Formal proofs and proof languages

Enrico Tassi

23 June 2011 — Grenoble
Roadmap

Formal proofs
- Odd order theorem
- Finite sets
- Canonical structures
- Progress & Future directions

Proof languages
- Main characteristics of SSR
- Contextual rewrite patterns
- Pervasive views
- Future directions
Why that *Mathematical Components* project

The beginning of the story (as I know it)

- Gonthier verifies the four color theorem with Coq
- Mathematicians are not “impressed”
Why that *Mathematical Components* project

The beginning of the story (as I know it)
- Gonthier verifies the four color theorem with Coq
- Mathematicians are not “impressed”

Let’s try again

**question** “What would impress you?”
**answer** “The odd order theorem”
The classification of finite simple groups

Every finite groups is built using only finite simple groups

- **simple**: no normal subgroups (proper and non trivial)
- **normal**: $N \triangleleft G$ iff $gN = Ng$
- **quotient**: smaller objects, e.g. $G_1/N = G_2/N \rightarrow G_1 = G_2$

Like prime numbers are the building blocks of natural numbers

- **series**: $1 = N_1 \triangleleft \ldots \triangleleft N_n = G$ where $N_{i+1}/N_i$ is simple

Jordan-Hölder says that composition series are unique (up to permutation and isomorphisms between the factors).

Finite simple groups are of the following families:

- $Z_p$, $A_n$, Lie-type, 26 sporadic groups
The classification of finite simple groups

This was the result of a huge effort:

- tens of thousands pages in several hundred journal
- about 100 authors
- published mostly between 1955 and 2004
- revised proof began in 1983 (still in progress)
- in 2004 the last known gap was filled
- (complete) revised proof should be around five thousands pages
The classification of finite simple groups

This was the result of a huge effort:

- tens of thousands pages in several hundred journal
- about 100 authors
- published mostly between 1955 and 2004
- revised proof began in 1983 (still in progress)
- in 2004 the last known gap was filled
- (complete) revised proof should be around five thousands pages

That’s too much for a single research team...
The odd order theorem

Every finite group with odd order is solvable

solvable composition series’ factors are products of $Z_p$

Does the job for half of the cases!

- Simpler case proved by Suzuki in 1957 (17 pages)
- Proved by Feit and Thompson in 1963 (250 pages)
- Revised: Bender & Glauberman 1995, Peterfalvi 2000
Mathematical Components

Objectives
- Develop reusable libraries for Coq
- Develop a good proof language for Coq

Why the odd order theorem
- Challenging
- Requires to model complex mathematical reasoning
- Touches many areas of math
My contribution (to the main proof)

Contents

Preface ix

Chapter I. Preliminary Results 1
1. Elementary Properties of Solvable Groups 1
2. General Results on Representations 9
3. Actions of Frobenius Groups and Related Results 17
4. $p$-Groups of Small Rank 33
5. Narrow $p$-Groups 44
6. Additional Results 49

Chapter II. The Uniqueness Theorem 55
7. The Transitivity Theorem 55
8. The Fitting Subgroup of a Maximal Subgroup 61
9. The Uniqueness Theorem 64

Chapter III. Maximal Subgroups 69
10. The Subgroups $M_\alpha$ and $M_\sigma$ 69
11. Exceptional Maximal Subgroups 80
12. The Subgroup $E$ 83
13. Prime Action 97

Chapter IV. The Family of All Maximal Subgroups of $G$ 105
14. Maximal Subgroups of Type $\mathcal{P}$ and Counting Arguments 105
15. The Subgroup $M_F$ 117
Which objects need to be modelled

We now have (a) and (b). Repeating our last argument, we see that

\[ |R/T| = |R/C_R(W)| = p. \]

Clearly, \( T \text{ char } R \). This proves (c) and completes the proof of the lemma. \( \square \)

**Theorem 5.3.** Suppose \( p \) is an odd prime, \( R \) is a \( p \)-group, and \( r(R) \geq 3 \). Then \( R \) is narrow if and only if \( \mathcal{E}^2(R) \cap \mathcal{E}^*(R) \) is not empty (i.e., some elementary abelian subgroup of order \( p^2 \) in \( R \) is contained in no elementary abelian subgroup of order \( p^3 \) in \( R \)).

