

An Automatic Prover for Sequent Calculus in *Isabelle*

University of St Andrews Computer Science Research Report
Peter Chapman

Using the templates given in the *Isabelle* release [2], it is a relatively simple matter to encode rules for various sequent calculi. The only problems arise with the representation of sequents as lists, which require extra structural rules to be encoded to emulate multisets, most notably exchange. This gives a formulation of the systems more closely related to Gentzen’s original system, which used sequences and hence required explicit Exchange rules.

This work was done in the *Isar* framework of *Isabelle* [4], which hides the ML code away. Whilst this has the benefit of flattening the learning curve, should we wish to do anything non-standard it becomes more than a trivial matter. An *Isar* command usually corresponds to several *Isabelle* commands, which has the dual effect of making most tasks easier, but certain tasks more difficult. Unfortunately, this work falls into the latter category.

1 The Rules

Suppose we wish to encode the rule from **G3c** for disjunction on the right of the sequent arrow (see, for instance [3]):

$$\frac{\Gamma \Rightarrow A, B, \Delta}{\Gamma \Rightarrow A \vee B, \Delta} R\vee$$

We need to be careful about the ordering of the formulae on the right if we do not have multisets at our disposal. If we encode this naïvely, *Isar* will only be able to analyse a disjunction on the right if it is the *first* formula in the succedent. Clearly this is not very helpful. We want to be able to analyse a formula wherever it may be. Luckily, this is possible. The rule in *Isar* looks like this:

```
disjR: ‘‘$H |- $F, A, B, $G ==> $H |- $F, A | B, $G’’
```

where the notation is as follows

- $\$H$ is a sequence of formulae, which is permitted to be empty
- $\mid-$ is the sequent arrow (\Rightarrow)
- \implies separate premisses from conclusions

Notice that there are still problems with the premiss for this rule; we require that A and B are adjacent in the sequence, and in the order A, B . For our purposes, however, this is acceptable, since we are performing proof-search, and so will start from conclusions and work up to premisses. More care must be taken when writing human-readable proof scripts, one of the nice features of *Isar*, since we create derivations from the axioms down to the conclusion.

Two premiss rules are formulated in *Isar* as follows

```
conjR: ‘ ‘ [ | $H | - $F, A, $G ; $H | - $F, B, $G | ] ==> $H | - $F, A & B, $G ’ ’
```

2 Proof Scripts

Once we have encoded all the rules for our calculus of choice, we now want to be able to perform some proofs in the system. In *Isar*, we apply our rules bottom-up using the command `apply (rule NAME)`. For instance, say we wanted to show, in **G3cp**, the sequent “ $\Rightarrow A \wedge B \supset B \wedge A$ ” is valid, we would proceed as follows; use the rule $R\supset$ to get “ $A \wedge B \Rightarrow B \wedge A$ ”, then the rule $L\wedge$ to obtain “ $A, B \Rightarrow B \wedge A$ ”, then the rule $R\wedge$ to get the pair of sequents “ $A, B \Rightarrow B$ ” and “ $A, B \Rightarrow A$ ”, which are axioms. In *Isar*, exactly the same process would be followed:

```
lemma ‘ ‘ | - A & B -> B & A ’ ’
  apply (rule impR)
  apply (rule conjL)
  apply (rule conjR)
  apply (rule basic)
  apply (rule basic)
done
```

where `basic` is the rule representing the use of an axiom. We have to apply `basic` twice because we have two *subgoals* corresponding to the two premisses of `conjR`; the first use of `basic` dispenses with the sequent $A, B \Rightarrow B$, the second with the remaining premiss.

3 Towards Automation

Obviously we want to not perform these proofs by “hand”, even with the assistance of a computer. We wish to be able to give *Isar* the instruction to find out, with no prompting from the user, if a given sequent is valid in the given system. Obviously this is only possible should derivability be decidable. For this purpose, we can combine rules. In *Isar*, the adding of the symbol `?` after a rule means we try the rule, giving back the premisses if it is applicable, otherwise returning the unchanged conclusion. Similarly, we have the symbol `+`, which repeats a rule at least once; it fails should the rule not be applicable once. Putting these two together, we can create a new rule which attempts a rule as many times as it is applicable:

```
apply ((rule impR)+)?
```

when applied to the sequent $\Rightarrow A \supset A \supset A \supset A$ returns the sequent $A, A, A \Rightarrow A$, whereas applied to $A, B \Rightarrow A \wedge B$ would return the same sequent.

We can also combine two rules together to be performed in sequence simply by separating them with a comma within the `apply` command. So, the statement `apply (rule impR, rule basic)` would apply the rule `impR` followed by `basic`, failing if either one is not applicable to the current sequent. We can also combine rules using the additional symbols given above to link together a chain of rules, each of which will be tried and repeated as many times as possible.

However, this is where the problems begin in the use of *Isar*. This formulation gives an implicit ordering to the rules, and this can cause problems. Much better is to delve into the *Isabelle* language which is behind the scenes to achieve the finer subtleties which are available there. We can circumvent the problems of ordering, and also give more explicit commands to the system to obtain the behaviour we want.

