---

layout: page title: “Calculi” category: doc date: 2015-04-17 09:08:29 —

The calculus toolbox has been designed around a special class of logics called *display calculi*, first introduced by Nuel Belnap as Display Logic. These logics are easily modified and extended, meaning that this toolbox, which is based around the aforementioned calculi can be very flexible.

The need for a toolbox for display calculi arose from the formalization of set calculi in the theorem prover Isabelle. The initial implementation of was a minimal calculus adapted from the D.EAK calculus described in the paper by A.Kurz, et al. This version of the calculus only contained a small subset of rules and connectives of the full calculus and was used to test out the implementation of such calculi in Isabelle.

Initially, a shallow embedding (abbreviated SE) of the calculus was formalized in Isabelle, however, with the aim of formalizing the cut elimination proof for the D.EAK calculus, it was soon discovered that the SE would not be sufficient and a deep embedding (abbreviated DE) of the calculus would be needed.

Basic Structure of a display calculus

The terms of the minimal display calculus are inductively built from atomic propositions, composed into formulas, which are in turn composed into structures. a valid term of a calculus is a sequent composed of two structures on either side of a turnstile: $X \vdash Y$, with the left structure \$X\$ referred to as the antecedent of the sequent and the right part \$Y\$ being the consequent/succedent. The definitions of the minimal calculus formula and structure terms are given below:

\$F:= ap \in \mathsf{AtProp} \mid   F \land F \mid F \rightarrow F\$ \$S:= F \mid \mathsf{Id} \mid S \,;

Shallow embedding

Both the shallow and the deep embedding follow a similar structure of formalization in Isabelle. Firstly, there is the inductive definition of the terms of the calculus, formalized via Isabelle’s `datatype` definition (formula terms defined in the previous section):

```
datatype formula = Atprop atProp
                 | And_F formula formula  (infix "∧F" 570)
                 | ImpR_F formula formula (infix "→F" 550)
```

Whereas in the abstract definition, the set of atomic propositions is not specifically defined, in the SE, we have chosen to use strings for atomic propositions:

```
type_synonym atProp = string
```

To keep things easily readable even in the Isabelle notation, syntactic sugar is introduced via the `infix` keyword, which allows infix syntax for the constructors of the Formula datatype.
Thus, a simple formula such as $p \land p$, written as `And_F (Atprop "p") (Atprop "p")` in Isabelle syntax without the syntactic sugar, can also be written as `Atprop "p" ∧F Atprop "p"` using the infix `∧F` constructor, instead of the `And_F` one.
Structures and Sequents of the calculus have been formalized in the SE in a similar fashion.

The next important part of the formalization is the notion of derivability of a sequent in a calculus. Isabelle provides facilities for inductively defining derivability through the `inductive` definition (note the following code snippet is not the full definition of derivable in the minimal calculus):

```
inductive derivable :: "sequent ⇒ bool" ("⊢d _" 300)
where
    Id: "⊢d Form (Atprop p) ⊢ Form (Atprop p)"
  | I_impR: " ⊢d (X ⊢ Y) ⟹ ⊢d (I ⊢ X>>Y)"
  | I_impR2: " ⊢d (I ⊢ X>>Y) ⟹ ⊢d (X ⊢ Y)"
  | ImpR_I: " ⊢d (X ⊢ Y) ⟹ ⊢d (Y>>X ⊢ I)"
  | ImpR_I2: " ⊢d (Y>>X ⊢ I) ⟹ ⊢d (X ⊢ Y)"
  | IW_L: " ⊢d (I ⊢ X) ⟹ ⊢d (Y ⊢ X)"
  | IW_R: " ⊢d (X ⊢ I) ⟹ ⊢d (X ⊢ Y)"
  ...
```

Deep embedding

Whilst the core SE implementation is quite short, it lacks features needed for the cut elimination proof, namely an explicit notion of proof trees.
Even though a derivable sequent always has a derivation tree, in SE, this tree is only implicitly encoded in Isabelle and cannot be accessed easily.
The inductive definition of the rules in the SE automatically generate rewrite rules of the same form, using Isabelle meta-variables along with the internal Isabelle pattern matching and rule application. Since these mechanisms are embedded deep within Isabelle's core, they are not easy to reason about and manipulate explicitly, as seen below in the proof tree for the derived weakening rule in the minimal calculus:

Note that since the proof tree is derived bottom-to-top, but we write proofs top-to-bottom in a text editor, the Isabelle proof is inverted:

```
lemma W_R:
```

```
assumes "⊢d X ⊢ Y"
shows "⊢d X ⊢ Y;Z"              (* goal: ⊢d X ⊢ Y;Z  *)
apply (rule ImpR_comma_disp2)   (* goal: ⊢d Y>>X ⊢ Z *)
apply (rule W_impR_L)           (* goal: ⊢d X ⊢ Y    *)
apply (rule assms)              (* done              *)
```

Since the notion of a proof tree in SE is very abstract as it is encoded using Isabelle's own proof environment, and the cut elimination proof for the display calculi needs an explicit way of manipulating proof trees, the DE version of the calculus introduces a different encoding that allows explicit reasoning with and manipulation of proof trees.
In order to encode proof trees, the notion of derivability had to be rewritten more explicitly.