Suppose that \( R \) is narrow. Let \( T = C_R(\Omega_1(Z_2(R))) \). Then

(a) no element of \( \mathcal{E}^2(R) \cap \mathcal{E}^*(R) \) is contained in \( T \);
(b) \( |\Omega_1(Z(R))| = p \) and \( \Omega_1(Z_2(R)) \in \mathcal{E}^2(R) \),
(c) \( T \) is a characteristic subgroup of index \( p \) in \( R \), and
(d) if \( S \) is a subgroup of order \( p \) in \( R \) and \( r(C_R(S)) \leq 2 \), then \( C_T(S) \) is cyclic, \( S \cap R' = S \cap T = 1 \), and \( C_R(S) = S \times C_T(S) \).

**Proof.** Let \( Z = \Omega_1(Z(R)) \) and \( T = C_R(\Omega_1(Z_2(R))) \).

First assume that \( R \) is narrow. Take a subgroup \( R_0 \) of order \( p \) such that \( C_R(R_0) = R_0 \times R_1 \) for some cyclic group \( R_1 \). Since

\[ r(C_R(R_0)) \leq 2 < 3 \leq r(R), \]

\( R_0 \not\subseteq Z \), and so \( R_0 \cap Z = 1 \). Hence \( R_0 \subseteq R_0 \times Z \subseteq C_R(R_0) = R_0 \times R_1 \).

Thus \( R_1 \neq 1 \). Let
Finite (intensional) sets

We must find a good “encoding” for sets.

- Sets as characteristic functions
- In Coq functions are not extensional

\[(\forall x. f\ x = g\ x) \not\Rightarrow f = g\]

In a finite setting we can represent functions as their graphs, and finite sets as bitmasks

\[
\begin{array}{cccccccc}
1 & a & b & c & \ldots & c^1 & b^1 & a^1 \\
tt & ff & ff & tt & \ldots & ff & tt & ff
\end{array}
\]

Equal bitmasks, equal sets: \[(\forall x. b_1[x] = b_2[x]) \rightarrow b_1 = b_2\]
Function graphs, tuples, permutations, . . .

The construction is way more general.

Structure finType := {
    T : eqType; enum : list T;
    _ : forall x : T, count ((==) x) enum = 1
}

Functions from a finite domain $D$ to any type $T$ can be represented by their graphs:

Structure fgraphType (D : finType) T := {
    fval : list T;
    _ : length fval = length (enum D)
}
The construction is way more general.

Structure finType := {
    T : eqType; enum : list T;
    _ : forall x : T, count (((==) x) enum) = 1
}

Finite sets can be represented as functions to bool:

Structure finSet (D : finType) := {
    charf : fgraphType D bool
}
Function graphs, tuples, permutations, . . .

The construction is way more general.

Structure finType := {
    T : eqType; enum : list T;
    _ : forall x : T, count ((==) x) enum = 1
}

Homogeneous $n$-tuples over a type $T$ are just function graphs from $\mathcal{I}_n$ to $T$

Structure $\mathcal{I}_n$ := {
    m : nat ;
    _ : m < n
}.
Function graphs, tuples, permutations, . . .

The construction is way more general.

```coq
Structure finType := {
  T : eqType; enum : list T;
  _ : forall x : T, count ((==) x) enum = 1
}
```

**Permutations** are just \( n \)-tuples with no repetitions:

```coq
Structure perm (D : finType) := {
  perm : fgraphType D D; _ : uniq (fval perm)
}
```
Function graphs, tuples, permutations, ... 

The construction is way more general.

```
Structure finType := {
  T : eqType; enum : list T;
_ : forall x : T, count ((==) x) enum = 1
}

CIC functions can be easily turned into function graphs:

Definition fgraph_of_fun f :=
  mk_fgraphType (map f (enum D)) (map_len ...)
Function graphs, tuples, permutations, . . .

The construction is way more general.

**Structure** `finType := {`  
T : eqType; enum : list T;  
_ : forall x : T, count ((==) x) enum = 1  
}`

**Rotations** can be obtained easily (especially in modular arithmetic):

