INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Small Scale Reflection Extension for the Coq *system*

Georges Gonthier — Assia Mahboubi — Enrico Tassi

N° 6455

July 2009

Thème SYM

Rapport de recherche

MICROSOFT RESEARCH INRIA JOINT CENTRE

ISSN 0249-6399 ISRN INRIA/RR--6455--FR+ENG

A Small Scale Reflection Extension for the Coq system

Georges Gonthier[*] , Assia Mahboubi[†] , Enrico Tassi[‡]

Thème SYM — Systèmes symboliques
Projets Composants Mathématiques, Centre Commun INRIA Microsoft Research

Rapport de recherche n° 6455 — July 2009 — 83 pages

Abstract: This document describes a set of extensions to the proof scripting language of the Coq proof assistant. While these extensions were developed to support a particular proof methodology - small-scale reflection - most of them actually are of a quite general nature, improving the functionality of Coq in basic areas such as script layout and structuring, proof context management, and rewriting. Consequently, and in spite of the title of this document, most of the extensions described here should be of interest for all Coq users, whether they embrace small-scale reflection or not.

Key-words: proof assistants, formal proofs, Coq, small scale reflection, tactics

[*] Microsoft Research, Cambridge, R-U, Centre commun INRIA Microsoft Research
[†] Équipe-projet TypiCal, INRIA Futurs, Centre commun INRIA Microsoft Research
[‡] Équipe-projet Mathematical Components, INRIA Futurs, Centre commun INRIA Microsoft Research

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 60 19 66 08

A Small Scale Reflection Extension for the Coq system

Résumé : Ce rapport présente une extension de l'assistant à la preuve Coq. Cette extension a été conçue pour améliorer le support d'une méthodologie de preuve formelle, appelée réflexion à petite échelle. Néanmoins, la majeure partie de ses apports sont des améliorations d'ordre général des fonctionnalités du système Coq comme la structuration des scripts, la gestion des contextes de preuve, et la réécriture. C'est pourquoi, en dépit du titre de ce document, la plupart des fonctionnalités décrites ici sont susceptibles d'intéresser tout utilisateur de Coq, utilisant ou non les techniques de réflexion à petite échelle.

Mots-clés : assistants à la preuve, preuve formelle, Coq, réflexion à petite échelle, tactiques

Contents

1 Introduction

Small-scale reflection is a formal proof methodology based on the pervasive use of computation with symbolic representations. Symbolic representations are usually hidden in traditional computational reflection (e.g., as used in the Coq[1] `ring`, or `romega`): they are generated on-the-fly by some heuristic algorithm and directly fed to some decision or simplification procedure whose output is translated back to "logical" form before being displayed to the user. By contrast, in small-scale reflection symbolic representations are ubiquitous; the statements of many top-level lemmas, and of most proof subgoals, explicitly contain symbolic representations; translation between logical and symbolic representations is performed under the explicit, fine-grained control of the proof script.

The efficiency of small-scale reflection hinges on the fact that fixing a particular symbolic representation strongly directs the behaviour of a theorem-prover:

- Logical case analysis is done by enumerating the symbols according to their inductive type: the representation describes which cases should be considered.
- Many logical functions and predicates are represented by concrete functions on the symbolic representation, which can be computed once (part of) the symbolic representation of objects is known: the representation describes what should be done in each case.

Thus by controlling the representation we also control the automated behaviour of the theorem prover, which can be quite complex, for example if a predicate is represented by a sophisticated decision procedure. The real strength of small-scale reflection, however, is that even very simple representations provide useful procedures. For example, the truth-table representation of connectives, evaluated left-to-right on the Boolean representation of propositions, provides sufficient automation for most propositional reasoning.

Small-scale reflection defines a basis for dividing the proof workload between the user and the prover: the prover engine provides computation and database functions (via partial evaluation, and definition and type lookup, respectively), and the user script guides the execution of these functions, step by step. User scripts comprise three kinds of steps:

- Deduction steps directly specify part of the construction of the proof, either top down (so-called forward steps), or bottom-up (backward steps). A reflection step that switches between logical and symbolic representation is just a special kind of deductive step.
- Bookkeeping steps manage the proof context, introducing, renaming, discharging, or splitting constants and assumptions. Case-splitting on symbolic representations is an efficient way to drive the prover engine, because most of the data required for the splitting can be retrieved from the representation type, and because specialising a single representation often triggers the evaluation of several representation functions.

[1] http://coq.inria.fr

- Rewriting steps use equations to locally change parts of the goal or assumptions. Rewriting is often used to complement partial evaluation, bypassing unknown parameters (e.g., simplifying `b && false` to `false`). Obviously, it's also used to implement equational reasoning at the logical level, for instance, switching to a different representation.

It is a characteristic of the small-scale reflection style that the three kinds of steps are roughly equinumerous, and interleaved; there are strong reasons for this, chief among them the fact that goals and contexts tend to grow rapidly through the partial evaluation of representations. This makes it impractical to embed most intermediate goals in the proof script - the so-called declarative style of proof, which hinges on the exclusive use of forward steps. This also means that subterm selection, especially in rewriting, is often an issue.

The basic Coq tactic language is not well adapted to small-scale reflection proofs. It is heavily biased towards backward steps, with little support for forward steps, or even script layout (these are deferred to the "vernacular", i.e., `Section`/`Module` layer of the input language). The support for rewriting is primitive, requiring a separate tactic for each kind of basic step, and the behaviour of subterm selection is undocumented. Many of the basic tactics, such as `intros`, `induction` and `inversion`, implement fragile context manipulation heuristics which hinder precise bookkeeping; on the other hand the under-utilised "intro patterns" provide excellent support for case splitting.

The extensions presented here were designed to improve the functionality of Coq in all those areas, providing:

- support for better script layout
- better support for forward steps
- common support for bookkeeping in all tactics
- common support for subterm selection in all tactics
- a unified interface for rewriting, definition expansion, and partial evaluation
- improved robustness with respect to evaluation and conversion
- support for reflection steps.

We should point out that only the last functionality is specific to small-scale reflection. All the others are of general use. Moreover most of these features are introduced not by adding new tactics, but by extending the functionality of existing ones: indeed we introduce only three new tactics, rename three others, but all subsume more than a dozen of the basic Coq tactics.

How to read this documentation

The syntax of the tactics is presented as follows:

- `terminals` are in typewriter font and ⟨*non terminals*⟩ are between angle brackets.
- Optional parts of the grammar are surrounded by [ ] brackets. These should not be confused with verbatim brackets `[ ]`, which are delimiters in the SSREFLECT syntax.
- A vertical rule | indicates an alternative in the syntax, and should not be confused with a verbatim vertical rule between verbatim brackets `[ | ]`.
- A non empty list of non terminals (at least one item should be present) is represented by ⟨*non terminals*⟩$^+$. A possibly empty one is represented by ⟨*non terminals*⟩*.
- In a list of non terminals, items are separated by blanks.

We follow the default color scheme of the SSREFLECT mode for ProofGeneral provided in the distribution:

`tactic` or `Command` or `keyword` or `tactical`

Closing tactics/tacticals like `exact` or `by` (see section 6.2) are in red.

2 Distribution

2.1 Files

The implementation of the small-scale reflection extension comprises a Caml extension module (`ssreflect.ml`), which provides the tactic language, and several Coq vernacular files:

- `ssreflect`: technical results for SSReflect tactics
- `ssrfun`: functions, functional equality, bijectivity, injectivity, surjectivity,...
- `ssrbool`: extended toolkit for reflection
- `eqtype`: structure for type with decidable intensionalequality
- `choice`: structures for types with a choice operator and for countable types
- `ssrnat`: natural numbers, arithmetic
- `div`: divisibility of natural numbers; GCDs and extended GCDs
- `prime`: primes, and prime decomposition of natural numbers
- `binomial`: factorials, binomials
- `seq`: lists
- `path`: non-empty seuqneces over an eqtype that obey a progression relation
- `fintype`: structure for finite types (types with finitely many elements)
- `fingraph`: finite graphs (as relations on finite types)
- `tuple`: lists with a fixed (known) length
- `finset`: finite sets (over finite types)
- `finfun`: functions with a finite domain
- `bigop`: finitely iterated operators
- `ssralg`: core algebraic hierarchy
- `finalg`: algebraic hierarchy for finite types
- `poly`: basic theory of polynomials
- `fingroup`: elementary finite group theory
- `perm`: finite permutation groups

- `morphism`: finite group morphisms
- `presentation`: generator and relation group presentations
- `automorphism`: finite group automorphisms
- `quotient`: cosets, quotients, isomorphism theorems
- `action`: finite group actions, orbits, stabilizers
- `gproduct`: direct, semidirect and central group products
- `gfunctor`: (characteristic) group functor hierarchy
- `zmodp`: properties of $\mathbf{Z}/n\mathbf{Z}$
- `cyclic`: properties of cyclic groups
- `center`: properties of group centers
- `commutator`: properties of commutator subgroups
- `gseries`: normal, central, chief group series
- `jordanholder`: sections, factors and the Jordan-Holder theorem
- `nilpotent`: nilpotent and solvable groups, lower and upper series
- `pgroup`: π-groups, π-cores, Hall and Sylow subgroups
- `sylow`: the Sylow theorems and its consequences; the Baer-Suzuki theorem
- `primitive_action`: primitive and n-transitive actions
- `alt`: symmetric and alternating groups
- `abelian`: structure of abelian groups; homocyclic and elementary abelian groups; group p-rank and rank
- `finmodule`: finite modules, transfer, and the Gaschütz theorems
- `maximal`: Frattini, Fitting, special, extraspecial, and critical subgroups
- `hall`: the Schur-Zassenhaus and Hall theorems; coprime action
- `extremal`: classification of extremal p-groups (modular or (generalized) dihedral/quaternion)
- `extraspecial`: classification of extraspecial groups
- `frobenius`: Frobenius groups, semiregular and semiprime action

- `matrix`: elementary matrix algebra; determinants
- `mxalgebra`: matrix rank, row spaces, and subalgebras
- `vector`: finite dimensional abstract linear algebra
- `mxpoly`: minimal and characteristic polynomials, resultants
- `mxrepresentation`: (modular) finite group representation theory
- `mxabelem`: representation induced by normal elementary abelian subgroups

Note that some files from the SSReflect 1.2 distribution have been renamed (trailing `s` are dropped, and `connect` becomes `fingraph`).

2.2 Compatibility issues

Every effort has been made to make the small-scale reflection extensions upward compatible with the basic Coq, but a few discrepancies were unavoidable:

- New keywords (`is`) might clash with variable, constant, tactic or tactical names, or with quasi-keywords in tactic or vernacular notations.
- New tactic(al)s names (`last`, `done`, `have`, `suffices`, `without loss`, `congr`, `unlock`) might clash with user tactic names.
- Identifiers with both leading and trailing `_`, such as `_x_`, are reserved by SSReflect and cannot appear in scripts.
- The extensions to the `rewrite` tactic are partly incompatible with those now available in current versions of Coq; in particular: `rewrite .. in (type of k)` or `rewrite .. in *` or any other variant of `rewrite` will not work, and the SSReflect syntax and semantics for occurrence selection and rule chaining is different. Use an explicit rewrite direction (`rewrite <-` ... or `rewrite ->` ...) to access the Coq `rewrite` tactic.
- New symbols (`//, /=, //=`) might clash with adjacent symbols (e.g., '`//`') instead of '/''/'). This can be avoided by inserting white spaces.
- New constant and theorem names might clash with the user theory. This can be avoided by not importing all of SSReflect:

```
Require ssreflect.
Import ssreflect.SsrSyntax.
```

Note that SSReflect extended rewrite syntax and reserved identifiers are enabled only if the `ssreflect` module has been required and if `SsrSyntax` has been imported. Thus a file that requires (without importing) `ssreflect` and imports `SsrSyntax`, can be required and imported without automatically ebnabling SSReflect's extended rewrite syntax and reserved identifiers.

- Some user notations (in particular, defining an infix ';') might interfere with "open term" syntax of tactics such as `have`, `set` and `pose`.

- The generalisation of `if` statements to non-Boolean conditions is turned off by SSReflect, because it is mostly subsumed by `Coercion` to `bool` of the `sumXXX` types (declared in `ssrfun.v`) and the `if ... is` construct (see 3.2). To use the generalised form, turn off the SSReflect Boolean `if` notation using the command:

```
Close Scope boolean_if_scope.
```

- The following two options can be unset to disable the incompatibile `rewrite` syntax and allow reserved identifiers to appear in scripts.

```
Unset SsrRewrite.
Unset SsrIdents.
```

3 Gallina extensions

Small-scale reflection makes an extensive use of the programming subset of Gallina, Coq's logical specification language. This subset is quite suited to the description of functions on representations, because it closely follows the well-established design of the ML programming language.The SSReflect extension provides three additions to Gallina, for pattern assignment, pattern testing, and polymorphism; these mitigate minor but annoying discrepancies between Gallina and ML.

3.1 Pattern assignment

The SSReflect extension provides the following construct for irrefutable pattern matching, that is, destructuring assignment:

`let:` $\langle pattern \rangle$ `:=` $\langle term \rangle_1$ `in` $\langle term \rangle_2$

Note the colon ':' after the `let` keyword, which avoids any ambiguity with a function definition or Coq's basic destructuring `let`. The `let:` construct differs from the latter in that

- The pattern can be nested (deep pattern matching), in particular, this allows expression of the form:

```
let: exist (x, y) p_xy := Hp in ...
```

- The destructured constructor is explicitly given in the pattern, and is used for type inference, e.g.,

  ```
  Let f u := let: (m, n) := u in m + n.
  ```

  using a colon `let:`, infers `f : nat * nat -> nat`, whereas

  ```
  Let f u := let (m, n) := u in m + n.
  ```

  with a standard `let`, requires an extra type annotation.

The `let:` construct is just (more legible) notation for the primitive Gallina expression

`match` ⟨*term*⟩$_1$ `with` ⟨*pattern*⟩ `=>` ⟨*term*⟩$_2$ `end`

Due to limitations of the Coq v8 display API, a `let:` expression will always be displayed with the Coq v8.2 `let 'C ...` syntax, which does not handle user notation and clashes with the lexical conventions of the SSReflect library.

The SSReflect destructuring assignment supports all the dependent match annotations; the full syntax is

`let:`⟨*pattern*⟩$_1$ `as` ⟨*ident*⟩ `in` ⟨*pattern*⟩$_2$ `:=` ⟨*term*⟩$_1$ `return` ⟨*term*⟩$_2$ `in` ⟨*term*⟩$_3$

where ⟨*pattern*⟩$_2$ is a *type* pattern and ⟨*term*⟩$_1$ and ⟨*term*⟩$_2$ are types.

When the `as` and `return` are both present, then ⟨*ident*⟩ is bound in both the type ⟨*term*⟩$_2$ and the expression ⟨*term*⟩$_3$; variables in the optional type pattern ⟨*pattern*⟩$_2$ are bound only in the type ⟨*term*⟩$_2$, and other variables in ⟨*pattern*⟩$_1$ are bound only in the expression ⟨*term*⟩$_3$, however.

3.2 Pattern conditional

The following construct can be used for a refutable pattern matching, that is, pattern testing:

`if` ⟨*term*⟩$_1$ `is` ⟨*pattern*⟩$_1$ `then` ⟨*term*⟩$_2$ `else` ⟨*term*⟩$_3$

Although this construct is not strictly ML (it does exits in variants such as the pattern calculus or the ρ-calculus), it turns out to be very convenient for writing functions on representations, because most such functions manipulate simple datatypes such as Peano integers, options, lists, or binary trees, and the pattern conditional above is almost always the right construct for analysing such simple types. For example, the `null` and `all` list function(al)s can be defined as follows:

```
Variable d: Set.
Fixpoint null (s : list d) := if s is nil then true else false.
Variable a : d -> bool.
Fixpoint all (s : list d) : bool :=
  if s is cons x s' then a x && all s' else true.
```

The pattern conditional also provides a notation for destructuring assignment with a refutable pattern, adapted to the pure functional setting of Gallina, which lacks a `Match_Failure` exception.

Like `let:` above, the `if`...`is` construct is just (more legible) notation for the primitive Gallina expression:

```
match ⟨term⟩1 with ⟨pattern⟩ => ⟨term⟩2 | _ => ⟨term⟩3 end
```

Similarly, it will always be displayed as the expansion of this form in terms of primitive `match` expressions (where the default expression $\langle term \rangle_3$ may be replicated).

Explicit pattern testing also largely subsumes the generalisation of the `if` construct to all binary datatypes; compare:

```
if ⟨term⟩ is inl _ then ⟨term⟩l else ⟨term⟩r
```

and:

```
if ⟨term⟩ then ⟨term⟩l else ⟨term⟩r
```

The latter appears to be marginally shorter, but it is quite ambiguous, and indeed often requires an explicit annotation term : `_+_` to type-check, which evens the character count.

Therefore, SSReflect restricts by default the condition of a plain `if` construct to the standard bool type; this avoids spurious type annotations, e.g., in:

```
Definition orb b1 b2 := if b1 then true else b2.
```

As pointed out in section 1.2, this restriction can be removed with the command:

```
Close Scope boolean_if_scope.
```

Like `let:` above, the `if`...`is` construct supports the dependent `match` annotations:

```
if ⟨term⟩1 is ⟨pattern⟩1 as ⟨ident⟩ in ⟨pattern⟩2 return ⟨term⟩2
  then ⟨term⟩3
  else ⟨term⟩4
```

As in `let:` the variable $\langle ident \rangle$ (and those in the type pattern $\langle pattern \rangle_2$) are bound in $\langle term \rangle_2$; $\langle ident \rangle$ is also bound in $\langle term \rangle_3$ (but not in $\langle term \rangle_4$), while the variables in $\langle pattern \rangle_1$ are bound only in $\langle term \rangle_3$.

Another variant allows to treat the else case first:

```
if ⟨term⟩1 isn't ⟨pattern⟩1 then ⟨term⟩2 else ⟨term⟩3
```

Note that $\langle pattern \rangle_1$ eventually binds variables in $\langle term \rangle_3$ and not $\langle term \rangle_2$.

3.3 Parametric polymorphism

Unlike ML, polymorphism in core Gallina is explicit: the type parameters of polymorphic functions must be declared explicitly, and supplied at each point of use. However, Coq provides two features to suppress redundant parameters:

- Sections are used to provide (possibly implicit) parameters for a set of definitions.
- Implicit arguments declarations are used to tell Coq to use type inference to deduce some parameters from the context at each point of call.

The combination of these features provides a fairly good emulation of ML-style polymorphism, but unfortunately this emulation breaks down for higher-order programming. Implicit arguments are indeed not inferred at all points of use, but only at points of call, leading to expressions such as

```
Definition all_null (s : list d) := all (@null d) s.
```

Unfortunately, such higher-order expressions are quite frequent in representation functions, especially those which use Coq's `Structure`s to emulate Haskell type classes.

Therefore, SSReflect provides a variant of Coq's implicit argument declaration, which causes Coq to fill in some implicit parameters at each point of use, e.g., the above definition can be written:

```
Definition all_null (s : list d) := all null s.
```

Better yet, it can be omitted entirely, since `all_null s` isn't much of an improvement over `all null s`.

The syntax of the new declaration is

`Prenex Implicits` ⟨*ident*⟩$^+$ `.`

Let us denote $const_1 \dots const_n$ the list of identifiers given to a `Prenex Implicits` command. The command checks that each $const_i$ is the name of a functional constant, whose implicit arguments are prenex, i.e., the first $n_i > 0$ arguments of $const_i$ are implicit; then it assigns `Maximal Implicit` status to these arguments.