A proof tree in the DE is a recursively defined term (using Isabelle's `datatype`), just like a sequent or a formula, containing a sequent and the accompanying rule that was applied to obtain it at its conclusion/root and sub-trees of previous steps/derivations forming its branches. All the leaves of the proof tree must contain valid axioms (in the minimal calculus, this means the *Id* rule).

The validity of such proof tree is determined by checking the derivation of the conclusion of every sub-tree from its branches. Since the data structure encoding these proof trees carries no notion of validity, a method *isProofTree* is introduced alongside the general proof tree definition. The *isProofTree* function recursively traverses a given tree and checks that the rules have been applied correctly at each level. A valid proof tree of a DE sequent is, in essence, an encoding of an Isabelle proof one would need to write in the SE to show derivability for the equivalent SE sequent.

The explicit application of rules to sequents in the *isProofTree* function introduces the need for explicit pattern matching of sequents and rules and that means, firstly, the introduction of free variables to the data structures of the SE.
In the DE, we introduce a new constructor for each type from the SE. For example, an atomic proposition in DE will now have two terms, the original one (*Atprop*) and an extra free variable term (*Atprop_Freevar*).

As well as introducing free variables to the formulas and structures, the DE of these terms has been made more parametric, separating the constructors for the n-ary connectives, to allow easier extension of these calculi (for example, the minimal calculus mainly differs from the full D.EAK calculus in the number of nullary, unary and binary connectives and rules ranging over them). The encoding of structure terms is now separated out into several definitions:

```
datatype Structure_ZerOp = Structure_Neutral
datatype Structure_BinOp = Structure_Comma
                         | Structure_ImpR

datatype Structure = Structure_Formula Formula
                   | Structure_Zer Structure_ZerOp
                   | Structure_Bin Structure Structure_BinOp Structure
                   | Structure_Freevar string
```

Note, that this encoding of the calculus also includes syntactic sugar notation of its terms to make them more concise and closer to the SE version, however the declaration of this notation has been omitted from the code snippet above for better readability.

Next, the notion of pattern-matching and derivation/rewriting of sequents through rule application needed to be encoded explicitly. The functions *match* and *replace*/*replaceAll* are matching functions that disassemble and reassemble a term of the calculus according to a template/rule.

The *match* function takes two terms; the first one is a concrete term and the second the rule containing free variables. The function then tries to match the free variables of the second term to sub-terms in the first expression, returning a list of the matched pairs. If the two terms cannot be matched against each other, i.e. have different structures, *match* will only return a list of partial matches which did not clash (or possibly even an empty list).

The *replaceAll* function is the converse of the *match* function, taking a list of matched pairs and a template/rule term and substituting the free variables in the rule with matched sub-terms from the list (The *replaceAll* function calls the *replace* function (which only replaces one pair in a given rule) repeatedly for the whole list).

For instance, the sequent \$?X \vdash ?Y\$, where \$?X\$ and \$?Y\$ represent free variables, can be matched against \$p \land p \vdash q\$, wherein \$?X\$ will be matched to \$p \land p\$ and \$?Y\$ will be matched to \$q\$. The actual list returned by applying match to these two sequents would look like this:

```
[(Sequent_Structure (?S"X"), Sequent_Structure ( (BF (Atprop "p")F ∧F (Atprop "p")F)S )), (
```

Notice the `Sequent_Structure` constructor that has no sugar syntax. This constructor is only used by *match* and *replace*/*replaceAll*. This constroctor was introduced to the DE due to the constraints of the class system in Isabelle, which does not allow polymorphism over functions with multiple type signatures and is in itself not a valid term of the calculus. However, the constructors `Structure_Formula` and `Formula_Atprop` which are part of the calculus take on a similar use in the *match* and *replace*/*replaceAll* functions. To get a better idea of what happens in these functions have a look at the core calculus template.

Finally, passing this list above to the *replaceAll* function and applying it to the rule $I \vdash ?X \rightarrow ?Y$ will replace the free variable $?X$ with $p \land p$ and $?Y$ with q, yielding $I \vdash (p \land p) \rightarrow q$.

The way the match function is defined allows for matching a rule ?X ?Y to a sequent ?Y ‘?Z. Even though the application of match to these two sequents would introduce a conflict in the free variable names (where the order of applying the replace back into a rule would potentially result in two different sequents), match is only meant to match terms with no free variables to rules. This is enforced by always checking that the first sequent contains no free variables before matching it to the rule. This condition is not enforced in the rule application itself, but is instead included in the definitions that use match as a guard before each application of match. Care must therefore be taken that this condition is met and enforced at the right times.

The der function in DE combines all the previous functions and the rule definitions and given a rule and a sequent, it will first check the rule is applicable and if so, will match the sequent with the rule and replace the matched substructures for the free variables of the rule conclusion (or conclusions for binary/ternary/etc. rules).