal{P}$-derivations and $\mathcal{Q}$-derivations. Typically this would require modelling, in the theorem prover, the "tree" of steps used in a derivation and the fact that each step is an instance of a rule of the calculus $\mathcal{P}$ or $\mathcal{Q}$; this in turn requires modelling the form of the statement of an inference rule of $\mathcal{P}$ or $\mathcal{Q}$, and a method of obtaining particular substitutional instances of such

rules of inference. The terms "shallow embedding" and "deep embedding" are often used to distinguish these styles of implementation [2].

Providing both shallow and deep embeddings leads to the possibility of linking the two implementations to give an efficient theorem prover for **δRA**, in the shallow embedding, but making use of proof-theoretic results proved in the deep embedding. We would then have a theorem prover based directly on machine-checked proof theory.

The only other example of a deep embedding of a logical calculus of which we know is found in Mikhajlova and von Wright [15], in which they compare classical first-order logic calculi. We contrast our approach with theirs in Section 5.

In this paper we describe work done so far towards a "deep embedding" of **δRA**. We explored the possibility of implementation in Twelf, Isabelle/CTT and Isabelle/HOL. In the subsequent sections we describe the original Isabelle/Pure implementation and this further work.

Acknowledgements We are grateful to Paul Jackson, Randy Pollack and Mark Staples for many useful discussions on the notions of "shallow" and "deep" embeddings, and to Frank Pfenning and Larry Paulson for answering our questions about Twelf and Isabelle.

2 The Isabelle/Pure implementation

Isabelle is an interactive computer-based proof system, described in [17]. Its capabilities include higher-order unification and term rewriting. It is written in Standard ML [16]; when it is running, the user can interact with it by entering further ML commands, and can program complex proof sequences in ML. As stated previously, the basic Isabelle constructs available to a user include inference rule templates of the form "from $\alpha_1, \alpha_2, \ldots, \alpha_n$, infer β". These can be used "forwards", to obtain β from the α_i, or "backwards", to reduce a given goal β to subgoals $\alpha_1, \alpha_2, \ldots, \alpha_n$. Isabelle provides a number of basic operations for backwards proof and proof-search (*tactics*), as well as *tacticals* for combining these. Isabelle also supports forward proof.

The inference rules collectively form a simple meta-logic: an intuitionistic typed higher-order logic. There are three logical operators: `==>` (implication, or deducibility), `==` (equality, or substitutability), and `!!` (universal quantification). For example, `[| A1; A2 |] ==> B`, which is an abbreviation for `A1 ==> (A2 ==> B)`, is the Isabelle representation for "from $\alpha_1, \alpha_2,$ infer β". These operators satisfy certain properties arising

from their intended meanings. For example, since `[| A1; A2 |] ==> B` means that β can be deduced from α_1 and α_2, it should also be possible to deduce β from α_2 and α_1. Indeed, using Isabelle, we can prove a new rule that states "if (from $\alpha_1, \alpha_2,$ infer β) is a rule, then (from $\alpha_2, \alpha_1,$ infer β) is a rule", as shown below:

```
[| [| A1; A2 |] ==> B |] ==> ([| A2; A1 |] ==> B).
```

That is, `==>` can be seen as the analogue of the horizontal bar used to state rules in a natural deduction system, in which the order of premises is not significant. Likewise, since `A == B` means that `A` can be replaced by `B`, or vice versa, `==` is reflexive, symmetric and transitive.

This meta-logic is known as Isabelle's Pure theory. Mostly the user would augment this by defining additional constants to capture the syntax of the object logic. For example, to capture set theory, we would add a constant `mem` (say) to stand for the $\in$ symbol, while sequents would require constants '`|-`' and '`,`'. Several such object logics are packaged with the Isabelle distribution [18]. Once these syntactic elements of the object logic are in place, we can build object-logic expressions into the α_i and β. For example, in $\boldsymbol{\delta}$**RA**, the ($\vdash \vee$) rule uses the meta-logical operator `==>` discussed above, and could be entered as

```
"[| $X |- P, Q |] ==> $X |- P v Q".
```

where the `$` distinguishes $\boldsymbol{\delta}$**RA**-structures such as `X` from $\boldsymbol{\delta}$**RA**-formulae such as `P`. This rule is an instance of the form "from α infer β".