```
(rot shift f)(x)
```

```
(fun x => nth (x+shift % |D|))
```

```
(rot shift f)(x)
```
Structures and Canonical instances

Who said that we don’t use automation?

- CIC features dependent types
- terms (thus proofs) can be stored inside types
- type inference compares types using unification
- unification is user extensible

We use type inference to infer content, and we extend its capabilities using “Canonical Structures”.

- we infer proofs (a.k.a. automation)
- we infer operations (a.k.a. notation overloading)

<table>
<thead>
<tr>
<th>basic</th>
<th>groups</th>
<th>bigop</th>
<th>linalg</th>
<th>algebra</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>384</td>
<td>33</td>
<td>258</td>
<td>596</td>
<td>1447</td>
</tr>
</tbody>
</table>
Require Import List.
Structure predType T := mkPredType {
    pred_sort :> Type; topred : pred_sort -> T -> bool }.
Notation "a \in A" := (topred _ _ A a) (at level 70).

Structure eqType := mkEqType {
    eq_sort :> Type; eq_cmp : eq_sort -> eq_sort -> bool }.
Notation "a == b" := (eq_cmp _ a b) (at level 70).

Definition mem (T : eqType) l (x : T) :=
    match find (fun y => y == x) l
    with None => false | _ => true end.

Definition listPredType (T : eqType) :=
    @mkPredType T (list T) (@mem T).
Canonical Structure listPredType.

Definition natEqType := @mkEqType nat EqNat.beq_nat.
Canonical Structure natEqType.
Definition s := 1 :: 2 :: 3 :: nil.
Check (3 \in s).
Eval hnf in (3 \in s).

Variable n : nat.
Variable l : list nat.
Check (n \in l).

Definition listEqType (T : eqType) := @mkEqType (list T)
  (fun l1 l2 => length l1 == length l2 &&
   forallb (fun x => fst x == snd x) (combine l1 l2)).
Canonical Structure listEqType.

Check (l \in s :: l :: nil).
Canonical Structures — gory details

The user types

\((n \in l)\)
The user types

topen ?T ?p l n
Canonical Structures — gory details

The user types

\[
\text{forall } (T: \text{Type}) \ (p: \text{predType } T), \ \text{pred_sort } T \ p \rightarrow T \rightarrow \text{bool}
\]

: 

topred ?T ?p l n
The user types

```plaintext
forall (T:Type) (p:predType T), pred_sort T p -> T -> bool
: topred ?T ?p l n
```

Well typedness constraints

```plaintext
l : list nat = pred_sort ?T ?p
n : nat = ?T
```
Canonical Structures — gory details

The user types

```agda
def topred T p l n = ...
```

Well typedness constraints

```agda
l : list nat \equiv \text{pred\_sort}\ ?T \ ?p
n : nat \equiv \ ?T
```

The canonical instance

```agda
listPredType (E : eqType) :=
  \text{mkPredType}\ E \ (\text{list}\ E)\ (@\text{mem}\ E)
```
Canonical Structures — gory details

The user types

\[
\text{forall } (T : \text{Type}) \ (p : \text{predType } T), \ \text{pred_sort } T \ p \rightarrow T \rightarrow \text{bool} \\
\text{topred } ?T \ ?p \ l \ n
\]

Well typedness constraints

\[
l : \text{list } \text{nat} = \text{pred_sort } ?T \ ?p \\
n : \text{nat} = ?T
\]

The canonical instance

\[
\text{listPredType } (E : \text{eqType}) \\
\qquad := \\
\text{@mkPredType } (\text{eq_sort } E) \ (\text{list } (\text{eq_sort } E)) \ (\text{@mem } E)
\]
Canonical Structures — gory details

The user types

\[
\text{forall } (T : \text{Type}) \ (p : \text{predType } T), \ \text{pred\_sort } T \ p \rightarrow T \rightarrow \text{bool} \\
\text{topred } ?T \ ?p \ l \ n
\]

Well typedness constraints

\[
l : \text{list } \text{nat} = \text{pred\_sort } ?T \ ?p \\
n : \text{nat} = ?T
\]

The canonical instance

\[
\text{listPredType } (E : \text{eqType}) : \text{predType } (\text{eq\_sort } E) := \\
\quad \text{@mkPredType } (\text{eq\_sort } E) \ (\text{list } (\text{eq\_sort } E)) \ (\text{@mem } E)
\]
The user types

```plaintext
forall (T:Type) (p:predType T), pred_sort T p -> T -> bool :
topred ?T ?p l n
```

Well typedness constraints

```plaintext
l : list nat = pred_sort ?T ?p
n : nat = ?T
```

The canonical instance