3.1 *Isabelle* to *Isar*, and back

We can insert *Isabelle* code directly into *Isar* by enclosing it in `{* *}`. This means we can now readily use the greater subtlety of *Isabelle*, whilst in the more user-friendly and human-readable world of *Isar*. In particular, because *Isabelle* has numbered sub-goals, we can directly address these sub-goals, something which is not possible in *Isar*. We can also rename complicated tactics, to combine several at without there being many lines of text. We must note, however, that *Isabelle* refers to our rules as theorems, whereas *Isar* refers to them as rules. The most important consequence of this is that

we need to enclose our rule names in quotes, so that rather than writing `apply (rule basic)` we need to write `apply (tactic (thm ‘‘basic’’))` within an ML environment.

The method we wish to employ tries out a series of rules on the current sequent, and returns the premiss(es) of the rule if it is applicable. This is done using the `FIRST[rule1,rule2,...]` command. It makes most sense to order the possible application of rules so that any two premiss rules are tried last; this will reduce the need for many subgoals to be dealt with at once. We also wish to apply the rule *basic* at the end of each cycle of rule applications, to dispense with leaves of the deduction tree. It is also desirable to restrict the use of non-invertible rules to a time when all other possibilities have been tried, to reduce the need for back-tracking, and loop-checking.

3.2 An Example - G3cp

G3cp is a useful system with which to begin; all the rules are invertible, and the validity of a sequent is decidable. With this in mind, we simply place all the two premiss rules at the end of the rule loop, along with the terminating rules *basic* and *false*. With the renaming of the tactic, this can be condensed to one line. We simply write, in the file with all of our rules, the following:

```
ML {* val BIGRULE = REPEAT (FIRST[  rtac (thm ‘‘impR’’) 1,
                                   ...
                                   rtac (thm ‘‘basic’’) ]) *}
```

To invoke this automated procedure, we type `apply (tactic BIGRULE)`. For instance, if we wanted to check the classical validity of the law of double negation $A \Leftrightarrow \neg\neg A$, the following proof script would be sufficient

```
lemma ‘‘|- A <-> ¬¬A’’
  apply (tactic BIGRULE)
  done
```

This is because **BIGRULE** returns no sub-goals. If, however, the formula is not classically derivable, such as $(A \vee B) \supset (A \wedge B)$, then **BIGRULE** will return the subgoal to which nothing can be done, which in this case returns the subgoals

1. B |- A
2. A | B |- B

4 Backtracking

The previous example is, as stated, the most simple case. We now consider systems which do not have the clean properties of **G3cp**. First, we investigate the problem of *backtracking*. This arises when we consider intuitionistic systems that require only a single formula on the right of a sequent arrow. Whereas in the classical cases we could decompose a disjunction on the right into the two constituent sub-formulae, now we have to make a choice between either the right subformula, or the left:

$$\frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A \vee B} \vee R_1 \quad \frac{\Gamma \Rightarrow B}{\Gamma \Rightarrow A \vee B} \vee R_2$$

This obviously can cause problems; whilst it may be obvious for us which rule to use, for instance in a sequent such as $A \Rightarrow A \vee B$, the *Isabelle* tactic must use backtracking. This means it tries using one rule, then if this does not lead to a solution, attempts the other. Unfortunately, this method quickly leads to an explosion in the number of cases to be attempted; if there are n disjunctions in a formula on the right, in the worst case we need to examine 2^n different cases.

There are two different methods which can be adopted for proof search. The first of these is called *breadth-first search*. This involves enumerating both cases after analysing one disjunction, then checking them for a solution. If neither has an immediate solution, we then branch again from a disjunction, creating four cases, and check each of these for a solution. This process continues until we find a solution, or have exhausted all possibilities for one. Clearly, this will use a lot of space, and in large applications quickly becomes unfeasible for a search tactic. It will, however, find a solution if one exists.

The other method available is known as *depth-first search*. Here we pick a right disjunction rule, and search down the branch that this creates, at each valid stage using another disjunction rule, until we either find a solution or can no longer apply any other rules. Then, we backtrack to the last used disjunction rule, and apply the other disjunction rule to the conclusion, and so on. In this way, we do not use much space, but we can also get “trapped” in an infinite branch. We must be careful to encode our tactic to avoid this, if possible.

4.1 The *Isabelle* Tacticals Used

Isabelle provides for both kinds of searching. The one which we will use is depth-first, because we have the tactical `DEPTH-SOLVE`, which searches for

states which have no subgoals. In other words, it searches for a solution, with the tactic failing if a solution cannot be found. We could have used this tactical for **G3cp**, but it would have been unnecessary. We need now, however, to use it, and moreover we need to make sure that we are using it correctly, because, as previously stated, it can lead to infinite searches in the worst case.

