# Formal proofs and proof languages

Enrico Tassi

CENTRE DE RECHERCHE COMMUN
INRIA MICROSOFT RESEARCH

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: $N \lhd G$ iff $gN = Ng$

quotient: smaller objects, e.g. $G_1/N = G_2/N \to G_1 = G_2$

Like prime numbers are the building blocks of natural numbers

series: $1 = N_1 \lhd \ldots \lhd N_n = G$ 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...

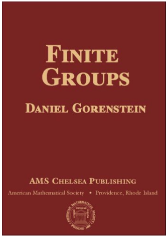
# The odd order theorem

Every finite group with odd order is solvable

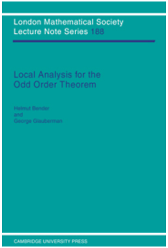
solvable composition series' factors are products of $Z_p$

Does the job for half of the cases!

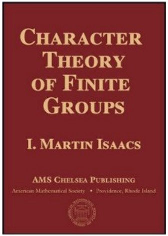
- Simpler case proved by Suzuki in 1957 (17 pages)
- Proved by Feit and Thompson in 1963 (250 pages)
- Revised: Bender & Glauberman 1995, Peterfalvi 2000



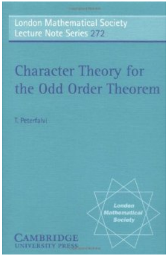

300(of 500)




170




?(of 300)




100(of 150)

# 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 $\mathscr{P}$ 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, $T$ char $R$. This proves (c) and completes the proof of the lemma. □

**Theorem 5.3.** Suppose $p$ is an odd prime, $R$ is a $p$-group, and $\mathrm{r}(R) \geq 3$. 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 $\mathrm{r}(C_R(S)) \leq 2$, 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

$$\mathrm{r}(C_R(R_0)) \leq 2 < 3 \leq \mathrm{r}(R),$$

$R_0 \not\subseteq Z$, and so $R_0 \cap Z = 1$. Hence $R_0 \subset R_0 \times Z \subseteq C_R(R_0) = R_0 \times R_1$. 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
  $(\forall x.f\ x = g\ x) \not\to f = g$

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | a | b | c | ⋯ | $c^{-1}$ | $b^{-1}$ | $a^{-1}$ |
| tt | ff | ff | tt | ⋯ | ff | tt | ff |

Equal bitmasks, equal sets: $(\forall x.b_1[x] = b_2[x]) \to b_1 = b_2$

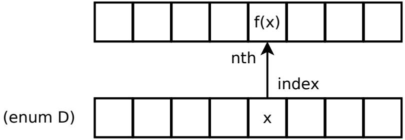
# Function graphs, tuples, permutations, ...

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)
}
```



# Function graphs, tuples, permutations, ...

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 'I_n to $T$

```
Structure 'I_ n := {
  m : nat ;
  _ : m < n
}.
```

# Function graphs, tuples, permutations, ...

The construction is way more general.

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

Permutations are just *n*-tuples with no repetitions:

```
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:

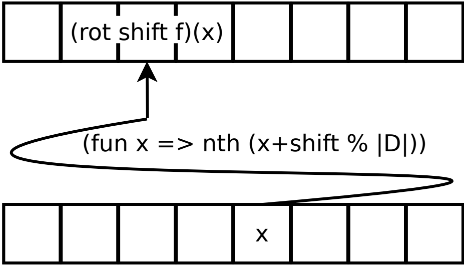
```
Definition fgraph_of_fun f :=
  mk_fgraphType (map f (enum D)) (map_len ...)
```

# 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 (especialaly in modular arithmetic):



# 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)

| basic | groups | bigop | linalg | algebra | total |
|---|---|---|---|---|---|
| 176 | 384 | 33 | 258 | 596 | 1447 |

# Canonical Structures — example 1

```
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.
```

# Canonical Structures — example 2

```
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)
```

# Canonical Structures — gory details

The user types

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

# Canonical Structures — gory details

The user types

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

# 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
```

# 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)                              :=
  @mkPredType          E  (list          E ) (@mem 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)                              :=
  @mkPredType (eq_sort E) (list (eq_sort E)) (@mem 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)
```

# 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
```

# 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)
```

# 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  =  list (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
?E := natEqType
l : list nat  =  list (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
?E := natEqType
l : list nat  =  list (eq_sort natEqType)
```

# 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
?E := natEqType
l : list nat  =  list nat
```

# Canonical Structures — gory details

## The user types

```
(n \in l)

topred (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
```

# Progress & future

The prerequisites and the local analysis book are complete, the character theory part is ongoing. Estimation 1 more year of work.

| development | lines | bytes | gzip |
|---|---:|---:|---:|
| Math. Comp. | 122.443 | 5.053.969 | 1.346.592 |
| 4 Colors | 53.282 | 2.203.626 | 449.663 |
| Prime Numbers | 29.753 | 1.021.313 | 163.525 |
| CoRN | 140.540 | 3.858.981 | 744.711 |

# 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, $\approx$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 becames equality on decidable predicates.