```plaintext
listPredType (E : eqType) : predType (eq_sort E) :=
  @mkPredType (eq_sort E) (list (eq_sort E)) (@mem E)
```

Suggests

```plaintext
?p := listPredType ?E
?T := eq_sort ?E
```
Canonical Structures — gory details

The user types

```
forall (T:Type) (p:predType T), pred_sort T p -> T -> bool :
topred ?T ?p l n
```

Well typedness constraints

```
l : list nat = pred_sort ?T ?p
n : nat = ?T
```

The canonical instance

```
listPredType (E : eqType) : predType (eq_sort E) :=
  @mkPredType (eq_sort E) (list (eq_sort E)) (@mem E)
```

Suggests

```
?p := listPredType ?E
?T := eq_sort ?E

l : list nat = pred_sort ?T (listPredType ?E)
```
The user types

```
forall (T:Type) (p:predType T), pred_sort T p -> T -> bool :
topred ?T ?p l n
```

Well typedness constraints

```
l : list nat = pred_sort ?T ?p
n : nat = ?T
```

The canonical instance

```
listPredType (E : eqType) : predType (eq_sort E) :=
  @mkPredType (eq_sort E) (list (eq_sort E)) (@mem E)
```

Suggests

```
?p := listPredType ?E
?T := eq_sort ?E

l : list nat = list (eq_sort ?E)
```
The user types

```lean
forall (T:Type) (p:predType T), pred_sort T p -> T -> bool
```

```lean
topred ?T ?p l n
```

Well typedness constraints

```lean
l : list nat = pred_sort ?T ?p
n : nat = ?T
```

The canonical instance

```lean
listPredType (E : eqType) : predType (eq_sort E) :=
@mkPredType (eq_sort E) (list (eq_sort E)) (@mem E)
```

Suggests

```lean
?p := listPredType ?E
?T := eq_sort ?E
?E := natEqType
l : list nat = list (eq_sort ?E)
```
The user types

```
forall (T:Type) (p:predType T), pred_sort T p -> T -> bool

topred ?T ?p l n
```

Well typedness constraints

```
l : list nat = pred_sort ?T ?p
n : nat = ?T
```

The canonical instance

```
listPredType (E : eqType) : predType (eq_sort E) :=
  @mkPredType (eq_sort E) (list (eq_sort E)) (@mem E)
```

Suggests

```
?p := listPredType ?E
?T := eq_sort ?E
?E := natEqType
l : list nat = list (eq_sort natEqType)
```
The user types

```
forall (T:Type) (p:predType T), pred_sort T p -> T -> bool
```

`topred ?T ?p l n`

Well typedness constraints

```
l : list nat = pred_sort ?T ?p
n : nat = ?T
```

The canonical instance

```
listPredType (E : eqType) : predType (eq_sort E) :=
  @mkPredType (eq_sort E) (list (eq_sort E)) (@mem E)
```

Suggests