The tactic we pass to **DEPTH-SOLVE** will not be the same as the tactic from the previous section for this very reason; the **REPEAT** command will be extremely harmful if a solution does not exist, since *Isabelle* will simply keep attempting to apply the system rules without success. So, we drop the repeat command. We also need to include our two rules for disjunction. A naïve application, say simply adding the two rules to the list, would result in only one rule ever being applied; if we have a disjunction on the right either rule is applicable, only the first rule which is encountered in the list will ever be used. We need to make sure that *both* are considered, which we do by use of the **APPEND** tactical. This gives all possible next states from the application of a pair of tactics, and so will create the two states that the **DEPTH-SOLVE** needs to operate.

4.2 An Example - G4ip

G4ip [1] has no need for a loop checker, but does require backtracking during proof search. It is still decidable. The tactic we write to create our proof checker is

```
ML {* val G4ipRULE = DEPTH-SOLVE (FIRST[ rtac (thm ‘‘impR’’) 1,
                                         ...
                                         (rtac (thm ‘‘disjR1’’) 1
                                         APPEND rtac (thm ‘‘disjR2’’) 1),
                                         ]) *}
```

In contrast with the tactic for **G3cp**, this now fails if there is no solution. Therefore, we do not see the uppermost leaves that the tactic returns to which no more analysis can be done. This is unfortunate, however it would be difficult to implement a rule which returned such leaves, since there could be many different collections of leaves, each stemming from different combinations of the right disjunction rules used.

5 Loop Checking

The next problem we encounter is rules which are non-terminating. The system which exhibits this property is **G3ip**, and specifically the rule $L\supset$. This is because we copy the principal formula from the antecedent of the conclusion to the antecedent of the first premiss:

$$\frac{\Gamma, A \supset B \Rightarrow A \quad \Gamma, B \Rightarrow C}{\Gamma, A \supset B \Rightarrow C} L\supset$$

Due to the duplication of the formula $A \supset B$, we can always apply the rule $L\supset$, which is obviously harmful, because it will manifest itself, within *Isabelle*, as an infinite loop. There are two cases to consider; the first where the sequent is provable, the second where it is not. For instance, the derivation on the left will should be terminate successfully, whereas the one on the right should not:

$$\frac{\frac{A \supset B, A \Rightarrow A \quad B, A \Rightarrow B}{A \supset B, A \Rightarrow A} \quad B, A \Rightarrow B}{\frac{A \supset B, A \Rightarrow B}{A \supset B \Rightarrow A \supset B}} \quad \frac{A \supset B \Rightarrow A \quad B \Rightarrow A}{\frac{A \supset B \Rightarrow A}{A \supset B \Rightarrow C} \quad B \Rightarrow C}$$

The problem on the left can be resolved easily enough; we simply add the axiom rules at the beginning of our list as well as the end, so we know that can never have an infinite branch stemming from $A \supset B, A \Rightarrow A$, since it will be pruned by an application of the rule **basic**. This still leaves us with the problem of non-derivable formulae being analysed *ad infinitum*, as would happen in the right hand case.

Closer inspection of what *Isabelle/Isar* outputs in this situation gives further insight into how it can be solved. The subgoals that *Isabelle* gives us are the current leaves in the deduction tree, ordered from left to right. So, from the tree above, we have the subgoals left to prove

1. $A \dashv\vdash B \mid - A$
2. $B \mid - A$
3. $B \mid - C$

Another application of the rule **impL** gives us the new subgoals

1. $A \dashv\vdash B \mid - A$

2. $B \vdash A$
3. $B \vdash A$
4. $B \vdash C$

and so on, so that n applications of the rule `impL` gives us n copies of the subgoal $B \vdash A$. All of these copied subgoals will be provable if and only if one of them is provable. Therefore, we can justifiably discard all of the copies and leave just a single subgoal. This is achieved using the tactic `distinct-subgoal-tac`. Repeated application of `impL` followed by `distinct-subgoal-tac` will simply yield the three subgoals

1. $A \rightarrow B \vdash A$
2. $B \vdash A$
3. $B \vdash C$

Now we can see that a loop is characterised by having the same set of subgoals before and after an application of our method from the previous example for **G4ip**. There is a tactical in *Isabelle* which will check for this condition, called `CHANGED`, which fails if and only if the subgoals before and after an application of a tactic are identical.

We must be careful about where we include this test. It must be implemented between loops, otherwise it will not have the desired effect. The tactic which works in this case can be included, as before, in the file with the rest of the rules, as

```
val G3ipRULE = DEPTH-SOLVE (CHANGED (FIRST[ rtac (thm 'impR') 1,
...
(rtac (thm 'disjR1') 1
APPEND rtac (thm 'disjR2') 1),
(rtac (thm 'impL') 1
THEN distinct-subgoal-tac)
]))
```

As was the case for **G4ip**, this no longer gives unprovable subgoals, it simply fails if the sequent is non-derivable.

References

- [1] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 1992.
- [2] L. Paulson. Isabelle's logics. Available at <http://www.cl.cam.ac.uk/research/hvg/Isabelle>, 2005.
- [3] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Number 43 in Cambridge Tracts in Computer Science. Cambridge University Press, second edition, 2000.
- [4] M. Wenzel. *Isabelle/Isar Reference Manual*, 2002.