As these prenex implicit arguments are ubiquitous and have often large diaplay strings, it is strongly recommended to change the default display settings of Coq so that they are not printed (except after a `Set Printing All` command). All SSReflect library files thus start with the incantation

```
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

3.4 Anonymous arguments

When in a definition, the type of a certain argument is mandatory, but not its name, one usually use "arrow" abstractions for prenex arguments, or the `(_ :` ⟨*term*⟩`)` syntax for inner arguments. In SSReflect, the latter can be replaced by the open syntax '`of` ⟨*term*⟩' or (equivalently) '`&` ⟨*term*⟩', which are both syntactically equivalent to the standard Coq `(_ :` ⟨*term*⟩`)` expression.

Hence the following declaration:

```
Inductive list (A : Type) : Type := nil | cons of A & list A.
```

defines a type which is syntactically equal to the type `list` of the COQ standard `List` library.

3.5 Wildcards

As in standard Gallina, the terms passed as arguments to SSREFLECT tactics can contain *holes*, materialised by wildcards `_`. Since SSREFLECT allows a more powerful form of type inference for these arguments, it enhances the possibilities of using such wildcards. These holes are in particular used as a convenient shorthand for abstractions, especially in local definitions or type expressions.

Wildcards may be interpreted as abstractions (see for example sections 4.1 and 6.6), or their content can be inferred from the whole context of the goal (see for example section 4.2).

4 Definitions

4.1 Definitions

The standard `pose` tactic allows to add a defined constant to a proof context. SSREFLECT generalises this tactic in several ways. In particular, the SSREFLECT `pose` tactic supports *open syntax*: the body of the definition does not need surrounding parentheses. For instance:

```
pose t := x + y.
```

is a valid tactic expression.

The standard `pose` tactic is also improved for the local definition of higher order terms. Local definitions of functions can use the same syntax as global ones. The tactic:

```
pose f x y := x + y.
```

adds to the context the defined constant:

```
f := fun x y : nat => x + y : nat -> nat -> nat
```

The SSREFLECT `pose` tactic also supports (co)fixpoints, by providing the local counterpart of the `Fixpoint f := ...` and `CoFixpoint f := ...` constructs. For instance, the following tactic:

```
pose fix f (x y : nat) {struct x} : nat :=
    if x is S p then S (f p y) else 0.
```

defines a local fixpoint `f`, which mimics the standard `plus` operation on natural numbers.

Similarly, local cofixpoints can be defined by a tactic of the form:

```
pose cofix f (arg : T) ...
```

The possibility to include wildcards in the body of the definitions offers a smooth way of defining local abstractions. The type of "holes" is guessed by type inference, and the holes are abstracted. For instance the tactic:

```
pose f := _ + 1.
```

is shorthand for:

```
pose f n := n + 1.
```

When the local definition of a function involves both arguments and holes, hole abstractions appear first. For instance, the tactic:

```
pose f x := x + _.
```

is shorthand for:

```
pose f n x := x + n.
```

The interaction of the `pose` tactic with the interpretation of implicit arguments results in a powerful and concise syntax for local definitions involving dependant types. For instance, the tactic:

```
pose f x y := (x, y).
```

adds to the context the local definition:

```
pose f (Tx Ty : Type) (x : Tx) (y : Ty) := (x, y).
```

The generalisation of wildcards makes the use of the `pose` tactic ressemble ML-like definitions of polymorphic functions.

4.2 Abbreviations

The SSReflect `set` tactic performs abbreviations: it introduces a defined constant for a subterm appearing in the goal and/or in the context.

SSReflect extends the standard Coq `set` tactic by supplying:

- an open syntax, similarly to the `pose` tactic;
- a more aggressive matching algorithm;
- an improved interpretation of wildcards, taking advantage of the matching algorithm;
- an improved occurrence selection mechanism allowing to abstract only selected occurrences of a term.

The general syntax of this tactic is

$$\texttt{set}\ \langle \textit{ident}\rangle\ [\texttt{:}\ \langle \textit{term}\rangle_1]\ \texttt{:=}\ [\langle \textit{occ-switch}\rangle]\ \langle \textit{term}\rangle_2$$

$$\langle \textit{occ-switch}\rangle \ \equiv\ \texttt{\{}[\texttt{+}|\texttt{-}]\ \langle \textit{num}\rangle^*\texttt{\}}$$

where:

- ⟨*ident*⟩ is a fresh identifier chosen by the user.
- ⟨*term*⟩$_1$ is an optional type annotation. The type annotation ⟨*term*⟩$_1$ can be given in open syntax (no surrounding parentheses). If no ⟨*occ-switch*⟩ (described hereafter) is present, it is also the case for ⟨*term*⟩$_2$. On the other hand, in presence of ⟨*occ-switch*⟩, parentheses surrounding ⟨*term*⟩$_2$ are mandatory.
- In the occurrence switch ⟨*occ-switch*⟩, if the first element of the list is a ⟨*num*⟩, this element should be a number, and not an $\mathcal{L}$-tac variable. The empty list `{}` is not interpreted as a valid occurrence switch.

The tactic:

```
set t := f _.
```

transforms the goal `f x + f x = f x` into `t + t = t`, adding `t := f x` to the context, and the tactic:

```
set t := {2}(f _).
```

transforms it into `f x + t = f x`, adding `t := f x` to the context.

The type annotation ⟨*term*⟩$_1$ may contain wildcards, which will be filled with the appropriate value by the matching process.

The tactic first tries to find a subterm of the goal matching ⟨*term*⟩$_2$ (and its type ⟨*term*⟩$_1$), and stops at the first subterm it finds. Then the occurrences of this subterm selected by the optional ⟨*occ-switch*⟩ are replaced by ⟨*ident*⟩ and a definition ⟨*ident*⟩`:=`⟨*term*⟩ is added to the context. If no ⟨*occ-switch*⟩ is present, then all the occurrences are abstracted.

Matching

The matching algorithm compares a pattern *term* with a subterm of the goal by comparing their heads and then pairwise unifying their arguments (modulo conversion). Head symbols match under the following conditions:

- If the head of *term* is a constant, then it should be syntactically equal to the head symbol of the subterm.
- If this head is a projection of a canonical structure, then canonical structure equations are used for the matching.
- If the head of *term* is *not* a constant, the subterm should have the same structure (λ abstraction, `let...in` structure ...).
- If the head of *term* is a hole, the subterm should have at least as many arguments as *term*. For instance the tactic:

  ```
  set t := _ x.
  ```

transforms the goal `x + y = z` into `t y = z` and adds `t := plus x : nat -> nat` to the context.

- In the special case where *term* is of the form `(let f :=` t_0 `in f)` $t_1 \dots t_n$, then the pattern *term* is treated as `(_` $t_1 \dots t_n$`)`. For each subterm in the goal having the form $(A\ u_1 \dots u_{n'})$ with $n' \geq n$, the matching algorithm successively tries to find the largest partial application $(A\ u_1 \dots u_{i'})$ convertible to the head t_0 of *term*. For instance the following tactic:

  ```
  set t := (let g y z := y.+1 + z in g) 2.
  ```

  transforms the goal

  ```
  (let f x y z := x + y + z in f 1) 2 3 = 6.
  ```

  into `t 3 = 6` and adds the local definition of `t` to the context.

Moreover:

- Multiple holes in *term* are treated as independent placeholders. For instance, the tactic:

  ```
  set t := _ + _.
  ```

  transforms the goal `x + y = z` into `t = z` and pushes `t := x + y : nat` in the context.

- The type of the subterm matched should fit the type (possibly casted by some type annotations) of the pattern *term*.

- The replacement of the subterm found by the instanciated pattern should not capture variables, hence the following script:

  ```
  Goal forall x : nat, x + 1 = 0.
  set u := _ + 1.
  ```

  raises an error message, since `x` is bound in the goal.

- Typeclass inference should fill in any residual hole, but matching should never assign a value to a global existential variable.

Occurrence selection

SSReflect provides a generic syntax for the selection of occurrences by their position indexes. These *occurrence switches* are shared by all SSReflect tactics which require control on subterm selection like rewriting, generalisation, ...

An *occurrence switch* can be:

- A list `{+`⟨*num*⟩`*}` of occurrences affected by the tactic. For instance, the tactic:

  ```
  set x := {1 3}(f 2).
  ```

  transforms the goal `f 2 + f 8 = f 2 + f 2` into `x + f 8 = f 2 + x`, and adds the abbreviation `x := f 2` in the context. Notice that, like in standard Coq, some occurrences of a given term may be hidden to the user, for example because of a notation. The vernacular `Set Printing All` command displays all these hidden occurrences and should be used to find the correct coding of the occurrences to be selected[2]. For instance, both in SSReflect and in standard Coq, the following script:

  ```
  Notation "a < b":= (le (S a) b).
  Goal forall x y, x < y -> S x < S y.
  intros x y; set t := S x.
  ```

  generates the goal `t <= y -> t < S y` since `x < y` is now a notation for `S x <= y`.

- A list `{`⟨*num*⟩`+}`. This is equivalent to `{+`⟨*num*⟩`+}` but the list should start with a number, and not with an $\mathcal{L}$-tac variable.

- A list `{-`⟨*num*⟩`*}` of occurrences *not* to be affected by the tactic. For instance, the tactic:

  ```
  set x := {-2}(f 2).
  ```

  behaves like

  ```
  set x := {1 3}(f 2).
  ```

  on the goal `f 2 + f 8 = f 2 + f 2` which has three occurrences of the the term `f 2`

- In particular, the switch `{+}` selects *all* the occurrences. This switch is useful to turn off the default behaviour of a tactic which automatically clears some assumptions (see section 5.3 for instance).

- The switch `{-}` imposes that *no* occurrences of the term should be affected by the tactic. The tactic:

  ```
  set x := {-}(f 2).
  ```

  leaves the goal unchanged and adds the definition `x := f 2` to the context. This kind of tactic may be used to take advantage of the power of the matching algorithm in a local definition, instead of copying large terms by hand.

It is important to remember that matching *precedes* occurrence selection, hence the tactic:

[2]Unfortunately, even after a call to the Set Printing All command, some occurrences are still not displayed to the user, essentially the ones possibly hidden in the predicate of a dependent match structure.

```
set a := {2}(_ + _).
```

transforms the goal `x + y = x + y + z` into `x + y = a + z` and fails on the goal `(x + y)+ (z + z)= z + z` with the error message:

```
User error: only 1 < 2 occurrence of (x + y + (z + z))
```

4.3 Localisation

It is possible to define an abbreviation for a term appearing in the context of a goal thanks to the `in` tactical.

A tactic of the form:

`set x :=` *term* `in` $\mathit{fact}_1 \ldots \mathit{fact}_n$`.`

introduces a defined constant called `x` in the context, and folds it in the facts $\mathit{fact}_1 \ldots \mathit{fact}_n$ The body of `x` is the first subterm matching *term* in $\mathit{fact}_1 \ldots \mathit{fact}_n$.

A tactic of the form:

`set x :=` *term* `in` $\mathit{fact}_1 \ldots \mathit{fact}_n$ `*.`

matches $\langle \mathit{term} \rangle$ and then folds `x` similarly in $\mathit{fact}_1 \ldots \mathit{fact}_n$, but also folds `x` in the goal.

A goal `x + t = 4`, whose context contains `Hx : x = 3`, is left unchanged by the tactic:

```
set z := 3 in Hx.
```

but the context is extended with the definition `z := 3` and `Hx` becomes `Hx : x = z`. On the same goal and context, the tactic:

```
set z := 3 in Hx *.
```

will moreover change the goal into `x + t = S z`. Indeed, remember that `4` is just a notation for `(S 3)`.

The use of the `in` tactical is not limited to the localisation of abbreviations: for a complete description of the `in` tactical, see section 5.1.

5 Basic tactics

A sizable fraction of proof scripts consists of steps that do not "prove" anything new, but instead perform menial bookkeeping tasks such as selecting the names of constants and assumptions or splitting conjuncts. Indeed, SSReflect scripts appear to divide evenly between bookkeeping, formal algebra (rewriting), and actual deduction. Although they are logically trivial, bookkeeping steps are extremely important because they define the structure of the dataflow of a proof script. This is especially true for reflection-based proofs, which often involve large numbers of constants and assumptions. Good bookkeeping consists in always explicitly declaring (i.e., naming) all new constants and assumptions in the script, and systematically pruning irrelevant constants and assumptions in the context. This is essential in the context of an interactive development environment (IDE), because it facilitates navigating the proof, allowing to instantly "jump back" to the point at which a questionable assumption was added, and to find relevant assumptions by browsing the pruned context. While novice or casual Coq users may find the automatic name selection feature of Coq convenient, this feature severely undermines the readability and maintainability of proof scripts, much like automatic variable declaration in programming languages. The SSReflect tactics are therefore designed to support precise bookkeeping and to eliminate name generation heuristics. The bookkeeping features of SSReflect are implemented as tacticals (or pseudo-tacticals), shared across most SSReflect tactics, and thus form the foundation of the SSReflect proof language.

5.1 Bookkeeping

During the course of a proof Coq always present the user with a *sequent* whose general form is

$$\begin{array}{l}
c_i \ \texttt{:}\ T_i \\
\ldots \\
d_j \ \texttt{:=}\ e_j \ \texttt{:}\ T_j \\
\ldots \\
F_k \ \texttt{:}\ P_k \\
\ldots \\
\hline\hline
\texttt{forall}\ (x_\ell \ \texttt{:}\ T_\ell)\ \ldots, \\
\texttt{let}\ y_m \ \texttt{:=}\ b_m\ \texttt{in}\ \ldots\ \texttt{in} \\
P_n \ \texttt{->}\ \ldots\ \texttt{->}\ C
\end{array}$$

The *goal* to be proved appears below the double line; above the line is the *context* of the sequent, a set of declarations of *constants* c_i, *defined constants* d_i, and *facts* F_k that can be used to prove the goal (usually, T_i, T_j `:` `Type` and P_k `:` `Prop`). The various kinds of declarations can come in any order. The top part of the context consists of declarations produced by the `Section` commands `Variable`, `Let`, and `Hypothesis`. This *section context* is never affected by the SSReflect tactics: they only operate on the the lower part — the *proof context*. As in the figure above, the goal often decomposes into a series of (universally) quantified *variables* $(x_\ell : T_\ell)$, local *definitions* `let` y_m `:=` b_m `in`, and *assumptions* P_n `->`, and a *conclusion* C (as in the context, variables, definitions, and assumptions can appear in any order). The conclusion is what actually needs to be proved — the rest of the goal can be seen as a part of the proof context that happens to be "below the line".

However, although they are logically equivalent, there are fundamental differences between constants and facts on the one hand, and variables and assumptions on the others. Constants and facts are *unordered*, but *named* explicitly in the proof text; variables and assumptions are *ordered*, but *unnamed*: the display names of variables may change at any time because of α-conversion.

Similarly, basic deductive steps such as `apply` can only operate on the goal because the Gallina terms that control their action (e.g., the type of the lemma used by `apply`) only provide unnamed bound variables.[3] Since the proof script can only refer directly to the context, it must constantly shift declarations from the goal to the context and conversely in between deductive steps.

In SSReflect these moves are performed by two *tacticals* '`=>`' and '`:`', so that the bookkeeping required by a deductive step can be directly associated to that step, and that tactics in an SSReflect script correspond to actual logical steps in the proof rather than merely shuffle facts. Still, some isolated bookkeeping is unavoidable, such as naming variables and assumptions at the beginning of a proof. SSReflect provides a specific `move` tactic for this purpose.

Now `move` does essentially nothing: it is mostly a placeholder for '`=>`' and '`:`'. The '`=>`' tactical moves variables, local definitions, and assumptions to the context, while the '`:`' tactical moves facts and constants to the goal. For example, the proof of[4]

```
Lemma subnK : forall m n, n <= m -> m - n + n = m.
```

might start with

```
move=> m n le_n_m.
```

where `move` does nothing, but `=> m n le_m_n` changes the variables and assumption of the goal in the constants `m n : nat` and the fact `le_n_m : n <= m`, thus exposing the conclusion `m - n + n = m`. This is exactly what the specialized Coq tactic `intros m n le_m_n` would do, but '`=>`' is much more general (see 5.4).

The '`:`' tactical is the converse of '`=>`': it removes facts and constants from the context by turning them into variables and assumptions. Thus

```
move: m le_n_m.
```

turns back `m` and `le_m_n` into a variable and an assumption, removing them from the proof context, and changing the goal to

```
forall m, n <= m -> m - n + n = m.
```

which can be proved by induction on `n` using `elim n`. The specialized Coq tactic `revert` does exactly this, but '`:`' is much more general (see 5.3).
Because they are tacticals, '`:`' and '`=>`' can be combined, as in

```
move: m le_n_m => p le_n_p.
```

simultaneously renames `m` and `le_m_n` into `p` and `le_p_n`, respectively, by first turning them into unnamed variables, then turning these variables back into constants and facts.

Furthermore, SSReflect redefines the basic Coq tactics `case`, `elim`, and `apply` so that they can take better advantage of '`:`' and '`=>`'. The Coq tactics lack uniformity in that they

[3] Thus scripts that depend on bound variable names, e.g., via `intros` or `with`, are inherently fragile.
[4] The name `subnK` reads as "right cancellation rule for `nat` substraction".

require an argument from the context but operate on the goal. Their SSReflect counterparts use the first variable or constant of the goal instead, so they are "purely deductive": they do not use or change the proof context. There is no loss since ':' can readily be used to supply the required variable; for instance the proof of `subnK` could continue with

```
elim: n.
```

instead of `elim n`; this has the advantage of removing `n` from the context. Better yet, this `elim` can be combined with previous `move` and with the branching version of the `=>` tactical (described in 5.4), to encapsulate the inductive step in a single command:

```
elim: n m le_n_m => [|n IHn] m => [_ | lt_n_m].
```

which breaks down the proof into two subgoals,

```
m - 0 + 0 = m
```

given `m : nat`, and

```
m - S n + S n = m
```

given `m n : nat`, `lt_n_m : S n <= m`, and

```
IHn : forall m, n <= m -> m - n + n = m.
```

The ':' and '=>' tacticals can be explained very simply if one views the goal as a stack of variables and assumptions piled on a conclusion:

- *tactic*: a b c pushes the context constants a, b, c as goal variables *before* performing *tactic*.
- *tactic*=> a b c pops the top three goal variables as context constants a, b, c, *after* *tactic* has been performed.

These pushes and pops do not need to balance out as in the examples above, so

```
move: m le_n_m => p.
```

would rename `m` into `p`, but leave an extra assumption `n <= p` in the goal.

Basic tactics like `apply` and `elim` can also be used without the ':' tactical: for example we can directly start a proof of `subnK` by induction on the top variable `m` with

```
elim=> [|m IHm] n le_n.
```

The general form of the localisation tactical `in` is also best explained in terms of the goal stack:

```
tactic in a H1 H2 *.
```

is basically equivalent to

```
move: a H1 H2; tactic => a H1 H2.
```

with two differences: the `in` tactical will preserve the body of `a` if `a` is a defined constant, and if the '`*`' is omitted it will use a temporary abbreviation to hide the statement of the goal from *tactic*.

The general form of the `in` tactical can be used directly with the `move`, `case` and `elim` tactics, so that one can write

```
elim: n => [|n IHn] in m le_n_m *.
```

instead of

```
elim: n m le_n_m => [|n IHn] m le_n_m.
```

This is quite useful for inductive proofs that involve many facts.
See section 6.5 for the general syntax and presentation of the `in` tactical.

5.2 The defective tactics

In this section we briefly present the three basic tactics performing context manipulations and the main backward chaining tool.

The `move` tactic.

The `move` tactic, in its defective form, behaves like the primitive `hnf` Coq tactic. For example, such a defective:

```
move.
```

exposes the first assumption in the goal, i.e. its changes the goal `~ False` into `False -> False`.

More precisely, the `move` tactic inspects the goal and does nothing (`idtac`) if an introduction step is possible, i.e. if the goal is a product or a `let ... in`, and performs `hnf` otherwise.

Of course this tactic is most often used in combination with the bookkeeping tacticals (see section 5.4 and 5.3). These combinations mostly subsume the `intros`, `generalize`, `rename`, `clear` and `pattern` tactics.

The `case` tactic.

The `case` tactic, like in standard Coq, performs *primitive case analysis* on (co)inductive types; specifically, it destructs the top variable or assumption of the goal, exposing its constructor(s) and its arguments, as well as setting the value of its type family indices if it belongs to a type family (see section 5.6).

The SSReflect `case` tactic has a special behaviour on equalities.[5] If the top assumption of the goal is an equality, the `case` tactic "destructs" it as a set of equalities between the constructor arguments of its left and right hand sides, as per the standard Coq tactic `injection`. For example, `case` changes the goal

[5]The primitive Coq behaviour, rewriting right to left, is somewhat counterintuitive.