# Why occurrence numbers are bad

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

```
rewrite {1}imgX
```

# Why occurrence numbers are bad

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

```
rewrite {1}imgX
```

```
rewrite {29}imgX
```

# Why occurrence numbers are bad

```
is_true (@minnormal gT (@morphim (@coset_groupType gT (@ker gT gT (@gval gT (@joing_group gT
(@gval gT A) (@gval gT X))) g (@MorPhantom gT gT (@mfun gT gT (@gval gT (@joing_group gT
(@gval gT A) (@gval gT X))) g)))) gT (@morphim gT (@coset_groupType gT (@ker gT gT (@gval gT
(@joing_group gT (@gval gT A) (@gval gT X))) g (@MorPhantom gT gT (@mfun gT gT (@gval gT
(@joing_group gT (@gval gT A) (@gval gT X))) g)))) (@gval gT (@normaliser_group gT (@ker gT gT
(@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g (@MorPhantom gT gT (@mfun gT gT
(@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g))))) (@coset_morphism gT (@ker gT gT
(@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g (@MorPhantom gT gT (@mfun gT gT
(@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g)))) (@MorPhantom gT (@coset_groupType
gT (@ker gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g (@MorPhantom gT gT
(@mfun gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g)))) (@mfun gT
(@coset_groupType gT (@ker gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g
(@MorPhantom gT gT (@mfun gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g))))
(@gval gT (@normaliser_group gT (@ker gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT
X))) g (@MorPhantom gT gT (@mfun gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT
X))) g))))) (@coset_morphism gT (@ker gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT
X))) g (@MorPhantom gT gT (@mfun gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT
X))) g)))))) (@gval gT (@joing_group gT (@gval gT A) (@gval gT X)))) (@factm_morphism gT
(@coset_groupType gT (@ker gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g
(@MorPhantom gT gT (@mfun gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g))))
gT (@joing_group gT (@gval gT A) (@gval gT X)) (@normaliser_group gT (@ker gT gT (@gval gT
(@joing_group gT (@gval gT A) (@gval gT X))) g (@MorPhantom gT gT (@mfun gT gT (@gval gT
(@joing_group gT (@gval gT A) (@gval gT X))) g)))) g (@coset_morphism gT (@ker gT gT (@gval gT
(@joing_group gT (@gval gT A) (@gval gT X))) g (@MorPhantom gT gT (@mfun gT gT (@gval gT
(@joing_group gT (@gval gT A) (@gval gT X))) g)))) skk nkA) (@MorPhantom (@coset_groupType gT
(@ker gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g (@MorPhantom gT gT (@mfun
gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g)))) gT fact_g) (@quotient gT
(@gval gT A) (@ker gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g (@MorPhantom
gT gT (@mfun gT gT (@gval gT (@joing_group gT (@gval gT A) (@gval gT X))) g))))) (@gval gT
X)) ->
is_true (@minnormal (@coset_groupType gT (@ker gT gT (@gval gT (@joing_group gT (@gval gT A)
(@gval gT X))) g (@MorPhantom gT gT (@mfun gT gT (@gval gT (@joing_group gT (@gval gT A)
(@gval gT X))) g)))) (@quotient gT (@gval gT A) (@ker gT gT (@gval gT (@joing_group gT (@gval
gT A) (@gval gT X))) g (@MorPhantom gT gT (@mfun gT gT (@gval gT (@joing_group gT (@gval gT A)
(@gval gT X))) g)))) (@quotient gT (@gval gT X) (@ker gT gT (@gval gT (@joing_group gT
```

# Why occurrence numbers are bad

Occurrence number 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 in minnormal _ R]imgX`
  to rewrite exactly

```
minnormal (fact_g @* (A / 'ker g)) X ->
  minnormal (A / 'ker g) (X / 'ker g)
```

# Rewrite (contextual) patterns

Terminology

matching head constant driven

```
addnC : forall a b, a + b = b + a
```

redex the term being rewritten, identified with a given patter 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

```
addnC : _ + _ = _ + _
```

The tactic invocation

```
rewrite addnC.
```

The goal

```
(x + y) + f x (x + y).+1 = 0
```

# Rewrite patterns — example 2

The rule

```
(addnC x.+1) : x.+1 + _ = _ + x.+1
```

The tactic invocation

```
rewrite [_.+1](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

```
addnC : _ + _ = _ + _
```

The tactic invocation

```
rewrite [in f _ _]addnC.
```

The goal

```
(x + y) + f x (x + y).+1 = 0
```

# Contextual rewrite patterns — example 4

The rule

```
(addnC x.+1) : x.+1 + _ = _ + x.+1
```

The tactic invocation

```
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

```
(addnC x) : x + _ = _ + x
```

The tactic invocation

```
rewrite [in R in f _ R](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

```
(addnC x.+1) : x.+1 + _ = _ + x.+1
```

The tactic invocation

```
rewrite [_.+1 in R in f _ R](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:

```
Lemma foo : forall x y, P x /\ Q y -> R x -> G.
intros x y [Px Qy] Rx.
```

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

```
Lemma foo : forall x y, P x && Q y -> R x -> G.
move=> x y; move/andP=> [Px Qy] Rx.
move=> x y; case/andP=> Px Qy Rx.
```

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

```
Lemma foo : forall x y, P x && Q y -> R x -> G.
move=> x y /andP[Px Qy] Rx.
```

And this allows interesting (ab)uses:

```
have/(nilpotent_pcoreC p)/dprodP[_ <- _ _]: nilpotent F := Fitting_nil _
```

# Future of ssreflect — v1.4

Patterns everywhere (idea of a user)

```
set t := {3}(a + _).
set t := (a + _ in R in _ = R).
```

User defined notations as patterns

```
Notation RHS := (X in _ = X).

set t := (a + _ in RHS).
rewrite [in RHS]addnC.
elim: (n in RHS).
```

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!