```
?p := listPredType ?E
?T := eq_sort ?E
?E := natEqType
l : list nat = list nat
```
Canonical Structures — gory details

The user types

(n \in l)

toppred (eq_sort natEqType) (listPredType natEqType) l n

Well typedness constraints

l : list nat = pred_sort ?T ?p
n : nat = ?T

The canonical instance

listPredType (E : eqType) : predType (eq_sort E) :=
    @mkPredType (eq_sort E) (list (eq_sort E)) (@mem E)

Suggests

?p := listPredType ?E
?T := eq_sort ?E
?E := natEqType
l : list nat = list nat
The prerequisites and the local analysis book are complete, the character theory part is ongoing. Estimation 1 more year of work.

<table>
<thead>
<tr>
<th>development</th>
<th>lines</th>
<th>bytes</th>
<th>gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math. Comp.</td>
<td>122.443</td>
<td>5.053.969</td>
<td>1.346.592</td>
</tr>
<tr>
<td>4 Colors</td>
<td>53.282</td>
<td>2.203.626</td>
<td>449.663</td>
</tr>
<tr>
<td>Prime Numbers</td>
<td>29.753</td>
<td>1.021.313</td>
<td>163.525</td>
</tr>
<tr>
<td>CoRN</td>
<td>140.540</td>
<td>3.858.981</td>
<td>744.711</td>
</tr>
</tbody>
</table>
Roadmap

Formal proofs
  Odd order theorem
  Finite sets
  Canonical structures
  Progress & Future directions

Proof languages
  Main characteristics of SSR
  Contextual rewrite patterns
  Pervasive views
  Future directions
Small scale reflection

Short history

v1.0  May 2006

Manual  February 2008

v1.1  November 2008, 4400 loc

v1.2  August 2009, 4600 loc

v1.3  March 2011, 5700 loc

v1.4  coming soon, ≈6300 loc

Objectives

▶ More compact and compositional than standard Coq’s vernacular
▶ Ease classical reasoning in the intuitionistic logics of Coq
▶ Robustness of scripts
Small scale reflection

Highlights

- **views** to link different incarnations of the same concept. In particular the computational and propositional aspect of a predicate.

- **rewrite** with surgical control as the main line of reasoning. In particular coimplication becomes equality on decidable predicates.
Why occurrence numbers are bad

... 

\( g := [\text{morphism of } \text{sdprodm defXA phiAiM}] : \{\text{morphism} \ \text{joing_group A X }\rightarrow gT} \)
\( \ker g : '\ker g = 'Mho^1(A) \)
\( \ker k : '\ker (\text{coset (}\ '\ker g)) \subset '\ker g \)
\( \ker nA : \text{joing_group A X } \subset 'N('\ker g) \)
\( \text{fact}_g := \text{factm skk nkA : coset_groupType ('ker g) }\rightarrow gT \)
\( \text{imnX} : X = \text{fact}_g \circ (X / '\ker g) \)
\( \text{nAA1} : A \subset 'N('Mho^1(A)) \)
\( \text{nXA1} : X \subset 'N('Mho^1(A)) \)

\[ \text{minnormal} (\text{fact}_g \circ (A / '\ker g)) X \rightarrow \text{minnormal} (A / '\ker g) (X / '\ker g) \]

\text{rewrite } \{1\}\text{imnX}
Why occurrence numbers are bad

... 

\[ g := \text{[morphism of sdprodm defXA phiAiM]} : \{\text{morphism}\} \text{joing_group A X} \rightarrow gT \] 
\[ \text{kerg} : \'\text{ker g} = \'\text{Mho}^1(A) \] 
\[ \text{skk} : \'\text{ker (coset ('ker g)) \subset 'ker g} \] 
\[ \text{nkA} : \text{joing_group A X \subset 'N('ker g)} \] 
\[ \text{fact_g := factm skk nkA : coset_groupType ('ker g)} \rightarrow gT \] 
\[ \text{imgX} : X = \text{fact_g @* (X / 'ker g)} \] 
\[ \text{nAA1} : A \subset '\text{N('Mho}^1(A)) \] 
\[ \text{nXA1} : X \subset '\text{N('Mho}^1(A)) \] 

\begin{align*}
\text{minnormal (fact_g @* (A / 'ker g)) X} & \rightarrow \\
\text{minnormal (A / 'ker g) (X / 'ker g)} & \\
\end{align*}

\text{rewrite} \{1\}\text{imgX} 

\text{rewrite} \{29\}\text{imgX}
Why occurrence numbers are bad
Why occurrence numbers are bad

Occurrence numbers are bad for the following reasons:

▶ can be hard to write
▶ scripts are less informative when they break

SSR 1.3 contextual patterns:

▶ specify the occurrences looking at their context
▶ rewrite \([R \text{ in minnormal } \_ R] \text{img} X\)