```
(x, y) = (1, 2) -> G.
```

into

```
x = 1 -> y = 2 -> G.
```

Note also that the case of SSReflect performs `False` elimination, even if no branch is generated by this case operation. Hence the command:

```
case.
```

on a goal of the form `False -> G` will succeed and prove the goal.

The `elim` tactic.

The `elim` tactic, like in standard Coq performs inductive elimination on inductive types. The defective:

```
elim.
```

tactic performs inductive elimination on a goal whose top assumption has an inductive type. For example on goal of the form:

```
forall n : nat, m <= n
```

in a context containing m : nat, the

```
elim.
```

tactic produces two goals,

```
m <= 0
```

on one hand and

```
forall n : nat, m <= n -> m <= S n
```

on the other hand.

The `apply` tactic.

The `apply` tactic is the main backward chaining tactic of the proof system. It takes as argument any *term* and applies it to the goal. Assumptions in the type of *term* that don't directly match the goal may generate one or more subgoals.

In fact the SSReflect tactic:

```
apply.
```

corresponds to the following standard Coq tactic:

```
intro top; first [refine top | refine (top _) | refine (top _ _) | ...];
    clear top.
```

where top is fresh name, and the sequence of refine tactics tries to catch the appropriate number of wildcards to be inserted.

This use of the refine tactic makes the SSReflect apply tactic considerably more robust than its standard Coq namesake, since it tries to match the goal up to expansion of constants and evaluation of subterms.

SSReflect's apply handles goals containing existential metavariables of sort Prop in a different way than standard Coq's apply. Consider the following example:

```
Goal (forall y, 1 < y -> y < 2 -> exists x : 'I_3, x > 0).
move=> y y_gt1 y_lt2; apply: (ex_intro _ (@Ordinal _ y _)).
  by apply: leq_trans y_lt2 _.
by move=> y_lt3; apply: leq_trans _ y_gt1.
```

Note that the last _ of the tactic apply: (ex_intro _ (@Ordinal _ y _)) represents a proof that y < 3. Instead of generating the following goal

```
0 < Ordinal (n:=3) (m:=y) ?54
```

the system tries to prove y < 3 calling the trivial tactic. If it succeeds, let's say because the context contains H : y < 3, then the system generates the following goal:

```
0 < Ordinal (n:=3) (m:=y) H
```

Otherwise the missing proof is considered to be irrelevant, and is thus discharged generating the following goals:

```
y < 3
forall Hyp0 : y < 3, 0 < Ordinal (n:=3) (m:=y) Hyp0
```

Last, the user can replace the trivial tactic by defining an $\mathcal{L}$-tac expression named ssrautoprop.

5.3 Discharge

The general syntax of the discharging tactical ':' is:

$$\langle \textit{tactic} \rangle\ [\langle \textit{ident} \rangle]\texttt{:}\ \langle \textit{d-item} \rangle_1\ \dots\ \langle \textit{d-item} \rangle_n\ [\langle \textit{clear-switch} \rangle]$$

where $n > 0$, and $\langle \textit{d-item} \rangle$ and $\langle \textit{clear-switch} \rangle$ are defined as

$$\begin{array}{rcl} \langle \textit{d-item} \rangle & \equiv & [\langle \textit{occ-switch} \rangle \mid \langle \textit{clear-switch} \rangle]\ \langle \textit{term} \rangle \\ \langle \textit{clear-switch} \rangle & \equiv & \texttt{\{}\langle \textit{ident} \rangle_1\ \dots\ \langle \textit{ident} \rangle_m\texttt{\}} \end{array}$$

with the following requirements:

- ⟨*tactic*⟩ must be one of the four basic tactics described in 5.2, i.e., move, case, elim or apply, the exact tactic (section 6.2), the congr tactic (section 7.4), or the application of the *view* tactical '/' (section 8.2) to one of move, case, or elim.

- The optional ⟨*ident*⟩ specifies *equation generation* (section 5.5), and is only allowed if ⟨*tactic*⟩ is `move`, `case` or `elim`, or the application of the view tactical '`/`' (section 8.2) to `case` or `elim`.
- An ⟨*occ-switch*⟩ selects occurrences of ⟨*term*⟩, as in 4.2; ⟨*occ-switch*⟩ is not allowed if ⟨*tactic*⟩ is `apply` or `exact`.
- A clear item ⟨*clear-switch*⟩ specifies facts and constants to be deleted from the proof context (as per the `clear` tactic).

The '`:`' tactical first *discharges* all the ⟨*d-item*⟩s, right to left, and then performs ⟨*tactic*⟩, i.e., for each ⟨*d-item*⟩, starting with $\langle \textit{d-item} \rangle_n$:

1. The SSREFLECT matching algorithm described in section 4.2 is used to find occurrences of ⟨*term*⟩ in the goal, after filling any holes '`_`' in ⟨*term*⟩; however if ⟨*tactic*⟩ is `apply` or `exact` a different matching algorithm, described below, is used [6].
2. These occurrences are replaced by a new variable, as per the standard COQ `revert` tactic; in particular, if ⟨*term*⟩ is a fact, this adds an assumption to the goal.
3. If ⟨*term*⟩ is *exactly* the name of a constant or fact in the proof context, it is deleted from the context as per the COQ `clear` tactic, unless there is an ⟨*occ-switch*⟩.

Finally, ⟨*tactic*⟩ is performed just after $\langle \textit{d-item} \rangle_1$ has been generalized — that is, between steps 2 and 3 for $\langle \textit{d-item} \rangle_1$. The names listed in the final ⟨*clear-switch*⟩ (if it is present) are cleared first, before $\langle \textit{d-item} \rangle_n$ is discharged.
Switches affect the discharging of a ⟨*d-item*⟩ as follows:

- An ⟨*occ-switch*⟩ restricts generalization (step 2) to a specific subset of the occurrences of ⟨*term*⟩, as per 4.2, and prevents clearing (step 3).
- All the names specified by a ⟨*clear-switch*⟩ are deleted from the context in step 3, possibly in addition to ⟨*term*⟩.

For example, the tactic:

```
move: n {2}n (refl_equal n).
```

- first generalizes `(refl_equal n : n = n)`;
- then generalizes the second occurrence of `n`.
- finally generalizes all the other occurrences of `n`, and clears `n` from the proof context (assuming `n` is a proof constant).

Therefore this tactic changes any goal `G` into

[6] Also, a slightly different variant may be used for the first ⟨*d-item*⟩ of `case` and `elim`; see section 5.6.

```
forall n n0 : nat, n = n0 -> G.
```

where the name `n0` is picked by the Coq display function, and assuming `n` appeared only in `G`.

Finally, note that a discharge operation generalizes defined constants as variables, and not as local definitions. To override this behaviour, prefix the name of the local definition with a `@`, like in `move: @n`.

This is in contrast with the behaviour of the `in` tactical (see section 6.5), which preserves local definitions by default.

Clear rules

The clear step will fail if ⟨*term*⟩ is a proof constant that appears in other facts; in that case either the facts should be cleared explicitly with a ⟨*clear-switch*⟩, or the clear step should be disabled. The latter can be done by adding an ⟨*occ-switch*⟩ or simply by putting parentheses around ⟨*term*⟩: both

```
move: (n).
```

and

```
move: {+}n.
```

generalize `n` without clearing `n` from the proof context.

The clear step will also fail if the ⟨*clear-switch*⟩ contains a ⟨*ident*⟩ that is not in the *proof* context. Note that SSReflect never clears a section constant.

If ⟨*tactic*⟩ is `move` or `case` and an equation ⟨*ident*⟩ is given, then clear (step 3) for ⟨*d-item*⟩$_1$ is suppressed (see section 5.5).

Matching for `apply` and `exact`

The matching algorithm for ⟨*d-item*⟩s of the SSReflect `apply` and `exact` tactics exploits the type of ⟨*d-item*⟩$_1$ to interpret wildcards in the other ⟨*d-item*⟩ and to determine which occurrences of these should be generalized. Therefore, ⟨*occur switch*⟩es are not needed for `apply` and `exact`.

Indeed, the SSReflect tactic `apply: H x` is equivalent to the standard Coq tactic

```
refine (@H _ ... _ x); clear H x
```

with an appropriate number of wildcards between `H` and `x`.

Note that this means that matching for `apply` and `exact` has much more context to interpret wildcards; in particular it can accomodate the '`_`' ⟨*d-item*⟩, which would always be rejected after '`move:`'. For example, the tactic

```
apply: trans_equal (Hfg _) _.
```

transforms the goal `f a = g b`, whose context contains `(Hfg : forall x, f x = g x)`, into `g a = g b`. This tactic is equivalent (see section 5.1) to:

```
refine (trans_equal (Hfg _) _).
```

and this is a common idiom for applying transitivity on the left hand side of an equation.

5.4 Introduction

The application of a tactic to a given goal can generate (quantified) variables, assumptions, or definitions, which the user may want to *introduce* as new facts, constants or defined constants, respectively. If the tactic splits the goal into several subgoals, each of them may require the introduction of different constants and facts. Furthermore it is very common to immediately destructure or rewrite with an assumption instead of adding it to the context, as the goal can often be simplified and even proved after this.

All these operations are performed by the introduction tactical '`=>`', whose general syntax is

$$\langle \textit{tactic} \rangle \; \texttt{=>} \; \langle \textit{i-item} \rangle_1 \ldots \langle \textit{i-item} \rangle_n$$

where $\langle \textit{tactic} \rangle$ can be any tactic, $n > 0$ and

$$\begin{array}{rcl} \langle \textit{i-item} \rangle & \equiv & \langle \textit{i-pattern} \rangle \mid \langle \textit{s-item} \rangle \mid \langle \textit{clear-switch} \rangle \mid \texttt{/}\langle \textit{term} \rangle \\ \langle \textit{s-item} \rangle & \equiv & \texttt{/=} \mid \texttt{//} \mid \texttt{//=} \\ \langle \textit{i-pattern} \rangle & \equiv & \langle \textit{ident} \rangle \mid \texttt{\_} \mid \texttt{?} \mid \texttt{*} \mid [\langle \textit{occ-switch} \rangle]\texttt{->} \mid [\langle \textit{occ-switch} \rangle]\texttt{<-} \mid \\ & & \texttt{[}\langle \textit{i-item} \rangle_1^* \; \texttt{|} \ldots \texttt{|} \; \langle \textit{i-item} \rangle_m^* \texttt{]} \mid \texttt{-} \end{array}$$

The '`=>`' tactical first executes $\langle \textit{tactic} \rangle$, then the $\langle \textit{i-item} \rangle$s, left to right, i.e., starting from $\langle \textit{i-item} \rangle_1$. An $\langle \textit{s-item} \rangle$ specifies a simplification operation; a $\langle \textit{clear switch} \rangle$ specifies context pruning as in 5.3. The $\langle \textit{i-pattern} \rangle$s are quite similar to COQ's *intro patterns*; each performs an introduction operation, i.e., pops some variables or assumptions from the goal.

An $\langle \textit{s-item} \rangle$ can simplify the set of subgoals or the subgoal themselves:

- `//` removes all the "trivial" subgoals that can be resolved by the SSREFLECT tactic `done` described in 6.2, i.e., it executes `try done`.
- `/=` simplifies the goal by performing partial evaluation, as per the COQ tactic `simpl`.[7]
- `//=` combines both kinds of simplification; it is equivalent to `/= //`, i.e., `simpl; try done`.

When an $\langle \textit{s-item} \rangle$ bears a $\langle \textit{clear-switch} \rangle$, then the $\langle \textit{clear-switch} \rangle$ is executed *after* the $\langle \textit{s-item} \rangle$, e.g., `{IHn}//` will solve some subgoals, possibly using the fact `IHn`, and will erase `IHn` from the context of the remaining subgoals.

The last entry in the $\langle \textit{i-item} \rangle$ grammar rule, `/`$\langle \textit{term} \rangle$, represents a view (see section 8). If $\langle \textit{i-item} \rangle_{k+1}$ is a view $\langle \textit{i-item} \rangle$, the view is appplied to the assumption in top position once $\langle \textit{i-item} \rangle_1 \ldots \langle \textit{i-item} \rangle_k$ have been performed.

The view is applied to the top assumption.

SSREFLECT supports the following $\langle \textit{i-pattern} \rangle$s:

[7]Except `/=` does not expand the local definitions created by the SSREFLECT `in` tactical.

- ⟨*ident*⟩ pops the top variable, assumption, or local definition into a new constant, fact, or defined constant ⟨*ident*⟩, respectively. As in Coq, defined constants cannot be introduced when δ-expansion is required to expose the top variable or assumption.

- `?` pops the top variable into an anonymous constant or fact, whose name is picked by the tactic interpreter. Unlike Coq, SSReflect only generates names that cannot appear later in the user script.[8]

- `_` pops the top variable into an anonymous constant that will be deleted from the proof context of all the subgoals produced by the `=>` tactical. They should thus never be displayed, except in an error message if the constant is still actually used in the goal or context after the last ⟨*i-item*⟩ has been executed (⟨*s-item*⟩s can erase goals or terms where the constant appears).

- `*` pops all the remaining apparent variables/assumptions as anonymous constants/-facts. Unlike `?` and `move` the `*` ⟨*i-item*⟩ does not expand definitions in the goal to expose quantifiers, so it may be useful to repeat a `move=> *` tactic, e.g., on the goal

  ```
  forall a b : bool, a <> b
  ```

  a first `move=> *` adds only `_a_ : bool` and `_b_ : bool` to the context; it takes a second `move=> *` to add `_Hyp_ : _a_ = _b_`.

- [⟨*occ-switch*⟩]`->` (resp. [⟨*occ-switch*⟩]`<-`) pops the top assumption (which should be a rewritable proposition) into an anonymous fact, rewrites (resp. rewrites right to left) the goal with this fact (using the SSReflect `rewrite` tactic described in section 7, and honouring the optional occurrence selector), and finally deletes the anonymous fact from the context.

- `[`⟨*i-item*⟩$^*_1$`|`...`|`⟨*i-item*⟩$^*_m$`]`, when it is the very *first* ⟨*i-pattern*⟩ after ⟨*tactic*⟩ `=>` tactical *and* ⟨*tactic*⟩ is not a `move`, is a *branching* ⟨*i-pattern*⟩. It executes the sequence ⟨*i-item*⟩$^*_i$ on the i^{th} subgoal produced by ⟨*tactic*⟩. The execution of ⟨*tactic*⟩ should thus generate exactly m subgoals, unless the `[...]` ⟨*i-pattern*⟩ comes after an initial `//` or `//=` ⟨*s-item*⟩ that closes some of the goals produced by ⟨*tactic*⟩, in which case exactly m subgoals should remain after the ⟨*s-item*⟩, or we have the trivial branching ⟨*i-pattern*⟩ `[]`, which always does nothing, regardless of the number of remaining subgoals.

- `[`⟨*i-item*⟩$^*_1$`|`...`|`⟨*i-item*⟩$^*_m$`]`, when it is *not* the first ⟨*i-pattern*⟩ or when ⟨*tactic*⟩ is a `move`, is a *destructing* ⟨*i-pattern*⟩. It starts by destructing the top variable, using the SSReflect `case` tactic described in 5.2. It then behaves as the corresponding branching ⟨*i-pattern*⟩, executing the sequence ⟨*i-item*⟩$^*_i$ in the i^{th} subgoal generated by the case analysis; unless we have the trivial destructing ⟨*i-pattern*⟩ `[]`, the latter should generate exactly m subgoals, i.e., the top variable should have an inductive type with exactly

[8] SSReflect reserves all identifiers of the form "`_x_`", which is used for such generated names.

m constructors.[9] While it is good style to use the ⟨*i-item*⟩$^*_i$ to pop the variables and assumptions corresponding to each constructor, this is not enforced by SSReflect.

- `-` does nothing, but counts as an intro pattern. It can be used to force the interpretation of `[`⟨*i-item*⟩$^*_1$`|`...`|`⟨*i-item*⟩$^*_m$`]` as a case analysis like in `move=> -[H1 H2]`. It can be used to visually link a view with a name like in `move=> /eqP-H1`. Last, it can serve as a separator between views. In section ?? it will be explained how `move=> /v1/v2` differs from `move=> /v1-/v2`.

Note that SSReflect does not support the alternative Coq syntax `(`⟨*ipat*⟩`,`...`,`⟨*ipat*⟩`)` for destructing intro-patterns.

Clears are deferred until the end of the intro pattern. For example, given the goal `0 < x -> (0 < x)&& (y < 2)`, the tactic `move=> {x} ->` successfully rewrites the goal and deletes `x` and the anonymous equation. If the cleared names are reused in the same intro pattern, a renaming is performed behind the scenes.

Facts mentioned in a clear switch must be valid names in the proof context (excluding the section context), unlike in the standard `clear` tactic.

The rules for interpreting branching and destructing ⟨*i-pattern*⟩ are motivated by the fact that it would be pointless to have a branching pattern if ⟨*tactic*⟩ is a `move`, and in most of the remaining cases ⟨*tactic*⟩ is `case` or `elim`, which implies destruction. The rules above imply that

```
move=> [a b].
case=> [a b].
case=> a b.
```

are all equivalent, so which one to use is a matter of style; `move` should be used for casual decompositions, such as splitting a pair, and `case` should be used for actual destructions, in particular for type families (see 5.6) and proof by contradiction.

The trivial branching ⟨*i-pattern*⟩ can be used to force the branching interpretation, e.g.,

```
case=> [] [a b] c.
move=> [[a b] c].
case; case=> a b c.
```

are all equivalent.

5.5 Generation of equations

The generation of named equations option stores the definition of a new constant as an equation. The tactic:

```
move En: (size l) => n.
```

where `l` is a list, replaces `size l` by `n` in the goal and adds the fact `En : size l = n` to the context. This is quite different from:

[9] More precisely, it should have a quantified inductive type with a assumptions and $m - a$ constructors.

```
pose n := (size l).
```

which generates a definition `n := (size l)`. It is not possible to generalise or rewrite such a definition; on the other hand, it is automatically expanded during computation, whereas expanding the equation `En` requires explicit rewriting.

The use of this equation name generation option with a `case` or an `elim` tactic changes the status of the first *i-item*, in order to deal with the possible parameters of the constants introduced.

On the goal `a <> b` where `a, b` are natural numbers, the tactic:

```
case E : a => [|n].
```

generates two subgoals. The equation `E : a = 0` (resp. `E : a = S n`, and the constant `n : nat`) has been added to the context of the goal `0 <> b` (resp. `S n <> b`).

If the user does not provide a branching *i-item* as first *i-item*, or if the *i-item* does not provide enough names for the arguments of a constructor, then the constants generated are introduced under fresh SSReflect names. For instance, on the goal `a <> b`, the tactic:

```
case E : a => H.
```

also generates two subgoals, both requiring a proof of `False`. The hypotheses `E : a = 0` and `H : 0 = b` (resp. `E : a = S _n_` and `H : S _n_ = b`) have been added to the context of the first subgoal (resp. the second subgoal).

Combining the generation of named equations mechanism with the `case` tactic strengthens the power of a case analysis. On the other hand, when combined with the `elim` tactic, this feature is mostly useful for debug purposes, to trace the values of decomposed parameters and pinpoint failing branches.

5.6 Type families

When the top assumption of a goal has an inductive type, two specific operations are possible: the case analysis performed by the `case` tactic, and the application of an induction principle, performed by the `elim` tactic. When this top assumption has an inductive type, which is moreover an instance of a type family, Coq may need help from the user to specify which occurrences of the parameters of the type should be substituted.

A specific `/` switch indicates the type family parameters of the type of a *d-item* immediately following this `/` switch, using the syntax:

$$[\texttt{case}|\texttt{elim}] : \langle \textit{d-item} \rangle^{+} / \langle \textit{d-item} \rangle^{*}$$

The ⟨*d-item*⟩s on the right side of the `/` switch are discharged as described in section 5.3. The case analysis or elimination will be done on the type of the top assumption after these discharge operations.

Every ⟨*d-item*⟩ preceding the `/` is interpreted as arguments of this type, which should be an instance of an inductive type family. These terms are not actually generalised, but rather selected for substitution. Occurrence switches can be used to restrict the substitution. If a ⟨*term*⟩ is left completely implicit (e.g. writing just `_`), then a pattern is inferred looking at the type of the top assumption. This allows for the compact syntax `case: {2}_ / eqP`, were `_` is interpreted as `(_ == _)`. Moreover if the ⟨*d-item*⟩s list is too short, it is padded with an initial sequence of `_` of the right length.

Here is a small example on lists. We define first a function which adds an element at the end of a given list.

```
Require Import List.

Section LastCases.
Variable A : Type.

Fixpoint add_last(a : A)(l : list A): list A :=
match l with
  |nil => a :: nil
  |hd :: tl => hd :: (add_last a tl)
end.
```

Then we define an inductive predicate for case analysis on lists according to their last element:

```
Inductive last_spec : list A -> Type :=
  | LastSeq0 : last_spec nil
  | LastAdd s x : last_spec (add_last x s).