However $\boldsymbol{\delta}$**RA** was implemented in Isabelle directly on top of the Pure theory, so that the only inference rules available are those of $\boldsymbol{\delta}$**RA**. As far as the Isabelle meta-logic is concerned, we can think of its atomic propositions as the formulae of the object-logic. In this case these are $\boldsymbol{\delta}$**RA**-sequents of the form $X \vdash Y$, built using object-logic "connectives" '$\vdash$', ',', ';', '*', '$\bullet$', '$\wedge$', '$\vee$', etc. These can be combined into more complex Isabelle propositions using the Isabelle meta-logic connectives `==>`, `==` and `!!`. Thus each Isabelle proposition is either a $\boldsymbol{\delta}$**RA** sequent $X \vdash Y$, or is made up of sequents combined into inference rules using the three Isabelle/Pure logical operators. It follows that expressions such as `X |- (P ==> Q)` or `A v (B ==> C)` are not possible. Note that the α_i and β as discussed above would most commonly simply be $\boldsymbol{\delta}$**RA**-sequents, but could be "complex" Isabelle propositions, such as `X |- P ==> Y |- Q`.

We may declare axioms and inference rules of the $\boldsymbol{\delta}$**RA** calculus to Isabelle. For example, the sequent $p \vdash p$ is an initial sequent (axiom) in $\boldsymbol{\delta}$**RA**; thus it is declared in Isabelle as a rule, `P |- P`, with no premises.

Similarly, each rule of **δRA** is declared as an Isabelle rule, as in the example of the ($\vdash \vee$) rule above. This becomes an Isabelle theorem containing the metalogical implication operator `==>`. That this is a shallow embedding is reflected in the fact that the horizontal bar of sequent rules becomes the Isabelle `==>` operator. In a deep embedding, as in Section 4, the horizontal bar also becomes an object-level constant.

In this shallow embedding, access to and manipulation of the shape of **δRA** constructs (ie, sequents, structures, formulae) and derivations was provided only at the ML level. For example, in [4] we described having programmed a procedure to perform cut-elimination for **δRA**. The input to this was a **δRA** derivation (represented as a tree of sequent rule instances), and the output was a derivation not containing an instance of the cut rule. This required examining the shape of a derivation (represented as a tree of **δRA** rule instances) and of the Isabelle terms representing sequents. While the shape of an Isabelle term is easily accessible (by ML code), a derivation step is represented as a function; a complex derivation is the composition of functions representing the elementary derivation steps. Therefore, to look at the shape of a derivation (for example, to ask what rule was used in the final **δRA** inference step) involved changing the Isabelle code somewhat, so as to record the elementary **δRA** rules used and to construct a derivation-tree while a derivation was being performed.

What we aim to do (as described later) is to perform the cut-elimination proof entirely within a theorem prover (rather than writing a cut-elimination program in ML). We will therefore need to model derivation trees within the language of the theorem prover. We would also need to model the set of elementary inference rules of the calculus, so as to be able to say that every rule in a given derivation tree is an instance of one of the inference rules in that calculus.

In [4] we also described using the Isabelle **δRA** implementation to show the soundness of certain other calculi for relation algebra. In these cases, we in fact showed that all the axioms and rules of the other calculi were derivable in **δRA**. The argument that followed, namely that therefore an entire derivation in one of those other calculi could have been performed in **δRA**, was outside the scope of the mechanical theorem prover.

Again, doing this within the theorem prover would have required modelling the shape of sequents or formulae of the calculus in question, of derivations, and of the rules used in them. This was done by Mikhajlova

and von Wright [15] in their comparison of classical first-order logic proof systems.