  to rewrite exactly
  \[
  \text{minnormal} \left(\text{fact}_g @* \left(\frac{A}{'\ker g}\right)\right) X \rightarrow
  \text{minnormal} \left(\frac{A}{'\ker g}\right) \left(\frac{X}{'\ker g}\right)
  \]

Rewrite (contextual) patterns

Terminology

**matching** head constant driven

\[ \text{addnC} : \forall \ a \ b, \ a + b = b + a \]

**redex** the term being rewritten, identified with a given pattern or a pattern inferred looking at the rule

Rewrite pattern syntax

- `rewrite rule`
- `rewrite [t]rule`
- `rewrite [in t]rule`
- `rewrite [X in t]rule`
- `rewrite [in X in t]rule`
- `rewrite [e in X in t]rule`
- `rewrite [e as X in t]rule`
Rewrite patterns — example 1

The rule

\[ \text{addnC} : \_ + \_ = \_ + \_ \]

The tactic invocation

\texttt{rewrite addnC.}

The goal

\[(x + y) + f x (x + y).+1 = 0\]
Rewrite patterns — example 2

The rule

\[(\text{addnC } x.+1) : x.+1 + _ = _ + x.+1\]

The tactic invocation

\texttt{rewrite \[_.+1\]}(\texttt{addnC } x.+1).

The goal

\[(x + y) + f x (x + y).+1 = 0\]

Because \((x + y).+1 = x.+1 + _\)
Contextual rewrite patterns — example 3

The rule
\[ \text{addnC} : _ + _ = _ + _ \]

The tactic invocation
\[ \text{rewrite} \ [\text{in f} \ _ \_]\text{addnC}. \]

The goal
\[ (x + y) + f x (x + y) + 1 = 0 \]
Contextual rewrite patterns — example 4

The rule

\[(\text{addnC } x.+1) : x.+1 + _ = _ + x.+1\]

The tactic invocation

\texttt{rewrite [R in f \_ R](addnC x.+1).}

The goal

\[(x + y) + f (x.+1 + y) (x + y).+1 = 0\]

Because \(R\) captured \((x + y).+1 = x.+1 + _\)
Contextual rewrite patterns — example 5

The rule

\((\text{addnC} \ x) : x + _ = _ + x\)

The tactic invocation

\(\text{rewrite } [\text{in } R \text{ in } f \ _ \ R](\text{addnC} \ x).\)

The goal

\((x + y) + f \ x (z + (x + y).+1) = 0\)

Because R captured \(z + (x + y).+1\)
Contextual rewrite patterns — example 6

The rule

\[\text{(addnC } x.+1) : x.+1 + _ = _ + x.+1\]

The tactic invocation

\[\text{rewrite } [_.+1 \text{ in } R \text{ in } f \_ \text{ R}](\text{addnC } x.+1).\]

The goal

\[\(x + y\) + f x (z + (x + y).+1) = 0\]

Because \(R\) captured \(z + (x + y).+1\) and \(_.+1\) matched \((x + y).+1 = x.+1 + _\)
Contextual rewrite patterns — example 7

The rule
addnC : _ + _ = _ + _

The tactic invocation
rewrite [x.+1 + y as R in f _ (_ + R)]addnC.

The goal
(x + y) + f x (z + (x + y).+1) = 0

Because R captured (x + y).+1 = x.+1 + y = _ + _
Views everywhere

In standard Coq, one would begin this proof this way:

\textbf{Lemma} foo : \texttt{forall} x y, P x \LAND Q y \imp R x \imp G.
\texttt{intros} x y \[Px \ Qy\] \texttt{Rx}.

With \texttt{ssreflect} you use a boolean conjunction, thus you need a view to perform the case analysis.

\textbf{Lemma} foo : \texttt{forall} x y, P x \LAND Q y \imp R x \imp G.
\texttt{move=> x y; move/andP=> \[Px \ Qy\] \texttt{Rx}.}
\texttt{move=> x y; case/andP=> Px Qy Rx.}

Views were tactic flags, now they can be placed everywhere.

\textbf{Lemma} foo : \texttt{forall} x y, P x \LAND Q y \imp R x \imp G.
\texttt{move=> x y /andP[Px Qy] Rx.}

And this allows interesting (ab)uses:

\texttt{have/(nilpotent_pcoreC p)/dprodP[\_ \<-> \_ \_]: nilpotent F := Fitting-nil \_}
Future of ssreflect — v1.4

Patterns everywhere (idea of a user)

\[
\begin{align*}
\text{set } t &:= \{3\}(a + \_).
\text{set } t &:= (a + \_ \text{ in } R \text{ in } \_ = R).
\end{align*}
\]

User defined notations as patterns

\textbf{Notation} RHS := (X \text{ in } \_ = X).

\[
\begin{align*}
\text{set } t &:= (a + \_ \text{ in } \text{RHS}).
\text{rewrite } [\text{in RHS}]\text{addnC}.
\text{elim: (n in RHS)}.
\end{align*}
\]

Library of v1.3 completely ported to the new features of v1.3 plus some minor additions

Sould be ready for the ITP conference (end August)
Thanks

Thanks for your attention!