  Theorem lastP : forall l : list A, last_spec l.
```

Applied to the goal:

```
Goal forall l : list A, (length l) * 2 = length (app l l).
```

the command:

```
move=> l; case: (lastP l).
```

generates two subgoals:

```
length nil * 2 = length (nil ++ nil)
```

and

```
forall (s : list A) (x : A),
  length (add_last x s) * 2 = length (add_last x s ++ add_last x s)
```

both having `l : list A` in their context.

Applied to the same goal, the command:

```
move=> l; case: l / (lastP l).
```

generates the same subgoals but `l` has been cleared from both contexts.

Again applied to the same goal, the command:

```
move=> l; case: {1 3}l / (lastP l).
```

generates the subgoals `length l * 2 = length (nil ++ l)` and `forall (s : list A)( x : A), length l * 2 = length (add_last x s++l)` where the selected occurrences on the left of the `/` switch have been substituted with `l` instead of being affected by the case analysis.

The equation name generation feature combined with a type family `/` switch generates an equation for the *first* dependent d-item specified by the user. Again starting with the above goal, the command:

```
move=> l; case E: {1 3}l / (lastP l)=>[|s x].
```

adds `E : l = nil` and `E : l = add_last x s`, respectively, to the context of the two subgoals it generates.

There must be at least one *d-item* to the left of the `/` switch; this prevents any confusion with the view feature. However, the *d-items* to the right of the `/` are optional, and if they are omitted the first assumption provides the instance of the type family.

The equation always refers to the first *d-item* in the actual tactic call, before any padding with initial `_`s. Thus, if an inductive type has two family parameters, it is possible to have SSReflect generate an equation for the second one by omitting the pattern for the first; note however that this will fail if the type of the second parameter depends on the value of the first parameter.

6 Control flow

6.1 Indentation and bullets

The linear development of Coq scripts gives little information on the structure of the proof. In addition, replaying a proof after some changes in the statement to be proved will usually not display information to distinguish between the various branches of case analysis for instance.

To help the user in this organisation of the proof script at development time, SSReflect provides some bullets to highlight the structure of branching proofs. The available bullets are `-`, `+` and `*`. Combined with tabulation, this lets us highlight four nested levels of branching; the most we have ever needed is three. Indeed, the use of "simpl and closing" switches, of terminators (see above section 6.2) and selectors (see section 6.3) is powerful enough to avoid most of the time more than two levels of indentation.

Here is a fragment of such a structured script:

```
case E1: (abezoutn _ _) => [[| k1] [| k2]].
- rewrite !muln0 !gexpn0 mulg1 => H1.
  move/eqP: (sym_equal F0); rewrite -H1 orderg1 eqn_mul1.
```

```
  by case/andP; move/eqP.
- rewrite muln0 gexpn0 mulg1 => H1.
  have F1: t %| t * S k2.+1 - 1.
    apply: (@dvdn_trans (orderg x)); first by rewrite F0; exact: dvdn_mull.
    rewrite orderg_dvd; apply/eqP; apply: (mulgI x).
    rewrite -{1}(gexpn1 x) mulg1 gexpn_add leq_add_sub //.
    by move: P1; case t.
  rewrite dvdn_subr in F1; last by exact: dvdn_mulr.
  + rewrite H1 F0 -{2}(muln1 (p ^ l)); congr (_ * _).
    by apply/eqP; rewrite -dvdn1.
  + by move: P1; case: (t) => [| [| s1]].
- rewrite muln0 gexpn0 mul1g => H1.
...
```

6.2 Terminators

To further structure scripts, SSREFLECT supplies *terminating* tacticals to explicitly close off tactics. When replaying scripts, we then have the nice property that an error immediately occurs when a closed tactic fails to prove its subgoal.

It is hence recommended practise that the proof of any subgoal should end with a tactic which *fails if it does not solve the current goal*. Standard COQ already provides some tactics of this kind, like `discriminate`, `contradiction` or `assumption`.

SSREFLECT provides a generic tactical which turns any tactic into a closing one. Its general syntax is:

by ⟨*tactic*⟩.

The $\mathcal{L}$-tac expression:

by [⟨*tactic*⟩$_1$ | [⟨*tactic*⟩$_2$ | ...].

is equivalent to:

[by ⟨*tactic*⟩$_1$ | by ⟨*tactic*⟩$_2$ | ...].

and this form should be preferred to the former.

In the script provided as example in section 6.1, the paragraph corresponding to each subcase ends with a tactic line prefixed with a `by`, like in:

```
by apply/eqP; rewrite -dvdn1.
```

The `by` tactical is implemented using the user-defined, and extensible `done` tactic. This `done` tactic tries to solve the current goal by some trivial means and fails if it doesn't succeed. Indeed, the tactic expression:

by ⟨*tactic*⟩.

is equivalent to:

```
⟨tactic⟩; done.
```

Conversly, the tactic

```
by [ ].
```

is equivalent to:

```
done.
```

The default implementation of the `done` tactic, in the `ssreflect.v` file, is:

```
Ltac done :=
  trivial; hnf; intros; solve
   [ do ![solve [trivial | apply: sym_equal; trivial]
         | discriminate | contradiction | split]
   | case not_locked_false_eq_true; assumption
   | match goal with H : ~ _ |- _ => solve [case H; trivial] end ].
```

The lemma `not_locked_false_eq_true` is needed to discriminate *locked* boolean predicates (see section 7.3). The iterator tactical `do` is presented in section 6.4. This tactic can be customised by the user, for instance to include an `auto` tactic.

A natural and common way of closing a goal is to apply a lemma which is the `exact` one needed for the goal to be solved. The defective form of the tactic:

```
exact.
```

is equivalent to:

```
do [done | by move=> top; apply top].
```

where `top` is a fresh name affected to the top assumption of the goal. This applied form is supported by the `:` discharge tactical, and the tactic:

```
exact: MyLemma.
```

is equivalent to:

```
by apply: MyLemma.
```

(see section 5.3 for the documentation of the `apply:` combination).

Warning The list of tactics, possibly chained by semi-columns, that follows a `by` keyword is considered as a parenthesised block applied to the current goal. Hence for example if the tactic:

```
by rewrite my_lemma1.
```

succeeds, then the tactic:

```
by rewrite my_lemma1; apply my_lemma2.
```

usually fails since it is equivalent to:

```
by (rewrite my_lemma1; apply my_lemma2).
```

6.3 Selectors

When composing tactics, the two tacticals `first` and `last` let the user restrict the application of a tactic to only one of the subgoals generated by the previous tactic. This covers the frequent cases where a tactic generates two subgoals one of which can be easily disposed of.

This is an other powerful way of linearisation of scripts, since it happens very often that a trivial subgoal can be solved in a less than one line tactic. For instance, the tactic:

$\langle tactic\rangle_1$`; last by` $\langle tactic\rangle_2$`.`

tries to solve the last subgoal generated by $\langle tactic\rangle_1$ using the $\langle tactic\rangle_2$, and fails if it does not succeeds. Its analogous

$\langle tactic\rangle_1$`; first by` $\langle tactic\rangle_2$`.`

tries to solve the first subgoal generated by $\langle tactic\rangle_1$ using the tactic $\langle tactic\rangle_2$, and fails if it does not succeeds.

SSReflect also offers an extension of this facility, by supplying tactics to *permute* the subgoals generated by a tactic. The tactic:

$\langle tactic\rangle$`; last first.`

inverts the order of the subgoals generated by $\langle tactic\rangle$. It is equivalent to:

$\langle tactic\rangle$`; first last.`

More generally, the tactic:

$\langle tactic\rangle$`; last` $\langle num\rangle$ `first.`

where $\langle num\rangle$ is standard Coq numeral or $\mathcal{L}$-tac variable denoting
a standard Coq numeral having the value k, rotates the n subgoals $G_1, \ldots, G_n$ generated by $\langle tactic\rangle$. The first subgoal becomes G_{n+1-k} and the circular order of subgoals remains unchanged.

Conversly, the tactic:

$\langle tactic\rangle$`; first` $\langle num\rangle$ `last.`

rotates the n subgoals $G_1, \ldots, G_n$ generated by `tactic` in order that the first subgoal becomes G_k.

Finally, the tactics `last` and `first` combine with the branching syntax of $\mathcal{L}$-tac: if the tactic $\langle tactic\rangle_0$ generates n subgoals on a given goal, then the tactic

$tactic_0$`; last` $\langle num\rangle$ `[`$tactic_1$`|...|`$tactic_m$`] ||` $tactic_{m+1}$`.`

where $\langle num\rangle$ denotes the integer k as above, applies $tactic_1$ to the $n-k+1$-th goal, $\ldots$ $tactic_m$ to the $n-k+2-m$-th goal and $tactic_{m+1}$ to the others.

For instance, the script:

```
Inductive test : nat -> Prop :=
  C1 : forall n, test n | C2 : forall n, test n |
  C3 : forall n, test n | C4 : forall n, test n.

Goal forall n, test n -> True.
move=> n t; case: t; last 2 [move=> k| move=> l]; idtac.
```

creates a goal with four subgoals, the first and the last being `nat -> True`, the second and the third being `True` with respectively `k : nat` and `l : nat` in their context.

6.4 Iteration

SSReflect offers an accurate control on the repetition of tactics, thanks to the `do` tactical, whose general syntax is:

`do` [⟨*mult*⟩] `[` ⟨*tactic*⟩$_1$ `|` ... `|` ⟨*tactic*⟩$_n$ `]`

where ⟨*mult*⟩ is a *multiplier*.

Brackets can only be omitted if a single tactic is given *and* a multiplier is present.

A tactic of the form:

`do [`*tactic*$_1$ `| ... |` *tactic*$_n$`].`

is equivalent to the standard $\mathcal{L}$-tac expression:

`first [`*tactic*$_1$ `| ... |` *tactic*$_n$`].`

The optional multiplier ⟨*mult*⟩ specifies how many times the action of ⟨*tactic*⟩ should be repeated on the current subgoal.

There are four kinds of multipliers:

- n`!`: the step ⟨*tactic*⟩ is repeated exactly n times (where n is a positive integer argument).
- `!`: the step ⟨*tactic*⟩ is repeated as many times as possible, and done at least once.
- `?`: the step ⟨*tactic*⟩ is repeated as many times as possible, optionally.
- n`?`: the step ⟨*tactic*⟩ is repeated up to n times, optionally.

For instance, the tactic:

⟨*tactic*⟩`; do 1?rewrite mult_comm.`

rewrites at most one time the lemma `mult_com` in all the subgoals generated by ⟨*tactic*⟩, whereas the tactic:

⟨*tactic*⟩`; do 2!rewrite mult_comm.`

rewrites exactly two times the lemma `mult_com` in all the subgoals generated by ⟨*tactic*⟩, and fails if this rewrite is not possible in some subgoal.

Note that the combination of multipliers and `rewrite` is so often used that multipliers are in fact integrated to the syntax of the SSReflect `rewrite` tactic, see section 7.

6.5 Localisation

In sections 4.3 and 5.1, we have already presented the *localisation* tactical `in`, whose general syntax is:

$$\langle tactic \rangle \ \texttt{in} \ \langle ident \rangle^+ [\texttt{*}]$$

where $\langle ident \rangle^+$ is a non empty list of fact names in the context. On the left side of `in`, $\langle tactic \rangle$ can be `move`, `case`, `elim`, `rewrite`, `set`, or any tactic formed with the general iteration tactical `do` (see section 6.4).

The operation described by $\langle tactic \rangle$ is performed in the facts listed in $\langle ident \rangle^+$ and in the goal if a `*` ends the list.

The `in` tactical successively:

- generalises the selected hypotheses, possibly "protecting" the goal if `*` is not present,
- performs $\langle tactic \rangle$, on the obtained goal,
- reintroduces the generalised facts, under the same names.

This defective form of the `do` tactical is useful to avoid clashes between standard $\mathcal{L}$-tac `in` and the SSREFLECT tactical `in`. For example, in the following script:

```
Ltac mytac H := rewrite H.

Goal forall x y, x = y -> y = 3 -> x + y = 6.
move=> x y H1 H2.
do [mytac H2] in H1 *.
```

the last tactic rewrites the hypothesis `H2 : y = 3` both in `H1 : x = y` and in the goal `x + y = 6`.

By default `in` keeps the body of local definitions. To erase the body of a local definition during the generalization phase, the name of the local definition must be written between parentheses, like in `rewrite H in H1 (def_n) H2`.

6.6 Structure

Forward reasoning structures the script by explicitly specifying some assumptions to be added to the proof context. It is closely associated with the declarative style of proof, since an extensive use of these highlighted statements make the script closer to a (very detailed) text book proof.

Forward chaining tactics allow to state an intermediate lemma and start a piece of script dedicated to the proof of this statement. The use of closing tactics (see section 6.2) and of indentation makes syntactically explicit the portion of the script building the proof of the intermediate statement.

The `have` tactic.

The main SSReflect forward reasoning tactic is the `have` tactic. It can be use in two modes: one starts a new (sub)proof for an intermediate result in the main proof, and the other provides explicitly a proof term for this intermediate step.

In the first mode, the syntax of `have` in its defective form is:

`have:` ⟨*term*⟩.

This tactic supports open syntax for ⟨*term*⟩. Apart from the open syntax, when ⟨*term*⟩ does not contain any wildcard, this tactic is almost[10] equivalent to the standard Coq:

`assert` ⟨*term*⟩.

Applied to a goal `G`, it generates a first subgoal requiring a proof of ⟨*term*⟩ is the context of `G`. The difference with the standard Coq tactic is that the second subgoal generated is of the form ⟨*term*⟩`-> G`, where ⟨*term*⟩ becomes the new top assumption, instead of being introduced with a fresh name.

Like in the case of the `pose` tactic (see section 4.1), the types of the holes are abstracted in ⟨*term*⟩. For instance, the tactic:

```
have: _ * 0 = 0.
```

is equivalent to:

```
have: forall n : nat, n * 0 = 0.
```

The `have` tactic also enjoys the same abstraction mechanism as the `pose` tactic for the non-inferred implicit arguments. For instance, the tactic:

```
have: forall x y, (x, y) = (x, y + 0).
```

opens a new subgoal to prove that:
`forall (T : Type)(x : T)(y : nat), (x, y)= (x, y + 0)`

The behaviour of the defective `have` tactic makes it possible to generalise it in the following general construction:

`have` ⟨*i-item*⟩* [⟨*i-pattern*⟩] [⟨*s-item*⟩ | ⟨*binder*⟩+] [`:` ⟨*term*⟩₁] [`:=`⟨*term*⟩₂ | `by`⟨*tactic*⟩]

Open syntax is supported for ⟨*term*⟩₁ and ⟨*term*⟩₂. For the description of *i-items* and clear switches see section 5.4. The first mode of the `have` tactic, which opens a subproof for an intermediate result, uses tactics of the form:

`have` ⟨*clear-switch*⟩ ⟨*i-item*⟩ `:` *term* `by` *tactic*.

which behave like:

[10] The `assert` tactic creates a ζ redex, whereas the `have` tactic creates a β redex, and it introduces the lemma under an automatically chosen fresh name.

```
have: term; first by tactic.
move=> ⟨clear-switch⟩ ⟨i-item⟩.
```

Note that the ⟨*clear-switch*⟩ *precedes* the ⟨*i-item*⟩, which allows to reuse a name of the context, possibly used by the proof of the assumption, to introduce the new assumption itself.

Hence the standard COQ:

```
assert ⟨term⟩.
```

is in fact equivalent[11] up to the open syntax to:

```
have ?: ⟨term⟩.
```

The `by` feature is especially convenient when the proof script of the statement is very short, basically when it fits in one line like in:

```
have H : forall x y, x + y = y + x by move=> x y; rewrite addnA.
```

The possibility of using *i-items* supplies a very concise syntax for the further use of the intermediate step. For instance,

```
have -> : forall x, x * a = a.
```

on a goal `G`, opens a new subgoal asking for a proof of `forall x, x * a = a`, and a second subgoal in which the lemma `forall x, x * a = a` has been rewritten in the goal `G`. Note that in this last subgoal, the intermediate result does not appear in the context. Note that, thanks to the deferred execution of clears, the following idiom is supported (assuming `x` occurs in the goal only):

```
have {x} -> : x = y
```

An other frequent use of the intro patterns combined with `have` is the destruction of existential assumptions like in the tactic:

```
have [x Px]: exists x : nat, x > 0.
```

which opens a new subgoal asking for a proof of `exists x : nat, x > 0` and a second subgoal in which the witness is introduced under the name `x : nat`, and its property under the name `Px : x > 0`.

An alternative use of the `have` tactic is to provide the explicit proof term for the intermediate lemma, using tactics of the form:

```
have [⟨ident⟩] := ⟨term⟩.
```

This tactic creates a new assumption of type the type of ⟨*term*⟩. If the optional ⟨*ident*⟩ is present, this assumption is introduced under the name ⟨*ident*⟩. Note that the body of the constant is lost for the user.

Again, non inferred implicit arguments and explicit holes are abstracted. For instance, the tactic:

[11] again, except that the kind of redex created is different

```
have H := forall x, (x, x) = (x, x).
```

adds to the context `H : Type -> Prop`. This is a schematic example but the feature is specially useful when the proof term to give involves for instance a lemma with some hidden implicit arguments.

After the ⟨*i-pattern*⟩, a list of binders is allowed. For example, if `Pos_to_P` is a lemma that proves that `P` holds for any positive, the following command:

```
have H x (y : nat) : 2 * x + y = x + x + y by auto.
```

will put in the context `H : forall x, 2 * x = x + x`. A proof term provided after `:=` can mention these bound variables (that are automatically introduced with the given names). Since the ⟨*i-pattern*⟩ can be omitted, to avoid ambiguitiy, bound variables can be surrounded with parentheses even if no type is specified:

```
have (x) : 2 * x = x + x by auto.
```

The ⟨*i-item*⟩s and ⟨*s-item*⟩ can be used to interpret the asserted hypothesis with views (see section 8) or simplify the resulting goals.

The `have` tactic also supports a `suff` modifier which allows for asserting that a given statement implies the current goal without copying the goal itself. For example, given a goal `G` the tactic `have suff H : P` results in the following two goals:

```
|- P -> G
H : P -> G |- G
```

Note that `H` is introduced in the second goal. The `suff` modifier is not compatible with the presence of a list of binders.

Variants: the `suff` and `wlog` tactics.

As it is often the case in mathematical textbooks, forward reasoning may be used in slightly different variants. One of these variants is to show that the intermediate step L easily implies the initial goal G. By easily we mean here that the proof of $L \Rightarrow G$ is shorter than the one of L itself. This kind of reasoning step usually starts with: “It suffices to show that ...”.

This is such a frequent way of reasoning that SSReflect has a variant of the `have` tactic called `suffices` (whose abridged name is `suff`). The `have` and `suff` tactics are equivalent and have the same syntax but:

- the order of the generated subgoals is inversed
- but the optional clear item is still performed in the *second* branch. This means that the tactic:

  ```
  suff {H} H : forall x : nat, x >= 0.
  ```

  fails if the context of the current goal indeed contains an assumption named `H`.

The rationale of this clearing policy is to make possible "trivial" refinements of an assumption, without changing its name in the main branch of the reasoning.

The `have` modifier can follow the `suff` tactic. For example, given a goal `G` the tactic `suff have H : P` results in the following two goals:

```
H : P |- G
|- (P -> G) -> G
```

Note that, in contrast with `have suff`, the name `H` has been introduced in the first goal.

Another useful construct is reduction, showing that a particular case is in fact general enough to prove a general property. This kind of reasoning step usually starts with: "Without loss of generality, we can suppose that ...". Formally, this corresponds to the proof of a goal `G` by introducing a cut *wlog_statement*`-> G`. Hence the user shall provide a proof for both `(`*wlog_statement*`-> G)-> G` and *wlog_statement*`-> G`.

SSReflect implements this kind of reasoning step through the `without loss` tactic, whose short name is `wlog`. The general syntax of `without loss` is:

`wlog` [`suff`][⟨*clear-switch*⟩][⟨*i-item*⟩] `:` [⟨*ident*⟩$_1$ ... ⟨*ident*⟩$_n$] `/` ⟨*term*⟩

where ⟨*ident*⟩$_1$ ... ⟨*ident*⟩$_n$ are identifiers for constants in the context of the goal. Open syntax is supported for ⟨*term*⟩.

In its defective form:

`wlog: /` ⟨*term*⟩`.`

on a goal `G`, it creates two subgoals, respectively ⟨*term*⟩`-> G` and `(`⟨*term*⟩`-> G)-> G`.

If the optional list ⟨*ident*⟩$_1$ ... ⟨*ident*⟩$_n$ is present on the left side of `/`, these constants are generalised on top of the first ⟨*term*⟩`-> G` subgoal. By default the body of local definitions is erased. This behaviour can be inhibited prefixing the name of the local definition with the `@` character.

In the second subgoal, the tactic:

`move=>` ⟨*clear-switch*⟩ ⟨*i-item*⟩`.`

is performed if at least one of these optional switches is present in the `wlog` tactic.

The `wlog` tactic is specially useful when a symmetry argument simplifies a proof. Here is an example showing the begining of the proof that quotient and reminder of natural number euclidean division are unique.