3 Dependent Type Theory implementations

3.1 Introduction

In a dependent type theory, a type may be parametrised not only over a type variable, but also over a term variable. This, coupled with the *judgements as types* principle (where a judgement is identified with the types of its proofs) enables us to express the derivation of a display calculus derived rule.

A simpler situation where the Curry-Howard isomorphism can be seen is in the simply-typed λ-calculus and a natural deduction calculus NDInt for intuitionistic propositional logic. Let A and B be formulae, and π be a derivation of A in NDInt. Suppose also that $A \to B$ is derivable in NDInt, and consider an NDInt-derivation of $A \to B$ where one assumes A and derives B. That is, it is a derivation of B in which A is regarded as true; at the points where A is used, it would contain some notation ξ meaning "true by assumption". Let $\rho(\xi)$ be this derivation of B. So if we substitute π (the derivation of A) for ξ in $\rho(\xi)$, we get a derivation $\rho(\pi)$ of B which does not rely on A as an assumption.

Then, using $\pi : A$ to mean "π is a derivation of A", we can write the rule

$$\frac{\pi : A \qquad \rho(\xi) : A \to B}{\rho(\pi) : B}$$

Erasing the annotations π, ρ and ':' gives the well-known $(\to E)$ natural deduction rule for intuitionistic logic; the rule as it stands also shows how to derive B. As indicated above, we can equally think of $\rho(.)$ as a derivation of $A \to B$ or as a function which takes, as argument, a derivation of A such as π, and returns a derivation of B. If we also think of A as the type "derivations of proposition A" (and similarly for B) we get the functional type-theory interpretation, under which $A \to B$ means the type of functions which take an argument of type A and return a result of type B. (Conveniently, the same symbol '$\to$' is conventionally used for both purposes.)

Dependent types extend what can be done in the simply-typed λ-calculus in two distinct ways. Firstly, consider a type I of individuals, and a parametrised type $A(i)$; that is, for each individual $i \in I$, there is

a type $A(i)$. Consider a function ρ whose domain is I, such that for each $i \in I$, $\rho(i) \in A(i)$. The type of such a function is written $\prod i : I.A(i)$ (denoting how the *type* of $\rho(i)$ *depends* on the *value* of i).

Now let A be a predicate, and therefore each $A(i)$ be a proposition (taking the value true or false), and let each $\rho(i)$ be a derivation of $A(i)$. That is, ρ is a function which, for each individual i, gives a derivation of $A(i)$; we can think of such a ρ as a derivation of $\forall i A(i)$. At the same time, if we think of each type $A(i)$ as the type "derivations of proposition $A(i)$", then the functions ρ of type $\prod i : I.A(i)$ are the derivations of $\forall i A(i)$.

Secondly, we install an object logic at the level of terms, so that the expressions (formulae or sequents) of the object logic become terms of the dependently-typed λ-calculus. That is, we can install a syntax for terms using the syntax of the chosen object logic, and declare a type constructor $P(.)$, so that for each term t, $P(t)$ will mean "derivation of expression t". This was our approach for modelling $\boldsymbol{\delta}$**RA**.

Regarding a formula t as derivable when we have a term of type $P(t)$ has certain consequences. For example, since from functions of type $X \to Z$, $U \to V \to W$ and $S \to S \to T$ we can construct functions of type $X \to Y \to Z$, $V \to U \to W$ and $S \to T$, we unavoidably have weakening, exchange and contraction. Note, however, that in our formulation of $\boldsymbol{\delta}$**RA**, these are at the meta-logic level, not in the object logic.

Two theories based on dependent types are the Edinburgh Logical Framework (LF) [11] and Constructive Type Theory (CTT) [14].

3.2 The Twelf implementation

Twelf [20] is an implementation of the Edinburgh Logical Framework (LF) [11]. This is based on a typed λ-calculus with dependent types. Twelf is written in Standard ML, and the user can interact with the system using its "ML Interface", but only a very limited range of functions is available. Figure 1 is an extract of our Twelf source file.