```
Lemma quo_rem_unicity: forall d q1 q2 r1 r2,
   q1*d + r1 = q2*d + r2 -> r1 < d -> r2 < d -> (q1, r1) = (q2, r2).
move=> d q1 q2 r1 r2.
wlog: q1 q2 r1 r2 / q1 <= q2.
  by case (le_gt_dec q1 q2)=> H; last symmetry; eauto with arith.
```

The `wlog suff` variant is simpler, since it cuts *wlog_statement* instead of *wlog_statement*`-> G`. It thus opens the goals *wlog_statement*`-> G` and *wlog_statement*.

Note that the list of generalised constants on the left side of the `/` switch can contain clear items between constants. These clear operations are intertwined with the generalisation ones, which helps in particular avoiding dependency issues while generalising some facts.

7 Rewriting

The generalised use of reflection implies that most of the intermediate results handled are properties of effectively computable functions. The most efficient mean of establishing such results are computation and simplification of expressions involving such functions, i.e., rewriting. We have therefore defined an extended `rewrite` tactic that unifies and combines most of the rewriting functionalities.

7.1 An extended `rewrite` tactic

The main improvements brought to the standard Coq `rewrite` tactic are:

- Whereas the primitive `rewrite` tactic can only perform a single rewriting operation in the goal or in the context, the extended `rewrite` can perform an entire series of such operations in any subset of the goal and/or context;
- The SSReflect `rewrite` tactic allows to perform rewriting, simplifications, folding/unfolding of definitions, closing of goals;
- Several rewriting operations can be chained in a single tactic;
- Control over the occurrence at which rewriting is to be performed is significantly enhanced.

The general form of an SSReflect rewrite tactic is:

`rewrite` $\langle rstep \rangle^+$`.`

The combination of a rewrite tactic with the `in` tactical (see section 4.3) performs rewriting in both the context and the goal.

A rewrite step $\langle rstep \rangle$ has the general form:

$$[\langle r\text{-}prefix \rangle]\langle r\text{-}item \rangle$$

where:

$$\begin{array}{rcl}
\langle r\text{-}prefix \rangle & \equiv & [\texttt{-}]\ [\langle mult \rangle]\ [\langle occ\text{-}switch \rangle | \langle clear\text{-}switch \rangle]\ [\texttt{[}\langle r\text{-}pattern \rangle\texttt{]}] \\
\langle r\text{-}pattern \rangle & \equiv & \langle term \rangle \mid \texttt{in}\ [\langle ident \rangle\ \texttt{in}]\ \langle term \rangle \mid [\langle term \rangle\ \texttt{in} \mid \langle term \rangle\ \texttt{as}]\ \langle ident \rangle\ \texttt{in}\ \langle term \rangle \\
\langle r\text{-}item \rangle & \equiv & [\texttt{/}]\langle term \rangle \mid \langle s\text{-}item \rangle
\end{array}$$

An ⟨*r-prefix*⟩ contains annotations to qualify where and how the rewrite operation should be performed:

- The optional initial `-` indicates the direction of the rewriting of ⟨*r-item*⟩.
- The multiplier ⟨*mult*⟩ (see section 6.4) specifies if and how the rewrite operation should be repeated.
- A rewrite operation matches the occurrences of a *rewrite pattern*, and replaces these occurrences by an other term, according to the given ⟨*r-item*⟩. The optional *redex switch* `[`⟨*r-pattern*⟩`]`, which should always be surrounded by brackets, gives explicitly this rewrite pattern. In its simplest form, it is a regular term. If no explicit redex switch is present the rewrite pattern to be matched is inferred from the ⟨*r-item*⟩.
- This optional ⟨*term*⟩, or the ⟨*r-item*⟩, may be preceded by an occurrence switch (see section 6.3) or a clear item (see section 5.3), these two possibilities being exclusive. An occurrence switch selects the occurrences of the rewrite pattern which should be affected by the rewrite operation.

An ⟨*r-item*⟩ can be:

- A *simplification r-item*, represented by a ⟨*s-item*⟩ (see section 5.4). Simplification operations are intertwined with the possible other rewrite operations specified by the list of r-items.
- A *folding/unfolding r-item*. The tactic:

  `rewrite /`*term*

  unfolds the head constant of *term* in every occurrence of the first matching of *term* in the goal. In particular, if `my_def` is a (local or global) defined constant, the tactic:

  ```
  rewrite /my_def.
  ```

  is in principle[12] equivalent to:

  ```
  unfold my_def.
  ```

  Conversely:

  ```
  rewrite -/my_def.
  ```

  is equivalent to:

  ```
  fold my_def.
  ```

  When an unfold r-item is combined with a redex pattern, a conversion operation is performed. A tactic of the form:

[12] The implementation of these fold/unfold tactics does not call standard Coq `fold` and `unfold`.

```
rewrite -[⟨term⟩1]/⟨term⟩2.
```

is equivalent to:

```
change ⟨term⟩1 with ⟨term⟩2.
```

If $\langle term\rangle_2$ is a single constant and $\langle term\rangle_1$ head symbol is not $\langle term\rangle_2$, then the head symbol of $\langle term\rangle_1$ is repeatedly unfolded until $\langle term\rangle_2$ appears.

```
Definition double x := x + x.
Definition ddouble x := double (double x).
Lemma ex1 x : ddouble x = 4 * x.
rewrite [ddouble _]/double.
```

The resulting goal is:

```
double x + double x = 4 * x
```

Warning The SSReflect terms containing holes are *not* typed as abstractions in this context. Hence the following script:

```
Definition f := fun x y => x + y.
Goal forall x y, x + y = f y x.
move=> x y.
rewrite -[f y]/(y + _).
```

raises the error message

```
User error: fold pattern (y + _) does not match redex (f y)
```

but the script obtained by replacing the last line with:

```
rewrite -[f y x]/(y + _).
```

is valid.

- A term, which can be:
  - A term whose type has the form:

    $$\texttt{forall}\ (x_1\ :\ A_1)\dots(x_n\ :\ A_n),\ eq\ term_1\ term_2$$

    where *eq* is the Leibniz equality or a registered setoid equality.
  - A list of terms $(t_1, \dots, t_n)$, each t_i having a type of the form:

    $$\texttt{forall}\ (x_1\ :\ A_1)\dots(x_n\ :\ A_n),\ eq\ term_1\ term_2$$

    where *eq* is the Leibniz equality or a registered setoid equality. The tactic:

`rewrite` *r–prefix* `(`t_1`,...,`t_n`).`

is equivalent to:

`do [rewrite` *r–prefix* t_1 `| ... | rewrite` *r–prefix* t_n`].`

– An anonymous rewrite lemma `(_ :` *term*`)`, where *term* has again the form:

$$\texttt{forall}\ (x_1\ :\ A_1)\ldots(x_n\ :\ A_n),\ eq\ term_1\ term_2$$

The tactic:

`rewrite (_ :` *term*`)`

is in fact equivalent to the standard COQ:

`cutrewrite (`*term*`).`

7.2 Remarks and examples

Rewrite redex selection

The general strategy of SSREFLECT is to grasp as many redexes as possible and to let the user select the ones to be rewritten thanks to the improved syntax for the control of rewriting.

This may be a source of incompatibilities between SSREFLECT and standard COQ.

In a rewrite tactic of the form:

`rewrite` ⟨*occ-switch*⟩`[`⟨*term*⟩$_1$`]`⟨*term*⟩$_2$`.`

⟨*term*⟩$_1$ is the explicit rewrite redex and ⟨*term*⟩$_2$ is the rewrite rule. This execution of this tactic unfolds as follows:

- First ⟨*term*⟩$_1$ and ⟨*term*⟩$_2$ are $\beta\iota$ normalised. Then ⟨*term*⟩$_2$ is put in head normal form if the Leibniz equality constructor `eq` is not the head symbol. This may involve ζ reductions.
- Then, the matching algorithm (see section 4.2) determines the first subterm of the goal matching the rewrite pattern. The rewrite pattern is given by ⟨*term*⟩$_1$, if an explicit redex pattern switch is provided, or by the type of ⟨*term*⟩$_2$ otherwise. However, matching skips over matches that would lead to trivial rewrites. All the occurrences of this subterm in the goal are candidates for rewriting.
- Then only the occurrences coded by ⟨*occ-switch*⟩ (see again section 4.2) are finally selected for rewriting.
- The left hand side of ⟨*term*⟩$_2$ is unified with the subterm found by the matching algorithm, and if this succeeds, all the selected occurrences in the goal are replaced by the right hand side of ⟨*term*⟩$_2$.

- Finally the goal is $\beta\iota$ normalised.

In the case $\langle term \rangle_2$ is a list of terms, the first top-down (in the goal) left-to-right (in the list) matching rule gets selected.

Chained rewritings

The possibility to chain rewrite operations in a single tactic makes scripts more compact and gathers in a single command line a bunch of surgical operations which would be described by a one sentence in a pen and paper proof.

Performing rewrite and simplification operations in a single tactic enhances significantly the concision of scripts. For instance the tactic:

```
rewrite /my_def {2}[f _]/= my_eq //=.
```

unfolds `my_def` in the goal, simplifies the second occurrence of the first subterm matching pattern `[f _]`, rewrites `my_eq`, simplifies the whole goal and closes trivial goals.

Here are some concrete examples of chained rewrite operations, in the proof of basic results on natural numbers arithmetic:

```
Lemma addnS : forall m n, m + n.+1 = (m + n).+1.
Proof. by move=> m n; elim: m. Qed.

Lemma addSnnS : forall m n, m.+1 + n = m + n.+1.
Proof. move=> *; rewrite addnS; apply addSn. Qed.

Lemma addnCA : forall m n p, m + (n + p) = n + (m + p).
Proof. by move=> m n; elim: m => [|m Hrec] p; rewrite ?addSnnS -?addnS.
    Qed.

Lemma addnC : forall m n, m + n = n + m.
Proof. by move=> m n; rewrite -{1}[n]addn0 addnCA addn0. Qed.
```

Note the use of the `?` switch for parallel rewrite operations in the proof of addnCA.

Explicit redex switches are matched first

If an $\langle$*r-prefix*$\rangle$ involves a *redex switch*, the first step is to find a subterm matching this redex pattern, independently from the left hand side `t1` of the equality the user wants to rewrite.

For instance, if `H : forall t u, t + u = u + t` is in the context of a goal `x + y = y + x`, the tactic:

```
rewrite [y + _]H.
```

transforms the goal into `x + y = x + y`.

Note that if this first pattern matching is not compatible with the *r-item*, the rewrite fails, even if the goal contains a correct redex matching both the redex switch and the left hand side of the equality. For instance, if `H : forall t u, t + u * 0 = t` is in the context of a goal `x + y * 4 + 2 * 0 = x + 2 * 0`, then tactic:

```
rewrite [x + _]H.
```

raises the error message:

```
User error: rewrite rule H doesn't match redex (x + y * 4)
```

while the tactic:

```
rewrite (H _ 2).
```

transforms the goal into `x + y * 4 = x + 2 * 0`.

Occurrence switches and redex switches

The tactic:

```
rewrite {2}[_ + y + 0](_: forall z, z + 0 = z).
```

transforms the goal:

```
x + y + 0 = x + y + y + 0 + 0 + (x + y + 0)
```

into:

```
x + y + 0 = x + y + y + 0 + 0 + (x + y)
```

and generates a second subgoal:

```
forall z : nat, z + 0 = z
```

The second subgoal is generated by the use of an anonymous lemma in the rewrite tactic. The effect of the tactic on the initial goal is to rewrite this lemma at the second occurrence of the first matching `x + y + 0` of the explicit rewrite redex `_ + y + 0`.

Occurrence selection and repetition

Occurrence selection has priority over repetition switches. This means the repetition of a rewrite tactic specified by a multiplier will perform matching each time an elementary rewrite operation is performed. Repeated rewrite tactics apply to every subgoal generated by the previous tactic, including the previous instances of the repetition. For example:

```
Goal forall x y z : nat, x + 1 = x + y + 1.
move=> x y z.
```

creates a goal `x + 1 = x + y + 1`, which is turned into `z = z` by the additional tactic:

```
rewrite 2!(_ : _ + 1 = z).
```

Moreover, this last tactic generates *three* other subgoals, respectively, `x + y + 1 = z`, `z = z` and `x + 1 = z`. Indeed, the second rewrite operation specified with the `2!` multiplier applies to the two subgoals generated by the first rewrite.

Wildcards vs abstractions

The `rewrite` tactic supports r-items containing holes. For example in the tactic (1):

```
rewrite (_ : _ * 0 = 0).
```

the term `_ * 0 = 0` is interpreted as `forall n : nat, n * 0 = 0`. Anyway this tactic is *not* equivalent to the tactic (2):

```
rewrite (_ : forall x, x * 0 = 0).
```

The tactic (1) transforms the goal `(y * 0)+ y * (z * 0)= 0` into `y * (z * 0)= 0` and generates a new subgoal to prove the statement `y * 0 = 0`, which is the *instance* of the `forall x, x * 0 = 0` rewrite rule that has been used to perform the rewriting. On the other hand, tactic (2) performs the same rewriting on the current goal but generates a subgoal to prove `forall x, x * 0 = 0`.

When SSReflect `rewrite` fails on standard Coq licit rewrite

In a few cases, the SSReflect `rewrite` tactic fails rewriting some redexes which standard Coq successfully rewrites. There are two main cases:

- SSReflect never accepts to rewrite indeterminate patterns like:

  ```
  Lemma foo : forall x : unit, x = tt.
  ```

  SSReflect will however accept the $\eta\zeta$ expansion of this rule:

  ```
  Lemma fubar : forall x : unit, (let u := x in u) = tt.
  ```

- In standard Coq, suppose that we work in the following context:

  ```
  Variable g : nat -> nat.
  Definition f := g.
  ```

  then rewriting `H : forall x, f x = 0` in the goal `g 3 + g 3 = g 6` succeeds and transforms the goal into `0 + 0 = g 6`.

  This rewriting is not possible in SSReflect because there is no occurrence of the head symbol `f` of the rewrite rule in the goal.

Existential metavariables and rewriting

The `rewrite` tactic will not instantiate existing existential metavariables when matching a redex pattern.

If a rewriting rule generates a goal with new existential metavariables, these will be generalized as for `apply` (see page 25) and corresponding new goals will be generated. For example, consider the following script:

```
Lemma ex3 (x : 'I_2) y (le_1 : y < 1) (E : val x = y) : Some x = insub y.
rewrite insubT ?(leq_trans le_1)// => le_2.
```

Since `insubT` has the following type:

```
forall T P (sT : subType P) (x : T) (Px : P x), insub x = Some (Sub x Px)
```

and since the implicit argument corresponding to the `Px` abstraction is not supplied by the user, the resulting goal should be `Some x = Some (Sub y` $?_{Px}$`)`. Instead, SSReflect `rewrite` tactic generates the two following goals:

```
y < 2
forall Hyp0 : y < 2, Some x = Some (Sub y Hyp0)
```

The script closes the former with `?(leq_trans le_1)//`, then it introduces the new generalization naming it `le_2`.

```
x : 'I_2
y : nat
le_1 : y < 1
E : val x = y
le_2 : y < 2
============================
 Some x = Some (Sub y le_2)
```

As a temporary limitation, this behaviour is available only if the rewriting rule is stated using Leibniz equality (as oposed to setoid relations). It will be extended to other rewriting relations in the future.

7.3 Locking, unlocking

As program proofs tend to generate large goals, it is important to be able to control the partial evaluation performed by the simplification operations that are performed by the tactics. These evaluations can for example come from a `/=` simpl switch, or from rewrite steps which may expand large terms while performing conversion. We definitely want to avoid repeating large subterms of the goal in the proof script. We do this by "clamping down" selected function symbols in the goal, which prevents them from being considered in simplification or rewriting steps. This clamping is accomplished by using the occurrences switches (see section 4.2) together with "term tagging" operations.

SSReflect provides two levels of tagging.

The first one uses auxiliary definitions to introduce a provably equal copy of any term `t`, which is *not convertible* to `t`. The job is done by the following construction:

```
Lemma master_key : unit. Proof. exact tt. Qed.
Definition locked A := let: tt := master_key in fun x : A => x.
Lemma lock : forall A x, x = locked x :> A.
```

Note that the definition of master_key is explicitly opaque. The equation `t = locked t` given by the `lock` lemma can be used for selective rewriting, blocking on the fly the reduction in the term `t`. For example the script:

```
Require Import List.
Variable A : Type.

Fixpoint my_has (p : A -> bool)(l : list A){struct l} : bool:=
  match l with
    |nil => false
    |cons x l => p x || (my_has p l)
  end.


Goal forall a x y l, a x = true -> my_has a ( x :: y :: l) = true.
move=> a x y l Hax.
```

where `||` denotes the boolean disjunction, results in a goal `my_has a ( x :: y :: l)= true`. The tactic:

```
rewrite {2}[cons]lock /= -lock.
```

turns it into `a x || my_has a (y :: l)= true`. Let us now start by reducing the initial goal without blocking reduction. The script:

```
Goal forall a x y l, a x = true -> my_has a ( x :: y :: l) = true.
move=> a x y l Hax /=.
```

creates a goal `(a x)|| (a y)|| (my_has a l)= true`. Now the tactic:

```
rewrite {1}[orb]lock orbC -lock.
```

where orbC states the commutativity of `orb`, changes the goal into
`(a x)|| (my_has a l)|| (a y)= true`: only the arguments of the second disjunction where permuted.

It is sometimes desirable to globally prevent a definition from being expanded by simplification; this is done by adding `locked` in the definition.

For instance, the function fgraph_of_fun maps a function whose domain and codomain are finite types to a concrete representation of its (finite) graph. Whatever implementation of this transformation we may use, we want it to be hidden to simplifications and tactics, to avoid the collapse of the graph object:

```
Definition fgraph_of_fun :=
  locked
  (fun (d1 :finType) (d2 :eqType) (f : d1 -> d2) => Fgraph (size_maps f _)
    ).
```

We provide a special tactic `unlock` for unfolding such definitions while removing "locks", e.g., the tactic:

```
unlock ⟨occ-switch⟩fgraph_of_fun.
```

replaces the occurrence(s) of `cube` coded by the ⟨*occ-switch*⟩ with `(Hypermap cube_monic3)` in the goal.

We found that it was usually preferable to prevent the expansion of some functions by the partial evaluation switch "/=", unless this allowed the evaluation of a condition. This is possible thanks to an other mechanism of term tagging, resting on the following *Notation*:

```
Notation "'nosimpl' t" := (let: tt := tt in t).
```

The term `(nosimpl t)` simplifies to t *except* in a definition. More precisely, given:

```
Definition foo := (nosimpl bar).
```

the term `foo (or (foo t'))` will *not* be expanded by the *simpl* tactic unless it is in a forcing context (e.g., in `match foo t' with ... end`, `foo t'` will be reduced if this allows `match` to be reduced). Note that `nosimpl bar` is simply notation for a term that reduces to `bar`; hence `unfold foo` will replace `foo` by `bar`, and `fold foo` will replace `bar` by `foo`.

Warning The `nosimpl` trick only works if no reduction is apparent in `t`; in particular, the declaration:

```
Definition foo x := nosimpl (bar x).
```

will usually not work. Anyway, the common practice is to tag only the function, and to use the following definition, which blocks the reduction as expected:

```
Definition foo x := nosimpl bar x.
```

A standard example making this technique shine is the case of arithmetic operations. We define for instance:

```
Definition addn := nosimpl plus.
```

The operation `addn` behaves exactly like plus, except that `(addn (S n)m)` will not simplify spontaneously to `(S (addn n m))` (the two terms, however, are inter-convertible). In addition, the unfolding step:

```
rewrite /addn
```

will replace `addn` directly with `plus`, so the `nosimpl` form is essentially invisible.

7.4 Congruence

Because of the way matching interferes with type families parameters, the tactic:

```
apply: my_congr_property.
```

will generally fail to perform congruence simplification, even on rather simple cases. We therefore provide a more robust alternative in which the function is supplied:

`congr` [⟨*int*⟩] ⟨*term*⟩

This tactic:

- checks that the goal is a Leibniz equality
- matches both sides of this equality with "⟨*term*⟩ applied to some arguments", infering the right number of arguments from the goal and the type of ⟨*term*⟩. This may expand some definitions or fixpoints.
- generates the subgoals corresponding to pairwise equalities of the arguments present in the goal.

The goal can be a non dependent product `P -> Q`. In that case, the system asserts the equation `P = Q`, uses it to solve the goal, and calls the `congr` tactic on the remainig goal `P = Q`. For example, this allows for the idiom `congr (_ = _): H` to perform a transitivity step using `H`.

The optional ⟨*int*⟩ forces the number of arguments for which the tactic should generate equality proof obligations.

This tactic supports equalities between applications with dependent arguments. Anyway as in standard Coq, dependent arguments should have exactly the same parameters on both sides, and these parameters should appear as first arguments.

The following script:

```
Definition f n := match n with 0 => plus | S _ => mult end.
Definition g (n m : nat) := plus.