Line 1 declares `str` as a new type to represent the $\boldsymbol{\delta}$**RA** structures. Line 2 declares `|-` as a binary type constructor, taking two *term* arguments of type `str` and returning a *type*. If we followed the general description above, we would declare `seq` as a new type to represent the $\boldsymbol{\delta}$**RA** sequents, and `P` as a function taking a *term* argument of type `seq` and returning a *type*, thus:

```
1  str : type.
2  |- : str -> str -> type. %infix none 10 |-.
3  * : str -> str. %prefix 200 *.

4  % Display postulates
5  sA : (* X |- Y) -> (* Y |- X).
6  sS : (X |- * Y) -> (Y |- * X).

7  % Derived structural rules
8  ssASl = [D] sA (sS D). % (X |- * * Y) -> (* * X |- Y)
```

Fig. 1. Sample Twelf source code

```
seq : type.
|- : str -> str -> seq. %infix none 10 |-.
P : seq -> type.
```

The actual code in Figure 1 abbreviates this by removing the type constructor `P` and (in effect) changing `|-` from a term constructor to a type constructor. Thus, for structures `X` and `Y`, the construct `X |- Y` represents derivations of the sequent $X \vdash Y$. Line 3 declares `*` as a unary structure operator. Lines 4 and 7 are comments.

In lines 5 and 6 the terms `sA` and `sS` are declared with the types that represent derivations of the $\boldsymbol{\delta}$**RA** rules (see [9]) shown below:

$$\frac{*X \vdash Y}{*Y \vdash X}\,(\texttt{sA}) \qquad\qquad \frac{X \vdash *Y}{Y \vdash *X}\,(\texttt{sS})$$

We can think of these declarations as assertions that there are derivations (which we name `sA` and `sS`) of these rules, or as defining these as "primitive" rules (one-step derivations).

Now, treating `sA` and `sS` (which stand for "star in the antecedent" and "star in the succedent") as functions each taking a derivation of a sequent to a derivation of another sequent, they can be composed for some given sequent $U \vdash * * V$ as follows

$$\begin{array}{c}\text{derivation of}\\ U \vdash * * V\end{array} \quad \xrightarrow{\texttt{sS}} \quad \begin{array}{c}\text{derivation of}\\ *V \vdash *U\end{array} \quad \xrightarrow{\texttt{sA}} \quad \begin{array}{c}\text{derivation of}\\ * * U \vdash V\end{array}$$

The definition of `ssSAl` does this; the notation means $\lambda D.sA(sS\,D)$. Twelf computes the type of the function `ssSAl`, giving $(S1 \vdash * * S2) \to (* * S1 \vdash S2)$. Note that $X, Y, S1$ and $S2$ are variables ($S1$ and $S2$ being names chosen by Twelf).

In a manner akin to Prolog [5], Twelf offers the facilities to make a query such as

```
_ssASl : (X1 |- * * Y1) -> (* * X1 |- Y1).
```

which searches for a term of the specified type (ie, searches for a derivation of the stated rule) and instantiates `_ssASl` with that term. In this example, because Twelf allows substituting for `X1` and `Y1` in the query as well as for the variables `X` and `Y` in the declarations of `sA` and `sS`, the search proceeds along an infinite branch in the wrong direction.

On the other hand, the code

```
% Using the theorem prover
%theorem
ssASl_th : exists
  {D:{S1:str} {S2:str} (S1 |- * * S2) -> (* * S1 |- S2)} true.
%prove 2 {} (ssASl_th D).
```

successfully uses Twelf's theorem prover to find the derivation, returning

```
 /ssASl_th/:
   ssASl_th ([S1:str] [S2:str] [X1:S1 |- * * S2] sA (sS X1)).
```

(Here the variables `S1` and `S2` need to be explicitly abstracted over – in the earlier code they were free variables).

Twelf's theorem prover is not extensively documented, and is stated to be under active development, with the proof search component expected to undergo major changes. The proof search strategy can be controlled by the user to only a very limited extent, whereas we have found that considerable control by the user (using tactics differing from proof to proof) has generally been necessary in our work so far.

We found that certain aspects of Isabelle which we very much appreciated are absent from Twelf. Isabelle offers a substantial number of user "commands", in the form of documented ML functions, which enable the user to programme proof procedures, or to examine the shape of a term or type expression. For example, one might write a tactic which explicitly examines the current proof state and then decides which of several tactics to apply. There are many and expressive "glue" functions for combining tactics in Isabelle. Inadequacies in the documentation, or in the selection of functions documented, can often be circumvented by looking at the source code, which can also be a great help to the user in programming his/her own tactics.

Twelf appears to offer no comparable capabilities to the user. Thus, although the theorem prover did successfully find the derivation above, we felt uncertain that we would be able to use it to find all required derivations.

3.3 The Cut-Elimination Theorem in Twelf

At this point we should refer to the proof of a cut-elimination theorem, using Twelf, described in [19]. This uses a rather ingenious representation of sequents; a cut-free proof of a traditional Gentzen sequent $A \vdash B$ is represented as the type `neg A -> pos B -> @`, and a cut-free proof of it as `neg A -> pos B -> #`. The two rules shown below left are represented as the type shown on the right:

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A\&B \vdash \Delta} \qquad \frac{A \vdash}{A\&B \vdash} \qquad \texttt{(neg A -> \#) -> (neg (A and B) -> \#)}$$

In fact, the second rule shown would be directly represented by the type shown, but the first rule shown could be directly represented by

```
(neg Γ -> pos Δ -> neg A -> #) ->
   (neg Γ -> pos Δ -> neg (A and B) -> #)
```

However the existence of a term (function) of the type shown first trivially implies the existence of a term (function) of the type shown second. Further constructed types represent stages of the transformation of a proof into a cut-free proof, and a termination checker checks that the function performing the entire transformation from a proof to a cut-free proof terminates. However, this termination checker only checks that the function would not run forever. It does not check that it terminates with a cut-free derivation – it does not complain if the code is run with some cases deleted.

This work on the cut-elimination theorem is based on an ingenious way of representing the problem in a way that fits into Twelf; even so, getting a proof of the cut-elimination theorem from it relies on the heuristically-based termination checker and upon a manual check that all possible cases for the rules used just above the "cut" have been covered. On the other hand, the Twelf type-checker proves that each individual step is correct, in that the changed derivation in fact does derive the sequent which it purports to derive.

Thus this proof of cut-elimination lies between our previous work (where we just programmed a cut-elimination function in ML) and a fully formal proof.

3.4 The Isabelle/CTT implementation

We then turned to the `CTT` theory of Isabelle, since it is based on a logic which is very similar to that of Twelf, in that it includes dependent types.

We implemented **δRA** similarly to its implementation in Twelf, so that we had, for example, (omitting premises of the form `?X : str`, which were fairly ubiquitous)

```
"sA : * ?X |- ?Y --> * ?Y |- ?X" : thm
"sA oo sS : ?X |- * * ?Y --> * * ?X |- ?Y" : thm
```

where `oo` denotes function composition. Tactics were written to determine the type of a term (ie, to determine the derived rule obtained by a given combination of rules), by solving a goal such as `sA oo sS : ?t` (which contains the type variable `?t`). We explored tactics to search for a term of a given type (ie, a proof of a given term), by solving a goal such as the one below, which contains the term variable `?P`.

```
"?P : ( X |- * * Y) --> ( * * X |- Y)".
```

The `CTT` theory of Isabelle is not extensively developed as some other theories are; we found it necessary to prove some fairly elementary though general theorems.

3.5 Conclusions

We experimented with Twelf because it had appeared that we get our hands on the proofs "for free", in that in the course of deriving a sequent or sequent rule, we produce a term which embodies the derivation performed. Given our intention to do a fully mechanised proof of the cut-elimination theorem (in which we manipulate derivations extensively) it had seemed that this feature of Twelf would be useful. Our work with Isabelle/CTT seemed to indicate that the same things could be done in it as in Twelf (with some effort to produce appropriate derived rules and tactics).

However we realised that although we could produce a term which represented the derivation and showed the elementary steps from which it is composed, we could not analyse derivations, *in the logic of the theorem prover*, in the required ways: for example, to ask which rule was used in the final step of the derivation. In the light of this, there seemed to be no benefit in pursuing an implementation using dependent type theory.

We have referred to a proof in Twelf of the sequent calculus cut-elimination theorem, noting that it was based on an ingenious way of representing the problem in a way that fits into Twelf, and that it was not a fully formal proof. Since we are aiming for a fully formal proof, using techniques which would be applicable equally to other proof-theoretic results, we felt that this cut-elimination proof does not indicate that dependent type theory would be powerful enough for our needs.

4 The Isabelle/HOL implementation

`HOL` is an Isabelle theory based on the higher-order logic of Church [3] and the HOL system of Gordon [8]. Thus it includes quantification and abstraction over higher-order functions and predicates. The `HOL` theory uses Isabelle's own type system and function application and abstraction (that is, object-level types and functions are identified with meta-level types and functions). Isabelle/HOL contains constructs found in functional programming languages (such as `datatype` and `let`) which greatly facilitates re-implementing a program in Isabelle/HOL, and then reasoning about it. However limitations (not found in, say, Standard ML itself) prevent defining types which are empty or which are not sets, or functions which may not terminate.

The work of embedding $\boldsymbol{\delta}\mathbf{RA}$ in Isabelle/HOL is still underway. However it is clear at this stage that the facility for `datatype` type declarations and associated primitive recursive function definitions is of enormous help. For example, we model $\boldsymbol{\delta}\mathbf{RA}$ structure expressions (see [9]) as follows

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

The first six lines correspond to the structure operators of $\boldsymbol{\delta}\mathbf{RA}$ and the next line is for "casting" a formula of $\boldsymbol{\delta}\mathbf{RA}$ as a structure of $\boldsymbol{\delta}\mathbf{RA}$. The last line had no analogue in the shallow embedding, where we just used an Isabelle variable such as `?S` to refer to any $\boldsymbol{\delta}\mathbf{RA}$-structure. In the deep embedding, `SV ``X`` ` will refer to the structure variable named `X` appearing in (say) the statement of a rule. We therefore also will need

to define functions (in Isabelle/HOL) which (for example) find all the variables in a structure expression, or substitute a given structure for such a variable. We wrote such functions for the work described in Section 2, but in Standard ML, not in Isabelle.

On the other hand, when writing Isabelle tactics which involve analysing the shape of a rule, we will be able to write (for example) `((SV ?V), ?S)` to match any structure expression consisting of two substructures joined by the comma operator (infix '`,`' is alternative notation for prefix `Comma`), of which the first must be a variable. This matching and any related substitution is performed by Isabelle.

For describing the structure of a derivation we have

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

where `Pr seq rule pts` is a derivation of the sequent `seq`, the last step of the derivation uses the $\boldsymbol{\delta}$**RA** rule `rule`, and the premises of that last step are the conclusions of the derivations in the list `pts`. `Unf` ("unfinished") means that a leaf is a sequent which is not an axiom, so is an assumption (or a "premise") of the derivation tree as a whole. We define functions which check that such a structure is a valid derivation (for example, that the premises of an instantiated rule really are the conclusions of the trees above, and that each step uses a legitimate rule). For example, to list the "premises" of the whole derivation tree:

```
primrec
  "premsPT (Unf seq) = [seq]"
  "premsPT (Pr seq rule pts) = premsPTs pts"

  "premsPTs [] = []"
  "premsPTs (pt # pts) =
            (premsPT pt @ premsPTs pts)"
```

Each datatype declaration generates a number of theorems which are made available for proofs, and of which some are installed in rule sets used by the more automated proof tactics. These theorems include ones expressing that a `datatype` type is a disjoint union and that the constructors are 1-1 functions, and one expressing a structural induction principle.

We give examples of some results proved so far. Here `IsProvableR rules prems concl` means that sequent `concl` can be derived from sequents `prems` using rules `rules`, and `IsProvable rules rule` means that the conclusion of the statement of `rule` may be derived from its premises using `rules`.

```
val IsProvableR_trans =
  "[| IsProvableR rules prems' concl ;
     ALL p:prems'. IsProvableR rules prems p |] ==>
  IsProvableR rules prems concl"

val IsProvableR_deriv =
  "[| ALL rule:rules'. IsProvable rules rule ;
      IsProvableR rules' prems concl |] ==>
  IsProvableR rules prems concl"
```

While we think that the corresponding results, for particular instances of the sequents and derivations involved, could be proved more easily in Twelf than in Isabelle, we cannot see how to express the general result in Twelf or Isabelle/CTT.

5 Further work and Related Work

As noted above, this work is still in progress. We need to specify and reason about derivability and derivation trees; for example, we need to express the requirement that each step of a derivation uses a substitution instance of a $\boldsymbol{\delta}$**RA** rule found in a given set. Reasoning about derivability and derivation trees is rather intricate. We note that in other work of this nature [15, p. 302] the authors chose to state, rather than prove, principles concerning the transitivity of provability, and provability from derived rules. (See the second and third clauses of the definitions given at [15, p. 302]). We have been able to prove corresponding rules from our definition of a proof tree. These are the theorems `IsProvableR_trans` and `IsProvableR_deriv` given near the end of Section 4. Proving rather than stating such rules is preferable as it avoids the possibility of stating an unsound rule. Interestingly, we doubt the suitability of the requirement for a derived rule used in [15], in the third clause of their definition of `IsProvable`. Looking at the code given in the paper, the phrase

```
(?inf. CorrectRuleInst {r} inf /\
    IsProvable rSet (InfHyps inf) (InfConcl inf))
```

suggests requiring that only one (not all) of the instances of the derived rule `r` be provable, which would allow using `P ==> Q` as a derived rule when `False ==> True` is provable.

It appears at this stage that the Isabelle/HOL logic is well-equipped for writing specifications which describe the process of performing proofs

in a given logical calculus. It is easy to specify the behaviour of a program written in functional programming style, and for reasoning about it. It is clearly a suitable choice for this project.

Another concept requiring further exploration is using the types of Twelf or Isabelle/CTT to express more than just the sequent being proved. For example, one might let $t : P(X \vdash Y, \rho, n)$ denote that t is a derivation of $X \vdash Y$, of length at most n, where the last rule used is ρ. Clearly, however, to get all the expressiveness we need in this way would require far more complex type expressions than in this example.

We aimed to produce a "deep" embedding of Display Calculi, sufficient (for example) to enable a mechanised proof of the cut-elimination theorem. While it is clear that we have done this (in a sense), our work has elucidated some further points.

Belnap's cut-elimination theorem for Display Calculi is generic. It applies to all Display Calculi whose rules satisfy certain conditions. Our work has been to model a particular display calculus, $\boldsymbol{\delta}$**RA** (for relation algebra). Accordingly, we will get a proof of the cut-elimination theorem for $\boldsymbol{\delta}$**RA**. While we will be able to identify the properties of the $\boldsymbol{\delta}$**RA** rules used in the proof, and match these with Belnap's conditions, we will not have proved the theorem as stated by Belnap (which is stated in terms of *any* Display Calculus). To model an arbitrary Display Calculus in Isabelle/HOL would be a further (and harder) activity.

Secondly, Belnap's conditions on the rules of the Display Calculus are listed by Kracht [12], who states "These conditions actually need some exegesis". Our work will produce a precise, formal, statement of those properties of the rules which we actually use in the proof of the theorem.

6 Conclusion

We have reviewed earlier work which described a shallow embedding of the $\boldsymbol{\delta}$**RA** calculus into the Isabelle/Pure logic, and noted its drawbacks for proving proof-theoretic results in Isabelle itself. Attracted by the notion that the tools based on dependent type theory would give expressiveness about proofs "for free", we then looked at the Twelf tool. While the Twelf tool is still under active development, we noted how a similar logic is available in Isabelle/CTT, which offers the advantages of Isabelle, such as access to a wide range of documented ML functions for programming proof tactics. However we also found that, while Twelf and Isabelle/CTT facilitate identifying the derivation of a sequent or derived rule, they do not facilitate reasoning about the shape of that derivation, or of the

sequents found in it. Thus we turned to Isabelle/HOL, which appears to offer the best facility for deep embedding of a calculus such as $\boldsymbol{\delta}\mathbf{RA}$, and reasoning about derivability, derivation trees and the associated proof-theoretic concepts we will need. Needless to say, this was the advice we received from Larry Paulson from the start.

References

1. N D Belnap. Display logic. *Journal of Philosophical Logic*, 11:375–417, 1982.
2. Richard J Boulton, Andrew D Gordon, Mike J C Gordon, John R Harrison, John Herbert, and John van Tassel. Experience with embedding hardware description languages in hol. In V Stavridou, T F Melham, and R T Boute, editors, *Proceedings of IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, IFIP Transactions volume A-10, pages 129–156. North-Holland/Elsevier, 1992.
3. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
4. J Dawson and R Goré. A mechanised proof system for relation algebra using display logic. In *JELIA98: Proceedings of the European Workshop on Logic in Artificial Intelligence*, LNAI 1489, pages 264–278. Springer, 1998.
5. P Deransart, A Ed-Dbali, and L Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.
6. Ramez A Elmasri and Shamkant B Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, 2 edition, 1994.
7. J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. John Wiley and Sons, 1987.
8. M J C Gordon and T F Melham. *Introduction to HOL: a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
9. R Goré. Cut-free display calculi for relation algebras. In D van Dalen and M Bezem, editors, *CSL96: Selected Papers of the Annual Conference of the European Association for Computer Science Logic*, LNCS 1258, pages 198–210. Springer, 1997.
10. R Goré. Substructural logics on display. *Logic Journal of the Interest Group in Pure and Applied Logic*, 6(3):451–504, 1998.
11. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40:143–184, 1993.
12. M Kracht. Power and weakness of the modal display calculus. In H Wansing, editor, *Proof Theory of Modal Logics*, pages 92–121. Kluwer, 1996.
13. R D Maddux. Introductory course on relation algebras, finite-dimensional cylindric algebras, and their interconnections. In H. Andreka, J. D. Monk, and I. Nemeti, editors, *Algebraic Logic*, volume 54 of *Colloquia Mathematica Societatis Janos Bolyai*, pages 361–392. North-Holland, Amsterdam, 1991.
14. Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.
15. 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.
16. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

17. Lawrence C Paulson. *The Isabelle Reference Manual*. Computer Laboratory, University of Cambridge, 1995.
18. Lawrence C Paulson. *Isabelle's Object-Logics*. Computer Laboratory, University of Cambridge, 1995.
19. Frank Pfennig. Structural cut elimination. In D Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166. IEEE Computer Society Press, 1995.
20. Frank Pfennig and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, LNAI 1632, pages 202–206. Springer, 1999.
21. Heinrich Wansing. *Displaying Modal Logic*, volume 3 of *TRENDS IN LOGIC*. Kluwer Academic Publishers, Dordrecht, August 1998. Hardbound, ISBN 0-7923-5205-X.