Goal forall x y, f 0 x y = g 1 1 x y.
by move=> x y; congr plus.
Qed.
```

shows that the `congr` tactic matches `plus` with `f 0` on the left hand side and `g 1 1` on the right hand side, and solves the goal.

The script:

```
Goal forall n m, m <= n -> S m + (S n - S m) = S n.
move=> n m Hnm; congr S; rewrite -/plus.
```

generates the subgoal `m + (S n - S m)= n`. The tactic `rewrite -/plus` folds back the expansion of `plus` which was necessary for matching both sides of the equality with an application of `S`.

Like most SSReflect arguments, ⟨*term*⟩ can contain wildcards. The script:

```
Goal forall x y, x + (y * (y + x - x)) = x * 1 + (y + 0) * y.
move=> x y; congr ( _ + (_ * _)).
```

generates three subgoals, respectively `x = x * 1`, `y = y + 0` and `y + x - x = y`.

8 Contextual patterns

The simple form of patterns used so far, ⟨*term*⟩*s* possibly containing wild cards, often require an additional ⟨*occ-switch*⟩ to be specified. While this may work pretty fine for small goals, the use of polymorphic functions and dependent types may lead to an invisible duplication of functions arguments. These copies usually end up in types hidden by the implicit arguments machinery or by user defined notations. In these situations computing the right occurrence numbers is very tedious because they must be counted on the goal as printed after setting the `Printing All` flag. Moreover the resulting script is not really informative for the reader, since it refers to occurrence numbers he cannot easily see.

Contextual patterns mitigate these issues allowing to specify occurrences according to the context they occur in.

8.1 Syntax

The following table summarises the full syntax of ⟨*c-pattern*⟩ and the corresponding subterm(s) identified by the pattern. In the third column we use s.m.r. for "the subterms matching the redex" specified in the second column.

⟨*c-pattern*⟩	redex	subterms affected
⟨*term*⟩	⟨*term*⟩	all occurrences of ⟨*term*⟩
⟨*ident*⟩ `in` ⟨*term*⟩	subterm of ⟨*term*⟩ selected by ⟨*ident*⟩	all the subterms identified by ⟨*ident*⟩ in all the occurrences of ⟨*term*⟩
⟨*term*⟩$_1$ `in` ⟨*ident*⟩ `in` ⟨*term*⟩$_2$	⟨*term*⟩$_1$	in all s.m.r. in all the subterms identified by ⟨*ident*⟩ in all the occurrences of ⟨*term*⟩$_2$
⟨*term*⟩$_1$ `as` ⟨*ident*⟩ `in` ⟨*term*⟩$_2$	⟨*term*⟩$_1$	in all the subterms identified by ⟨*ident*⟩ in all the occurrences of ⟨*term*⟩$_2$[⟨*term*⟩$_1$/⟨*ident*⟩]

The `rewrite` tactic supports two more patterns obtained prefixing the first two with `in`. The intended meaning is that the pattern identifies all subterms of the specified context. The `rewrite` tactic will infer a pattern for the redex looking at the rule used for rewriting.

⟨*r-pattern*⟩	redex	subterms affected
`in` ⟨*term*⟩	inferred from rule	in all s.m.r. in all occurrences of ⟨*term*⟩
`in` ⟨*ident*⟩ `in` ⟨*term*⟩	inferred from rule	in all s.m.r. in all the subterms identified by ⟨*ident*⟩ in all the occurrences of ⟨*term*⟩

The first ⟨*c-pattern*⟩ is the simplest form matching any context but selecting a specific redex and has been described in the previous sections. We have seen so far that the possibility of selecting a redex using a term with holes is already a powerful mean of redex selection. Similarity, any ⟨*term*⟩s provided by the user in the more complex forms of ⟨*c-pattern*⟩s presented in the tables above can contain holes.

For a quick glance at what can be expressed with the last ⟨*r-pattern*⟩ consider the goal `a = b` and the tactic

```
rewrite [in X in _ = X]rule.
```

It rewrites all occurrences of the left hand side of `rule` inside `b` only (`a`, and the hidden type of the equality, are ignored). Note that the variant `rewrite [X in _ = X]rule` would have rewritten `b` exactly (i.e., it would only work if `b` and the left hand side of `rule` were unifiable).

8.2 Matching contextual patterns

The ⟨*c-pattern*⟩s and ⟨*r-pattern*⟩s involving ⟨*term*⟩s with holes are matched against the goal in order to find a closed instantiation. This matching proceeds as follows:

⟨*c-pattern*⟩	instantiation order and place for $\langle term \rangle_i$ and redex
⟨*term*⟩	⟨*term*⟩ is matched against the goal, redex is unified with the instantiation of ⟨*term*⟩
⟨*ident*⟩ `in` ⟨*term*⟩	⟨*term*⟩ is matched against the goal, redex is unified with the subterm of the instantiation of ⟨*term*⟩ identified by ⟨*ident*⟩
$\langle term \rangle_1$ `in` ⟨*ident*⟩ `in` $\langle term \rangle_2$	$\langle term \rangle_2$ is matched against the goal, $\langle term \rangle_1$ is matched against the subterm of the instantiation of $\langle term \rangle_1$ identified by ⟨*ident*⟩, redex is unified with the instantiation of $\langle term \rangle_1$
$\langle term \rangle_1$ `as` ⟨*ident*⟩ `in` $\langle term \rangle_2$	$\langle term \rangle_2[\langle term \rangle_1/\langle ident \rangle]$ is matched against the goal, redex is unified with the instantiation of $\langle term \rangle_1$

In the following patterns, the redex is intended to be inferred from the rewriting rule.

⟨*r-pattern*⟩	instantiation order and place for $\langle term \rangle_i$ and redex
`in` ⟨*ident*⟩ `in` ⟨*term*⟩	⟨*term*⟩ is matched against the goal, the redex is matched against the subterm of the instantiation of ⟨*term*⟩ identified by ⟨*ident*⟩
`in` ⟨*term*⟩	⟨*term*⟩ is matched against the goal, redex is matched against the instantiation of ⟨*term*⟩

8.3 Examples

8.3.1 Contextual pattern in `set` and the `:` tactical

As already mentioned in section 4.2 the `set` tactic takes as an argument a term in open syntax. This term is interpreted as the simplest for of ⟨*c-pattern*⟩. To void confusion in the grammar, open syntax is supported only for the simplest form of patterns, while round parentheses are required around more complex patterns.

```
set t := (X in _ = X).
set t := (a + _ in X in _ = X).
```

Given the goal `a + b + 1 = b + (a + 1)` the first tactic captures `b + (a + 1)`, while the latter `a + 1`.

Since the user may define an infix notation for `in` the former tactic may result ambiguous. The disambiguation rule implemented is to prefer patterns over simple terms, but to interpret a pattern with double round parentheses as a simple term. For example the following tactic would capture any occurrence of the term '`a in A`'.

```
set t := ((a in A)).
```

Contextual pattern can also be used as arguments of the `:` tactical. For example:

```
elim: n (n in _ = n) (refl_equal n).
```

8.3.2 Contextual patterns in `rewrite`

As a more comprehensive example consider the following goal:

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

The tactic `rewrite [in f _ _]addSn` turns it into:

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

since the simplification rule `addSn` is applied only under the `f` symbol. Then we simplify also the first addition and expand `0` into `0+0`.

```
rewrite addSn -[X in _ = X]addn0.
```

obtaining:

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

Note that the right hand side of `addn0` is undetermined, but the rewrite pattern specifies the redex explicitly. The right hand side of `addn0` is unified with the term identified by `X`, `0` here.

The following pattern does not specify a redex, since it identifies an entire region, hence the rewriting rule has to be instantiated explicitly. Thus the tactic:

```
rewrite -{2}[in X in _ = X](addn0 0).
```

changes the goal as follows:

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

The following tactic is quite tricky:

```
rewrite [_.+1 in X in f _ X](addnC x.+1).
```

and the resulting goals is:

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

The explicit redex `_.+1` is important since its head constant `S` differs from the head constant inferred from `(addnC x.+1)` (that is `addn`, denoted `+` here). Moreover, the pattern `f _ X` is important to rule out the first occurrence of `(x + y).+1`. Last, only the subterms of `f _ X` identified by `X` are rewritten, thus the first argument of `f` is skipped too. Also note the pattern `_.+1` is interpreted in the context identified by `X`, thus it gets instantiated to `(y + x).+1` and not `(x + y).+1`.

The last rewrite pattern allows to specify exactly the shape of the term identified by `X`, that is thus unified with the left hand side of the rewriting rule.

```
rewrite [x.+1 + y as X in f X _]addnC.
```

The resulting goal is:

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

8.4 Patterns for recurrent contexts

The user can define shortcuts for recurrent contexts corresponding to the ⟨*ident*⟩`in`⟨*term*⟩ part. The notational scope identified with `%pattern` provides a special notation '`(X in t)`' the user must adopt to define contexts shortcuts.

The following example is taken from `ssreflect.v` where the `LHS` and `RHS` shortcuts are defined.

```
Notation RHS := (X in _ = X)%pattern.
Notation LHS := (X in X = _)%pattern.
```

Shortcuts defined this way can be freely used in place of the trailing ⟨*ident*⟩`in`⟨*term*⟩ part of any contextual pattern. Some examples follow:

```
set rhs := RHS.
rewrite [in RHS]rule.
case: (a + _ in RHS).
```

9 Views and reflection

The bookkeeping facilities presented in section 5 are crafted to ease simultaneous introduction/generalisation of facts and casing, naming ... operations. It also a common practise to make a stack operation immediately followed by an *interpretation* of the fact being pushed, that is to say to apply a lemma to this fact before passing it to a tactic for decomposition, application and so on.

SSReflect provides a convenient, unified syntax to combine these interpretation operations with the proof stack operations. This *view mechanism* relies on the combination of the `/` view switch with bookkeeping tactics and tacticals.

9.1 Interpreting eliminations

The view syntax combined with the `elim` tactic specifies an elimination scheme to be used instead of the default, generated, one. Hence the SSReflect tactic:

```
elim/V.
```

corresponds to the standard Coq tactic:

```
intro top; elim top using V; clear top.
```

where `top` is a fresh name and `V` any second-order lemma.

Since an elimination view supports the two bookkeeping tacticals of discharge and introduction (see section 5), the SSReflect tactic:

```
elim/V: x => y.
```

corresponds to the standard Coq tactic:

```
elim x using V; clear x; intro y.
```

where `x` is a variable in the context, `y` a fresh name and `V` any second order lemma; SSReflect relaxes the syntactic restrictions of the Coq `elim`. The first pattern following `:` can be a `_` wildcard if the conclusion of the view `V` specifies a pattern for its last argument (e.g., if `V` is a functional induction lemma generated by the `Function` command).

The elimination view mechanism is compatible with the equation name generation (see section 5.5).

The following script illustrate a toy example of this feature. Let us define a function adding an element at the end of a list:

```
Require Import List.

Variable d : Type.

Fixpoint add_last(s : list d) (z : d) {struct s} : list d :=
  match s with
  | nil => z :: nil
  | cons x s' => cons x (add_last s' z)
  end.
```

One can define an alternative, reversed, induction principle on inductively defined `list`s, by proving the following lemma:

```
Lemma last_ind_list : forall (P : list d -> Type),
P nil ->
(forall (s : list d) (x : d), P s -> P (add_last s x)) -> forall s : list
    d, P s.
```

Then the combination of elimination views with equation names result in a concise syntax for reasoning inductively using the user defined elimination scheme. The script:

```
Goal forall (x : d)(l : list d), l = l.
move=> x l.
elim/last_ind_list E : l=> [| u v]; last first.
```

generates two subgoals: the first one to prove `nil = nil` in a context featuring `E : l = nil` and the second to prove `add_last u v = add_last u v`, in a context containing `E : l = add_last u v`.

User provided eliminators (potentially generated with the `Function` Coq's command) can be combined with the type family switches described in section 5.6. Consider an eliminator `foo_ind` of type:

`foo_ind : forall ...., forall x : T, P` $\mathtt{p}_1$ `...` $\mathtt{p}_m$

and consider the tactic

`elim/foo_ind:` $\mathtt{e}_1$ `... /` $\mathtt{e}_n$

The `elim` tactic distinguishes two cases:

truncated eliminator when `x` does not occur in `P` $\mathtt{p}_1$`...` $\mathtt{p}_m$ and the type of $\mathtt{e}_n$ unifies with `T` and $\mathtt{e}_n$ is not `_`. In that case, $\mathtt{e}_n$ is passed to the eliminator as the last argument (`x` in `foo_ind`) and $\mathtt{e}_{n-1}$`...` $\mathtt{e}_1$ are used as patterns to select in the goal the occurrences that will be bound by the predicate `P`, thus the sub-term of the goal matched by $\mathtt{e}_{n-1}$ must be unifiable with $\mathtt{p}_m$, the one matched by $\mathtt{e}_{n-2}$ with $\mathtt{p}_{m-1}$ and so on.

regular eliminator in all the other cases. Here the term matched by $\mathtt{e}_n$ must be unifiable with $\mathtt{p}_m$, the one matched by $\mathtt{e}_{n-1}$ must be unifiable with $\mathtt{p}_{m-1}$ and so on. Note that standard eliminators have the shape `...forall x, P ... x`, thus $\mathtt{e}_n$ is the pattern identifying the eliminated term, as expected.

As explained in section 5.6, the initial prefix of $\mathtt{e}_i$ can be omitted.

Here an example of a regular, but non trivial, eliminator:

```
Function plus (m n : nat) {struct n} : nat :=
   match n with 0 => m | S p => S (plus m p) end.
```

The type of `plus_ind` is

```
plus_ind : forall (m : nat) (P : nat -> nat -> Prop),
  (forall n : nat, n = 0 -> P 0 m) ->
  (forall n p : nat, n = p.+1 -> P p (plus m p) -> P p.+1 (plus m p).+1) ->
    forall n : nat, P n (plus m n)
```

Consider the following goal

```
Lemma exF x y z: plus (plus x y) z = plus x (plus y z).
```

The following tactics are all valid and perform the same elimination on that goal.

```
elim/plus_ind: z / (plus _ z).
elim/plus_ind: {z}(plus _ z).
elim/plus_ind: {z}_.
elim/plus_ind: z / _.
```

In the two latter examples, being the user provided pattern a wildcard, the pattern inferred from the type of the eliminator is used instead. For both cases it is `(plus _ _)` and matches the subterm `plus (plus x y) z` thus instantiating the latter `_` with `z`. Note that the tactic `elim/plus_ind: y / _` would have resulted in an error, since `y` and `z` do no unify but the type of the eliminator requires the second argument of `P` to be the same as the second argument of `plus` in the second argument of `P`.

Here an example of a truncated eliminator. Consider the goal

```
p : nat_eqType
n : nat
n_gt0 : 0 < n
pr_p : prime p
=================
p %| \prod_(i <- prime_decomp n | i \in prime_decomp n) i.1 ^ i.2 ->
  exists2 x : nat * nat, x \in prime_decomp n & p = x.1
```

and the tactic

```
elim/big_prop: _ => [| u v IHu IHv | [q e] /=].
```

where the type of the eliminator is

```
big_prop: forall (R : Type) (Pb : R -> Type) (idx : R) (op1 : R -> R -> R),
  Pb idx ->
  (forall x y : R, Pb x -> Pb y -> Pb (op1 x y)) ->
  forall (I : Type) (r : seq I) (P : pred I) (F : I -> R),
  (forall i : I, P i -> Pb (F i)) ->
    Pb (\big[op1/idx]_(i <- r | P i) F i)
```

Since the pattern for the argument of `Pb` is not specified, the inferred one is used instead: `(\big[_/_]_(i <- _ | _ i)_ i)`, and after the introductions, the following goals are generated.

```
subgoal 1 is:
 p %| 1 -> exists2 x : nat * nat, x \in prime_decomp n & p = x.1
subgoal 2 is:
 p %| u * v -> exists2 x : nat * nat, x \in prime_decomp n & p = x.1
subgoal 3 is:
 (q, e) \in prime_decomp n -> p %| q ^ e ->
   exists2 x : nat * nat, x \in prime_decomp n & p = x.1
```

Note that the pattern matching algorithm instantiated all the variables occurring in the pattern.

9.2 Interpreting assumptions

Interpreting an assumption in the context of a proof is applying it a correspondence lemma before generalising, and/or decomposing it. For instance, with the extensive use of boolean reflection (see section 8.4), it is quite frequent to need to decompose the logical interpretation of (the boolean expression of) a fact, rather than the fact itself. This can be achieved by a combination of `move : _ => _` switches, like in the following script, where `||` is a standard Coq notation for the boolean disjunction:

```
Variables P Q : bool -> Prop.
Hypothesis P2Q : forall a b, P (a || b) -> Q a.

Goal forall a, P (a || a) -> True.
move=> a HPa; move: {HPa}(P2Q _ _ HPa) => HQa.
```

which transforms the hypothesis `HPn : P n` which has been introduced from the initial statement into `HQn : Q n`. This operation is so common that the tactic shell has specific syntax for it. The following scripts:

```
Goal forall a, P (a || a) -> True.
move=> a HPa; move/P2Q: HPa => HQa.
```

or more directly:

```
Goal forall a, P (a || a) -> True.
move=> a; move/P2Q=> HQa.
```

are equivalent to the former one. The former script shows how to interpret a fact (already in the context), thanks to the discharge tactical (see section 5.3) and the latter, how to interpret the top assumption of a goal. Note that the number of wildcards to be inserted to find the correct application of the view lemma to the hypothesis has been automatically inferred.

The view mechanism is compatible with the `case` tactic and with the equation name generation mechanism (see section 5.5):

```
Variables P Q: bool -> Prop.
Hypothesis Q2P : forall a b, Q (a || b) -> P a \/ P b.

Goal forall a b, Q (a || b) -> True.
move=> a b; case/Q2P=> [HPa | HPb].
```

creates two new subgoals whose contexts no more contain `HQ : Q (a || b)` but respectively `HPa : P a` and `HPb : P b`. This view tactic performs:

```
move=> a b HQ; case: {HQ}(Q2P _ _ HQ) => [HPa | HPb].
```

The term on the right of the `/` view switch is called a *view lemma*. Any SSReflect term coercing to a product type can be used as a view lemma.

The examples we have given so far explicitly provide the direction of the translation to be performed. In fact, view lemmas need not to be oriented. The view mechanism is able to detect which application is relevant for the current goal. For instance, the script:

```
Variables P Q: bool -> Prop.
Hypothesis PQequiv : forall a b, P (a || b) <-> Q a.

Goal forall a b, P (a || b) -> True.
move=> a b; move/PQequiv=> HQab.
```

has the same behaviour as the first example above.

The view mechanism can insert automatically a *view hint* to transform the double implication into the expected simple implication. The last script is in fact equivalent to:

```
Goal forall a b, P (a || b) -> True.
move=> a b; move/(iffLR (PQequiv _ _)).
```

where:

```
Lemma iffLR : forall P Q, (P <-> Q) -> P -> Q.
```

Specialising assumptions

The special case when the *head symbol* of the view lemma is a wildcard is used to interpret an assumption by *specialising* it. The view mechanism hence offers the possibility to apply a higher-order assumption to some given arguments.

For example, the script:

```
Goal forall z, (forall x y, x + y = z -> z = x) -> z = 0.
move=> z; move/(_ 0 z).
```

changes the goal into:

```
(0 + z = z -> z = 0) -> z = 0
```

9.3 Interpreting goals

In a similar way, it is also often convenient to interpret a goal by changing it into an equivalent proposition. The view mechanism of SSREFLECT has a special syntax `apply/` for combining simultaneous goal interpretation operations and bookkeeping steps in a single tactic.

With the hypotheses of section 8.2, the following script, where `~~` denotes the boolean negation:

```
Goal forall a, P ((~~ a) || a).
move=> a; apply/PQequiv.
```

transforms the goal into `Q (~~ a)`, and is equivalent to:

```
Goal forall a, P ((~~ a) || a).
move=> a; apply: (iffRL (PQequiv _ _)).
```

where `iffLR` is the analogous of `iffRL` for the converse implication.

Any SSReflect term whose type coerces to a double implication can be used as a view for goal interpretation.

Note that the goal interpretation view mechanism supports both `apply` and `exact` tactics. As expected, a goal interpretation view command `exact`/*term* should solve the current goal or it will fail.

Warning Goal interpretation view tactics are *not* compatible with the bookkeeping tactical `=>` since this would be redundant with the `apply: V=> _` construction.

9.4 Boolean reflection

In the Calculus of Inductive Construction, there is an obvious distinction between logical propositions and boolean values. On the one hand, logical propositions are objects of *sort* `Prop` which is the carrier of intuitionistic reasoning. Logical connectives in `Prop` are *types*, which give precise information on the structure of their proofs; this information is automatically exploited by Coq tactics. For example, Coq knows that a proof of `A \/ B` is either a proof of `A` or a proof of `B`. The tactics `left` and `right` change the goal `A \/ B` to `A` and `B`, respectively; dually, the tactic `case` reduces the goal `A \/ B => G` to two subgoals `A => G` and `B => G`.

On the other hand, `bool` is an inductive *datatype* with two constructors `true` and `false`. Logical connectives on `bool` are *computable functions*, defined by their truth tables, using case analysis:

```
Definition (b1 || b2) := if b1 then true else b2.
```

Properties of such connectives are also established using case analysis: the tactic `by case: b` solves the goal

```
b || ~~ b = true
```

by replacing `b` first by `true` and then by `false`; in either case, the resulting subgoal reduces by computation to the trivial `true = true`.

Thus, `Prop` and `bool` are truly complementary: the former supports robust natural deduction, the latter allows brute-force evaluation. SSReflect supplies a generic mechanism to have the best of the two worlds and move freely from a propositional version of a decidable predicate to its boolean version.

First, booleans are injected into propositions using the coercion mechanism:

```
Coercion is_true (b : bool) := b = true.
```

This allows any boolean formula `b` to be used in a context where Coq would expect a proposition, e.g., after `Lemma ... :`. It is then interpreted as `(is_true b)`, i.e., the proposition `b = true`. Coercions are elided by the pretty-printer, so they are essentially transparent to the user.

9.5 The `reflect` predicate

To get all the benefits of the boolean reflection, it is in fact convenient to introduce the following inductive predicate `reflect` to relate propositions and booleans:

```
Inductive reflect (P: Prop): bool -> Type :=
  | Reflect_true: P => reflect P true
  | Reflect_false: ~P => reflect P false.
```

The statement `(reflect P b)` asserts that `(is_true b)` and `P` are logically equivalent propositions.

For instance, the following lemma:

```
Lemma andP: forall b1 b2, reflect (b1 /\ b2) (b1 && b2).
```

relates the boolean conjunction `&&` to the logical one `/\`. Note that in `andP`, `b1` and `b2` are two boolean variables and the proposition `b1 /\ b2` hides two coercions. The conjunction of `b1` and `b2` can then be viewed as `b1 /\ b2` or as `b1 && b2`.

Expressing logical equivalences through this family of inductive types makes possible to take benefit from *rewritable equations* associated to the case analysis of COQ's inductive types.

Since the standard equivalence predicate is defined in Coq as:

```
Definition iff (A B:Prop) := (A -> B) /\ (B -> A).
```

where `/\` is a notation for `and`:

```
Inductive and (A B:Prop) : Prop :=
 conj : A -> B -> and A B
```

This make case analysis very different according to the way an equivalence property has been defined.

For instance, if we have proved the lemma:

```
Lemma andE: forall b1 b2, (b1 /\ b2) <-> (b1 && b2).
```

let us compare the respective behaviours of `andE` and `andP` on a goal:

```
Goal forall b1 b2, if (b1 && b2) then b1 else ~~(b1||b2).
```

The command:

```
move=> b1 b2; case (@andE b1 b2).
```

generates a single subgoal:

```
(b1 && b2 -> b1 /\ b2) -> (b1 /\ b2 -> b1 && b2) ->
                if b1 && b2 then b1 else ~~ (b1 || b2)
```

while the command:

```
move=> b1 b2; case (@andP b1 b2).
```

generates two subgoals, respectively `b1 /\ b2 -> b1` and `~ (b1 /\ b2)-> ~~ (b1 || b2)`.

Expressing reflection relation through the `reflect` predicate is hence a very convenient way to deal with classical reasoning, by case analysis. Using the `reflect` predicate allows moreover to program rich specifications inside its two constructors, which will be automatically taken into account during destruction. This formalisation style gives far more efficient specifications than quantified (double) implications.

A naming convention in SSReflect is to postfix the name of view lemmas with `P`. For example, `orP` relates `||` and `\/`, `negP` relates `~~` and `~`.

The view mechanism is compatible with `reflect` predicates.

For example, the script

```
Goal forall a b : bool, a -> b -> a /\ b.
move=> a b Ha Hb; apply/andP.
```

changes the goal `a /\ b` to `a && b` (see section 8.3).

Conversely, the script

```
Goal forall a b : bool, a /\ b -> a.
move=> a b; move/andP.
```

changes the goal `a /\ b -> a` into `a && b -> a` (see section 8.2).

The same tactics can also be used to perform the converse operation, changing a boolean conjunction into a logical one. The view mechanism guesses the direction of the transformation to be used i.e., the constructor of the `reflect` predicate which should be chosen.

9.6 General mechanism for interpreting goals and assumptions

Specialising assumptions

The SSReflect tactic:

`move/(_` $\langle term\rangle_1$ `...` $\langle term\rangle_n$`)`

is equivalent to the tactic:

`intro top; generalize (top` $\langle term\rangle_1$ `...` $\langle term\rangle_n$`); clear top.`

where `top` is a fresh name for introducing the top assumption of the current goal.

Interpreting assumptions

The general form of an assumption view tactic is:

[`move`|`case`]`/`$\langle term\rangle_0$`.`

The term $\langle term\rangle_0$, called the *view lemma* can be:

- a (term coercible to a) function;

- a (possibly quantified) implication;
- a (possibly quantified) double implication;
- a (possibly quantified) instance of the `reflect` predicate (see section 8.5).

Let `top` be the top assumption in the goal.
There are three steps in the behaviour of an assumption view tactic:

- It first introduces `top`.
- If the type of $\langle term \rangle_0$ is neither a double implication nor an instance of the `reflect` predicate, then the tactic automatically generalises a term of the form:

  ($\langle term \rangle_0$ $\langle term \rangle_1$ ... $\langle term \rangle_n$)

  where the terms $\langle term \rangle_1$ ... $\langle term \rangle_n$ instantiate the possible quantified variables of $\langle term \rangle_0$, in order for ($\langle term \rangle_0$ $\langle term \rangle_1$ ... $\langle term \rangle_n$`top`) to be well typed.
- If the type of $\langle term \rangle_0$ is an equivalence, or an instance of the `reflect` predicate, it generalises a term of the form:

  ($\langle term \rangle_{vh}$ ($\langle term \rangle_0$ $\langle term \rangle_1$ ... $\langle term \rangle_n$))

  where the term $\langle term \rangle_{vh}$ inserted is called an *assumption interpretation view hint*.
- It finally clears `top`.

For a `case/`$\langle term \rangle_0$ tactic, the generalisation step is replaced by a case analysis step.

View hints are declared by the user (see section 8.8) and are stored in the `Hint View` database. The proof engine automatically detects from the shape of the top assumption `top` and of the view lemma $\langle term \rangle_0$ provided to the tactic the appropriate view hint in the database to be inserted.

If $\langle term \rangle_0$ is a double implication, then the view hint `A` will be one of the defined view hints for implication. These hints are by default the ones present in the file `ssreflect.v`:

```
Lemma iffLR : forall P Q, (P <-> Q) -> P -> Q.
```

which transforms a double implication into the left-to-right one, or:

```
Lemma iffRL : forall P Q, (P <-> Q) -> Q -> P.
```

which produces the converse implication. In both cases, the two first `Prop` arguments are implicit.

If $\langle term \rangle_0$ is an instance of the `reflect` predicate, then `A` will be one of the defined view hints for the `reflect` predicate, which are by default the ones present in the file `ssrbool.v`. These hints are not only used for choosing the appropriate direction of the translation, but they also allow complex transformation, involving negations. For instance the hint:

```
Lemma introN : forall (P : Prop) (b : bool), reflect P b -> ~ P -> ~~ b.
```

makes the following script:

```
Goal forall a b : bool, a -> b -> ~~ (a && b).
move=> a b Ha Hb. apply/andP.
```

transforms the goal into `~ (a / b)`. In fact[13] this last script does not exactly use the hint `introN`, but the more general hint:

```
Lemma introNTF : forall (P : Prop) (b c : bool),
    reflect P b -> (if c then ~ P else P) -> ~~ b = c
```

The lemma `introN` is an instantiation of `introNF` using `c := true`.

Note that views, being part of ⟨*i-pattern*⟩, can be used to interpret assertions too. For example the following script asserts `a && b` but actually used its propositional interpretation.

```
Lemma test (a b : bool) (pab : b && a) : b.
have /andP [pa ->] : (a && b) by rewrite andbC.
```

Interpreting goals

A goal interpretation view tactic of the form:

`apply/`$\langle term \rangle_0$`.`

applied to a goal `top` is interpreted in the following way:

- If the type of $\langle term \rangle_0$ is not an instance of the `reflect` predicate, nor an equivalence, then the term $\langle term \rangle_0$ is applied to the current goal `top`, possibly inserting implicit arguments.
- If the type of $\langle term \rangle_0$ is an instance of the `reflect` predicate or an equivalence, then a *goal interpretation view hint* can possibly be inserted, which corresponds to the application of a term `(`$\langle term \rangle_{vh}$`(`$\langle term \rangle_0$`_ ... _))` to the current goal, possibly inserting implicit arguments.

Like assumption interpretation view hints, goal interpretation ones are user defined lemmas stored (see section 8.8) in the `Hint View` database bridging the possible gap between the type of $\langle term \rangle_0$ and the type of the goal.

9.7 Interpreting equivalences

Equivalent boolean propositions are simply *equal* boolean terms. A special construction helps the user to prove boolean equalities by considering them as logical double implications (between their coerced versions), while performing at the same time logical operations on both sides.

[13]The current state of the proof shall be displayed by the `Show Proof` command of Coq proof mode.

The syntax of double views is:

apply/⟨*term*⟩$_l$/⟨*term*⟩$_r$.

The term ⟨*term*⟩$_l$ is the view lemma applied to the left hand side of the equality, ⟨*term*⟩$_r$ is the one applied to the right hand side.

In this context, the identity view:

```
Lemma idP : reflect b1 b1.
```

is useful, for example the tactic:

```
apply/idP/idP.
```

transforms the goal `~~ (b1 || b2)= b3` into two subgoals, respectively `~~ (b1 || b2)-> b3` and
`b3 -> ~~ (b1 || b2)`.

The same goal can be decomposed in several ways, and the user may choose the most convenient interpretation. For instance, the tactic:

```
apply/norP/idP.
```

applied on the same goal `~~ (b1 || b2)= b3` generates the subgoals `~~ b1 /\ ~~ b2 -> b3` and
`b3 -> ~~ b1 /\ ~~ b2`.

9.8 Declaring new Hint Views

The user can declare his own hints for the view mechanism, following the syntax used in `ssrbool.v`:

Hint View for ⟨*tactic*⟩/ ⟨*ident*⟩[|⟨*num*⟩].

where ⟨*tactic*⟩ ∈ {`move, apply`}, ⟨*ident*⟩ is the name of the lemma to be declared as a hint, and ⟨*num*⟩ a natural number. If `move` is used as ⟨*tactic*⟩, the hint is declared for assumption interpretation tactics, `apply` declares hints for goal interpretations. Goal interpretation view hints are declared for both simple views and left hand side views. The optional natural number ⟨*num*⟩ is the number of implicit arguments to be considered for the declared hint view lemma `name_of_the_lemma`.

The command:

Hint View for apply// ⟨*ident*⟩[|⟨*num*⟩].

with a double slash `//`, declares hint views for right hand sides of double views.
See the files `ssreflect.v` and `ssrbool.v` for examples.

9.9 Multiple views

The hypotheses and the goal can be interpreted applying multiple views in sequence. Both `move` and `apply` can be followed by an arbitrary number of /⟨*term*⟩$_i$. The main difference between the following two tactics

```
apply/v1/v2/v3.
apply/v1; apply/v2; apply/v3.
```

is that the former applies all the views to the principal goal. Applying a view with hypotheses generates new goals, and the second line would apply the view `v2` to all the goals generated by `apply/v1`. Note that the NO-OP intro pattern `-` can be used to separate two views, making the two following examples equivalent:

```
move=> /v1; move=> /v2.
move=> /v1-/v2.
```

The tactic `move` can be used together with the `in` tactical to pass a given hypothesis to a lemma. For example, if `P2Q : P -> Q` and `Q2R : Q -> R`, the following tactic turns the hypothesis `p : P` into `P : R`.

```
move/P2Q/Q2R in p.
```

If the list of view is of length two, `Hint View`s for interpreting equivalences are indeed taken into account, otherwise only single `Hint View`s are used.

10 SSReflect searching tool

SSReflect proposes an extension of the `Search` command of standard Coq. Its syntax is:

`Search` [⟨*pattern*⟩] [[-][⟨*string*⟩[%⟨*key*⟩] | ⟨*pattern*⟩]]* [`in` [[-]⟨*name*⟩]$^+$].

where ⟨*name*⟩ is the name of a open module. This command searches returns the list of lemmas:

- whose *conclusion* contains a subterm matching the optional first ⟨*pattern*⟩. A `-` reverses the test, producing the list of lemmas whose conclusion does not contain any subterm matching the pattern;
- whose name contains the given strings. A `-` prefix reverses the test, producing the list of lemmas whose name does not contain the string. A string that contains symbols or is followed by a scope ⟨*key*⟩, is interpreted as the constant whose notation involves that string (e.g., `+` for `addn`), if this is unambiguous; otherwise the diagnostic includes the output of the `Locate` standard vernacular command.
- whose statement, including assumptions and types contains a subterm matching the next patterns. If a pattern is prefixed by `-`, the test is reversed;

- contained in the given list of modules, except the ones in the given modules prefixed by a `-`.

Note that:

- As for regular terms, patterns can feature scope indications. For instance, the command:

  ```
  Search _ (_ + _)%N.
  ```

  lists all the lemmas whose statement (conclusion or hypotheses) involve an application of the binary operation denoted by the infix `+` symbol in the `N` scope (which is SSReflect scope for natural numbers).

- Patterns with holes should be surrounded by parentheses.

- Search always volunteers the expansion of the notation, avoiding the need to execute Locate independently. Moreover, a string fragment looks for any notation that contains fragment as a substring. If the `ssrbool` library is imported, the command:

  ```
  Search "~~".
  ```

  answers :

  ```
  "~~" is part of notation (~~ _)
  In bool_scope, (~~ b) denotes negb b
  negbT forall b : bool, b = false -> ~~ b
  contra forall c b : bool, (c -> b) -> ~~ b -> ~~ c
  introN forall (P : Prop) (b : bool), reflect P b -> ~ P -> ~~ b
  ```

- A diagnostic is issued if there are different matching notations; it is an error if all matches are partial.

- Similarly, a diagnostic warns about multiple interpretations, and signals an error if there is no default one.

- The command `Search in M.` is a way of obtaining the complete signature of the module `M`.

- Strings and pattern indications can be interleaved, but the first indication has a special status if it is a pattern, and only filters the conclusion of lemmas:

  - The command :

    ```
    Search (_ =1 _) "bij".
    ```

    lists all the lemmas whose conclusion features a '`=1`' and whose name contains the string `bij`.

– The command :

```
Search "bij" (_ =1 _).
```

lists all the lemmas whose statement, including hypotheses, features a '`=1`' and whose name contains the string `bij`.

11 Using the libraries

This section contains a short description of the libraries present in the distribution of SSReflect. For further information, please refer to the comments included in the `.v` source files of the distributed libraries.

11.1 Some general design choices

Use of CoInductive types

Coinductive types declared throughout the libraries should be considered as data structures enjoying case analysis but not induction. This design choice is meant to stop the user from doing meaningless induction rather than to represent infinite objects.

Boolean propositions, support for small scale reflection

The `ssreflect` library provides basic support for the view mechanism, together with some technical lemmas and definitions, dedicated to the internal mechanism of some SSReflect tactics, and to the reduction tags (`nosimpl`, `lock`, see section 7.3). This `ssreflect.v` file being devoted to technical infrastructure, like `ssreflect.ml`, it does not contain much insight on the constructions provided.

The `ssrbool` library extends the possibilities of the view mechanism by providing the support needed for small scale reflection.

The main ingredients of this support are:

- The `is_true: bool >-> Prop` coercion;
- The definition of the `reflect` predicate (see section 8.5), together with the:

  ```
  Coercion elimT : reflect >-> Funclass
  ```

  which makes possible to apply directly a reflection lemma to a boolean assertion;
- A bunch of reflection lemmas, which directly reflect boolean connectives into their logical counterparts (like `andP`), or propagate negations (like `norP`);
- Predefined view hints, which help the reflection mechanism handle complex boolean statements to be reflected into logical ones (like `introNTF`).

The `ssrbool` library also contains several toolkits for handling boolean predicates that become pervasive in a boolean small scale reflection setting, like :

- Properties of boolean connectives, including the various commutativity, associativity, distributivity (rewritable) properties and a `bool_congr` heuristic for weeding out common terms from boolean equalities;

- Iterated boolean operations, comprizing some convenient n-ary "list-style" notations for these kind of predicates, in the form:

  `[op arg1, arg2, ... last_separator last_arg]`

  See the comments in `ssrbool.v` for more details.

- Support for applicative and collective predicate, where collective predicates deserve infix notation like `(x \ in p)`;

- Properties and predicates on boolean relations and the localized

For more details, please refer to the comments in `ssrbool`.

Canonical Structures

The use of `Canonical Structure`s is pervasive in SSReflect libraries. This provides in particular a mechanism of proof inference, used as a Prolog-like proof inference engine as often as possible.

For instance, once the following irrelevance result on `eqType` structures (see `eqtype.v`) is proved:

```
Theorem eq_irrelevance (T : eqType) (x y : T) (e1 e2 : x = y) : e1 = e2.
```

it is instantiated for booleans (see again `eqtype.v`), and proved in this way:

```
Lemma bool_irrelevance : forall (x y : bool) (E E' : x = y), E = E'.
Proof. exact: eq_irrelevance. Qed.
```

Notice that the `eqType` structure of booleans, which is required to apply the `eq_irrelevance` theorem, is automatically inferred by the system from the `bool` type of `x` and `y`.

Notice also that the tactic:

```
exact eq_irrelevance.
```

(without the ': tactical) would fail closing the goal because the unification algorithm used by the standard `apply` and `exact` tactics does not take `Canonical Structure` equations into account. On the other hand, the standard `refine` tactic knows about these equations. This is why the combinations `apply:` and `exact:` make the use of `Canonical Structure`s really tractable, when this feature of standard Coq is not usable in practise with standard 8.2 Coq.

In fact, Coq's `Canonical Structure` mechanism precomputes some projection equations that are later used as hints during unification. Canonical structures are hence inferred automatically when a statement explicitly involves projections *values*. In contrast, canonical structure inference cannot directly unify the structure *type* with that of its projections, despite the fact that `Coercion`s may give the illusion that this happens.

For example, the following script will be accepted

```
Require Import ssreflect eqtype.
Variables A B C : eqType.
Hypothesis BtimesCisA : (B * C = A)%type.
```

because type-checking will insert the `Equality.sort` coercion projection ao `A` in the right-hand side of the equation. However,

```
Hypothesis AisBtimesC : (A = B * C)%type.
```

raises the error message: `Error: The term "(B * C)%type" has type "Type" while it is expected to have type "eqType"` even though the `eqtype` library contains a canonical construction of the eqType structure of the cartesian product of two eqType structures: unlike canonical projections, coercions only work one way.

This illustrates the need for a way of directly looking up a canonical structure. This operation is supported, with the general syntax:

[*struct_type* `of` *carrier_type* [`for` *t*]]

This computes the canonical structure of type *struct_type* declared up to conversion on the type *carrier_type*. The structure infered should be convertible to the optional argument *t*.

Going back to the previous example:

```
Hypothesis AisBtimesC : A = [eqType of (B * C)%type].
```

is accepted by the system.

Some types are defined by requiring a structure on their parameter. For instance, the type `polynomial` of polynomials (in `poly.v`) has one parameter, which should be equipped with a ring structure. In that case, a curly bracket notation is provided to allow Coq to infer said structure rather that having to supply it explicitly. This notation should always be preferred to the type itself. Hence one should always use `{poly R}`, rather than `(polynomial R_ringType)`, when `R_ringType` would be the ring structure with carrier `R`. If this `R_ringType`structure is canonical, it will automatically be inferred in `{poly R}`.

11.2 SSReflect proving guidelines

Here are a few tips to write proofs using SSReflect.

Proof formatting guidelines

- **Loaded libraries.** The order matters! For instance, `bigop` should be loaded before `ssralg`, to make the notations work.

- **Delimiters.** A space should always follow a delimiter symbol, and spaces should surround operator symbols. Similarly, spaces should always surround a type casting colon.

- **How to write pairs.** A tuple is parenthesised and the commas therein (delimiters) are each followed by a space: `(1, 2)`.

- **How to write sequences.** Write `x :: l` with spaces around the `::` (since :: is an infix operator, hence surrounded by spaces) and `[:: x1; x2; x3]` or `[:: x1, x2 & tl]` (since `,` is a delimiter, hence followed by a space).

- **How to write auto-simplifying functions.** Autosimplifying functions should be defined using the bracket notation provided by the `ssrfun` library. The expression `[fun x y => f x y]` denotes the auto-simplifying version of the binary function `f`.

- **How to write auto-simplifying predicates.** Following the bracket notation for auto-simplifying functions, the `ssrbool` library provides a comprehension-style notation for autosimplifying boolean predicates, in the form: `[type var separator expr]`. For instance `[pred x | x == y]` is the auto-simplifying boolean comparison to `y`.

- **How to write composed predicates.** The `ssrbool` library provides sequence-like notations for the iteration of a right associative operator on a list of predicates (boolean or not). For instance `[/\ P1, P2 & P3]` (resp. `[\/ P1, P2 | P3]`) denotes `P1 /\ P2 /\ P3` (resp. `P1 \/ P2 \/ P3`). See comments and reserved notations in `ssrbool.v` for the complete list of available notations.

- **How to write operators symbols.** For sake of readability, operators symbols should be separated from their arguments by spaces, like in `x * y` or `a (+) b`. Perhaps more importantly, this should prevent Coq from confusing them with multi-characters notations.

- **How to write a partial application wrt to the *second* argument.** The `fun` library provides a convenient "placeholder" notation for such abstractions:

  ```
  Notation "f ^~ y" := (fun x => f x y)
  ```

- **How to cast an equality.** To force the application of an existing coercion on the left hand side of a Leibniz equality, and hence in the type provided to `eq`, use the `:>` cast. For instance `(true = 1)` is not type-checked, but `(true = 1 :> nat)` is[14]. Note that this feature is provided by the standard Coq.

- **How to state standard properties on operators.** Use the predicates defined in the **ssrfun** library : `left_inverse`, `right_distributive`, `commutative`, etc to normalize these statements.

[14]The `ssrnat` library defines such a coercion from `bool` to `nat`.

Layout of proofs

- **Width of the page.** The library has been developed using a 80 characters wide page. Maintaining this convention in your developments improves the readability of scripts. This is specially relevant in SSReflect scripts, where tactics are usually chained in longer lines than in most standard Coq scripts.

- **Starting a proof.** Start each proof with `Proof`. If the scripts of the proof is a single line of tactics, then this line should start with `Proof.` and finish with `Qed.` If it doesn't the first line of the script should only contain `Proof.` and the last line should only contain `Qed.`.

- **Lines of tactics in a proof.** Tactics should be chained on the whole line using the semi-colon (or chained rewritings for instance), but a block between two points should never take more than one line. This often helps lines having a semantical unity.

  Start a new line each time you start a new subgoal proof, or use a forward chaining tactic, or state a local definition or an abbreviation.

- **Linearity of scripts.** Try to make your script as linear as possible. Only open an indented piece of script for a non trivial subgoal. To improve linearity use the closing switch `//` and the optional rewrite switch `?` as often as needed. The `first` and `last` selectors also help closing an easy goal on the same line as the tactic which had created it.

- **Indentation of proofs.** Use indentation to separate the scripts of two-cases proofs (like when using a forward chaining tactic). If the proof of one of the goals is short enough, use selectors (see section 6.3) to close it as soon as it is created. If a branching proof has more that two cases, use bullets (see section 6.1) to separate the corresponding scripts.

- **Grabbing subterms in the goal.** Do not copy and paste large subterms from the current goal to the script. Most often patterns with wildcards will do the job for you, for instance by the mean of an abbreviation (see section 4.2).

- **Closing goals.** The last tactic of a script, which closes the goal, should be a closing tactic. Remember any tactic is turned into a closing one by being preceded by the `by` tactical. When the last tactic of a script takes a whole line of possibly chained tactics, the `by` tactical should start this line.

Name Policy

Lemmas in the distributed libraries respect the following name policy:

- **Generalities**

- Most of the time the name of a lemma can be read off its statement: a lemma named `fee_fie_foe` will say something about `(fee .. (fie ..(foe ..)..)..)` , e.g. lemma `size_cat` in `seq.v`.
- We often use a one-letter suffix to resolve overloaded notation, e.g., addn, addb, addr denote nat, boolean, ring addition, respectively. This policy does not necessarily apply to constants that should always be hidden behind a generic notation, and handled by a more generic theory.
- Finally, a handful of theorems have a historical name, e.g, `Cayley_Hamilton` or `factor_theorem`.

- **Structures and Records**
  - Each structure type starts with a lower case letter, and its constructor has the same name but with a capital first letter.
  - Each instance of a structure type has a name formed with the name of the carrier type, followed by an underscore and the one of the structure type like in `seq_sub_subType`, the structure of `subType` defined on `seq_sub` (see `fintype.v`). Notable exceptions to this rule are canonical constructions taking benefits of modular name spaces, like in `ssralg.v`.

- **Suffixes**
  - Lemma whose conclusion is a predicate, or an equality for a predicate: that predicate is a suffix of the lemma name, like in `addn_eq0` or `rev_uniq`.
  - Lemmas whose conclusion is a standard property such as `\char`, `<|`, etc.: the property should be indicated by a suffix (like `_char`, `_normal`, etc), so the lemma name should start by a description of the argument of the property, such as its key property, or its head constant. Thus we have `quotient_normal`, not `normal_quotient`, etc. This convention does not apply to monotony rules, for which we either use the name of the property with the suffix for the operator (e.g., `groupM`), or the name of the operator with the S suffix for subset monotony (e.g., `mulgS`).
  - We try to use and maintain the following set of lemma suffixes:
    * `0` : zero, or the empty set
    * `1` : unit, or the singleton set (use `_set1` for the latter to disambiguate)
    * `2` : two, doubling, doubletons
    * `3` etc, similarly
    * `A` : associativity
    * `C` : commutativity, or set complement (use `Cr` for trailing complement)
    * `D` : set difference, addition
    * `E` : definition elimination (often conversion lemmas)

* F : boolean false, finite type variant (as in canF_eq), or group functor
* G : group argument
* I : set intersection, injectivity for binary operators
* J : group conjugation
* K : cancellation lemmas
* L : left hand side (in canLR)
* M : group multiplication
* N : boolean negation, additive opposite
* P : characteristic properties (often reflection lemmas)
* R : group commutator, or right hand side (in canRL)
* S : subset argument, or integer successor
* T : boolean truth and Type-wide sets
* U : set union
* V : group or ring multiplicative inverse
* W : weakening
* X : exponentiation, or set cartesian product
* Y : group join
* Z : module/vector space scaling

Syntax for Gallina extensions

- **Irrefutable patterns.** Use the SSReflect syntax `let: ... in` for irrefutable patterns (see section 3.1).
- **Type annotations for anonymous arguments.** Using the open SSReflect syntax '`&` ⟨*term*⟩' and '`of` ⟨*term*⟩' instead of the standard `(_ :` ⟨*term*⟩`)` like in:

  ```
  Inductive list (A : Type) : Type := nil | cons of A & (list A).
  ```

  tend to make type annotations in definitions much more readable.
- **Open syntax.** Several SSReflect tactics (`pose`, `have`,...) support open syntax and applicative-style definitions for functions, which improves the readability of statements. It is hence recommended practise to use for instance:

  ```
  pose f x y := x + y.
  ```

  instead of the standard Coq equivalent tactic:

  ```
  pose f := (fun x y => x + y).
  ```

11.3 Interfaces

Figure 1 summarizes the main interfaces available in the present distribution, and their interdependencies.

The definition of each of these interfaces obeys a systematic pattern. However, unless the user wants to add a new structure to this this picture, it should not be necessary to delve into the details of these definition. On the other hand, this systematic way of defining the interfaces leads to a systematic way of populating them. Please refer to comments in the files for the rationale of each interface and for a review of the constructions provided.

Note that a `coqdoc` snapshot of the distributed libraries, including a search tool, is browsable on-line at `http://coqfinitgroup.gforge.inria.fr/ssreflect/`.

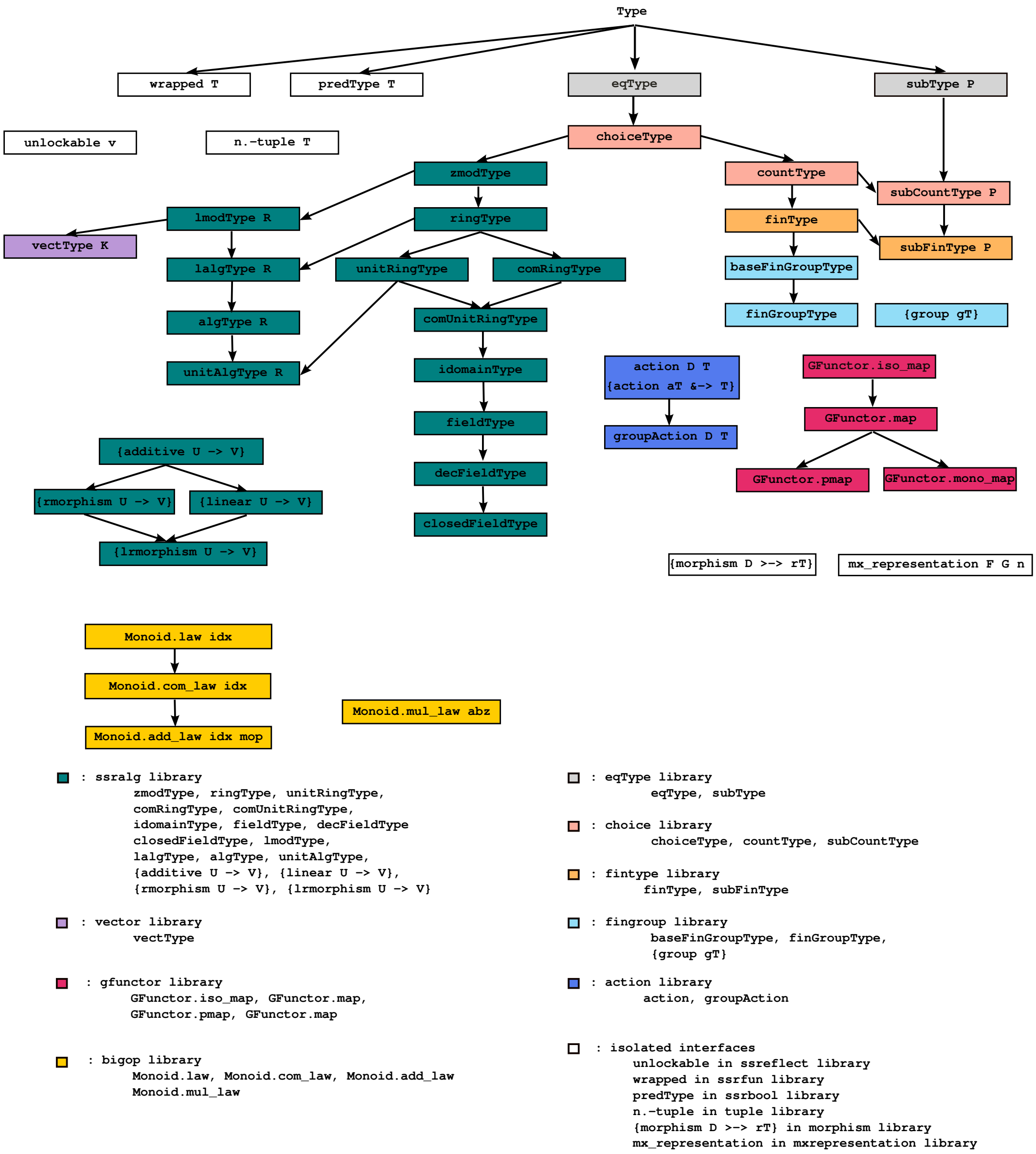

Figure 1: Interfaces defined in the ssreflect libraries. Note that the `finalg` library defines a clone of the ssralg sub-hierarchy for structures based on finite carriers.

12 Synopsis and Index

Parameters

⟨*d-tactic*⟩	one of the `elim`, `case`, `congr`, `apply`, `exact` and `move` SSReflect tactics
⟨*fix-body*⟩	standard Coq *fix_body*
⟨*ident*⟩	standard Coq identifier
⟨*int*⟩	integer literal
⟨*key*⟩	notation scope
⟨*name*⟩	module name
⟨*num*⟩	⟨*int*⟩ or $\mathcal{L}$-tac variable denoting a standard Coq numeral[a]
⟨*pattern*⟩	synonym for ⟨*term*⟩
⟨*string*⟩	standard Coq string
⟨*tactic*⟩	standard Coq tactic or SSReflect tactic
⟨*term*⟩	Gallina term, possibly containing wildcards

[a] The name of this $\mathcal{L}$-tac variable should not be the name of a tactic which can be followed by a bracket `[`, like `do`, `have`,...

Items and switches

<table>
<tr><td>⟨binder⟩</td><td>⟨ident⟩ | (⟨ident⟩[:⟨term⟩])</td><td>binder</td><td>p. 15</td></tr>
<tr><td>⟨clear-switch⟩</td><td>{⟨ident⟩⁺}</td><td>clear switch</td><td>p. 26</td></tr>
<tr><td>⟨ctx-ident⟩</td><td>⟨ident⟩|@⟨ident⟩|(⟨ident⟩)</td><td>context ident</td><td>p. 26</td></tr>
<tr><td>⟨c-pattern⟩</td><td>[⟨term⟩ in | ⟨term⟩ as] ⟨ident⟩ in ⟨term⟩</td><td>context pattern</td><td>p. 49</td></tr>
<tr><td>⟨d-item⟩</td><td>[⟨occ-switch⟩ | ⟨clear-switch⟩] [⟨term⟩|(⟨c-pattern⟩)]</td><td>discharge item</td><td>p. 26</td></tr>
<tr><td>⟨i-pattern⟩</td><td>⟨ident⟩ | _ | ? | * | [⟨occ-switch⟩]-> | [⟨occ-switch⟩]<- |
[⟨i-item⟩* | ... | ⟨i-item⟩*] | -</td><td>intro pattern</td><td>p. 28</td></tr>
<tr><td>⟨i-item⟩</td><td>⟨clear-switch⟩ | ⟨s-item⟩ | ⟨i-pattern⟩ | /⟨term⟩</td><td>intro item</td><td>p. 28</td></tr>
<tr><td>⟨int-mult⟩</td><td>[⟨int⟩]⟨mult-mark⟩</td><td>multiplier</td><td>p. 37</td></tr>
<tr><td>⟨occ-switch⟩</td><td>{[+|-]⟨num⟩*}</td><td>occur. switch</td><td>p. 18</td></tr>
<tr><td>⟨mult⟩</td><td>[⟨num⟩]⟨mult-mark⟩</td><td>multiplier</td><td>p. 37</td></tr>
</table>

⟨*mult-mark*⟩	`?` | `!`	multiplier mark	p. 37
⟨*r-item*⟩	[`/`]⟨*term*⟩ | ⟨*s-item*⟩	rewrite item	p. 43
⟨*r-prefix*⟩	[`-`][⟨*int-mult*⟩][⟨*occ-switch*⟩|⟨*clear-switch*⟩][`[`⟨*r-pattern*⟩`]`]	rewrite prefix	p. 43
⟨*r-pattern*⟩	⟨*term*⟩ | ⟨*c-pattern*⟩ | `in` [⟨*ident*⟩ `in`] ⟨*term*⟩	rewrite pattern	p. 43
⟨*r-step*⟩	[⟨*r-prefix*⟩]⟨*r-item*⟩	rewrite step	p. 43
⟨*s-item*⟩	`/=` | `//` | `//=`	simplify switch	p. 28

Tactics

Note: `without loss` and `suffices` are synonyms for `wlog` and `suff` respectively.

`move`	`idtac` or `hnf`	p. 21
`apply`	application	p. 24
`exact`		
`elim`	induction	p. 24
`case`	case analysis	p. 24
`rewrite` ⟨*rstep*⟩$^+$	rewrite	p. 43
`have` ⟨*i-item*⟩* [⟨*i-pattern*⟩] [⟨*s-item*⟩ | ⟨*binder*⟩$^+$] [`:` ⟨*term*⟩] `:=` ⟨*term*⟩	forward	p. 39
`have` ⟨*i-item*⟩* [⟨*i-pattern*⟩] [⟨*s-item*⟩ | ⟨*binder*⟩$^+$] `:` ⟨*term*⟩ [`by` ⟨*tactic*⟩]	chaining	
`have suff` [⟨*clear-switch*⟩] [⟨*i-pattern*⟩] [`:` ⟨*term*⟩] `:=` ⟨*term*⟩		
`have suff` [⟨*clear-switch*⟩] [⟨*i-pattern*⟩] `:` ⟨*term*⟩ [`by` ⟨*tactic*⟩]		
`wlog` [`suff`] [⟨*i-item*⟩] `:` [⟨*ctx-ident*⟩|⟨*clear-switch*⟩]* `/` ⟨*term*⟩	specializing	p. 39
`suff` ⟨*i-item*⟩* [⟨*i-pattern*⟩] [⟨*binder*⟩$^+$] `:` ⟨*term*⟩ [`by` ⟨*tactic*⟩]	backchaining	p. 39
`suff` [`have`] [⟨*clear-switch*⟩] [⟨*i-pattern*⟩] `:` ⟨*term*⟩ [`by` ⟨*tactic*⟩]		
`pose` ⟨*ident*⟩ `:=` ⟨*term*⟩	local definition	p. 15
`pose` ⟨*ident*⟩ ⟨*binder*⟩$^+$`:=` ⟨*term*⟩	local function definition	
`pose fix` ⟨*fix-body*⟩	local fix definition	
`pose cofix` ⟨*fix-body*⟩	local cofix definition	
`set` ⟨*ident*⟩ [`:`⟨*term*⟩] `:=` [⟨*occ-switch*⟩] [⟨*term*⟩|`(`⟨*c-pattern*⟩`)`]	abbreviation	p. 16

`unlock` [[⟨*r-prefix*⟩]⟨*ident*⟩]*	unlock	p. 53
`congr` [⟨*int*⟩] ⟨*term*⟩	congruence	p. 55

Tacticals

⟨*d-tactic*⟩ [⟨*ident*⟩]`:` ⟨*d-item*⟩+ [⟨*clear-switch*⟩]	discharge	p. 26
⟨*tactic*⟩ `=>` ⟨*i-item*⟩+	introduction	p. 28
⟨*tactic*⟩ `in` ⟨*ctx-ident*⟩+ [`|- *`]	localisation	p. 38
`do` [⟨*mult*⟩]`[`⟨*tactic*⟩`|...|`⟨*tactic*⟩`]` `do` ⟨*mult*⟩⟨*tactic*⟩	iteration	p. 37
⟨*tactic*⟩`; first` ⟨*num*⟩ `[`[⟨*tactic*⟩]`|...|`[⟨*tactic*⟩]`]` [`||` ⟨*tactic*⟩] ⟨*tactic*⟩`; last` ⟨*num*⟩ `[`[⟨*tactic*⟩]`|...|`[⟨*tactic*⟩]`]` [`||` ⟨*tactic*⟩]	selector	p. 36
⟨*tactic*⟩`; first` [⟨*num*⟩] `last` ⟨*tactic*⟩`; last` [⟨*num*⟩] `first`	subgoals rotation	p. 36

`by` `[`[⟨*tactic*⟩]`|...|`[⟨*tactic*⟩]`]`
`by []`
`by` ⟨*tactic*⟩
	closing	p. 35

Commands

`Hint View for` [`move`|`apply`]`/` ⟨*ident*⟩`[|`⟨*num*⟩`]`	view hint declaration	p. 67
`Hint View for apply//` ⟨*ident*⟩`[|`⟨*num*⟩`]`	right hand side double view hint declaration	p. 67
`Prenex Implicits` [⟨*ident*⟩]+	prenex implicits decl.	p. 13

13 Changes

13.1 SSReflect version 1.3

All changes are retrocompatible extensions but for:

- Occurrences in the type family switch now refer only to the goal, while before they used to refer also to the types in the abstractions of the predicate used by the eliminator. This bug used to affect lemmas like `boolP`. See the relative comments in `ssrbool.v`.
- Clear switches can only mention existing hypothesis and otherwise fail. This can in particular affect intro patterns simultaneously applied to several goals.
- A bug in the `rewrite` tactic allowed to instantiate existential metavariables occurring in the goal. This is not the case any longer (see section 7.2).
- The `fold` and `unfold` ⟨*r-items*⟩ for `rewrite` used to fail silently when used in combination with a ⟨*r-pattern*⟩ matching no goal subterm. They now fail. The old behaviour can be obtained using the `?` multiplier (see section 7.1).
- Coq 8.2 users with a statically linked toplevel must comment out the
  `Declare ML Module "ssreflect".`
  line at the beginning of `ssreflect.v` to compile the 1.3 library.

New features:

- Contextual `rewrite` patterns. The context surrounding the redex can now be used to specify which redex occurrences should be rewritten (see section 7.2).
  `rewrite [in X in _ = X]addnC.`
- Proof irrelevant interpretation of goals with existential metavariables. Goals containing an existential metavariable of sort `Prop` are generalized over it, and a new goal for the missing subproof is generated (see page 25 and section 7.2).
  `apply: (ex_intro _ (@Ordinal _ y _)).`
  `rewrite insubT.`
- Views are now part of ⟨*i-pattern*⟩ and can thus be used inside intro patterns (see section 5.4).
  `move=> a b /andP [Ha Hb].`
- Multiple views for `move`, `move ... in` and `apply` (see section 8.9).
  `move/v1/v2/v3.`
  `move/v1/v2/v3 in H.`
  `apply/v1/v2/v3.`
- `have` and `suff` idiom with view (see section 8.6).

  ```
  Lemma test (a b : bool) (pab : a && b) : b.
  have {pab} /= /andP [pa ->] // : true && (a && b) := pab.
  ```

- `have suff`, `suff have` and `wlog suff` forward reasoning tactics (see section 6.6).
  `have suff H : P.`

- Binders support in `have` (see section 6.6).
  `have H x y (r : R x y): P x -> Q y.`
- Deferred clear switches. Clears are deferred to the end of the intro pattern. In the meanwhile, cleared variables are still part of the context, thus the goal can mention them, but are renamed to non accessible dummy names (see section 5.4).
  `suff: G \x H = K; first case/dprodP=> {G H} [[G H -> -> defK]].`
- Relaxed alternation condition in intro patterns. The ⟨*i-item*⟩ grammar rule is simplified (see section 5.4).
  `move=> a {H} /= {H1} // b c /= {H2}.`
- Occurrence selection for `->` and `<-` intro pattern (see section 5.4).
  `move=> a b H {2}->.`
- Modifiers for the discharging '`:`' and `in` tactical to override the default behaviour when dealing with local definitions (let-in): `@f` forces the body of `f` to be kept, `(f)` forces the body of `f` to be dropped (see sections 5.3 and 6.5).
  `move: x y @f z.`
  `rewrite rule in (f) H.`
- Type family switch in `elim` and `case` can contain patterns with occurrence switch (see section 5.6).
  `case: {2}(_ == x)/ eqP.`
- Generic second order predicate support for `elim` (see section 8).
  `elim/big_prop: _`
- The `congr` tactic now also works on products (see section 7.4).

```
Lemma test x (H : P x) : P y.
congr (P _): H.
```

- Selectors now support $\mathcal{L}$-tac variables (see section 6.3).
  `let n := 3 in tac; first n last.`
- Deprecated use of `Import Prenex Implicits` directive. It must be replaced with the standard COQ `Unset Printing Implicit Defensive` vernacular command.
- New synonym `Canonical` for `Canonical Structure`.

13.2 SSREFLECT version 1.4

- User definable recurrent contexts (see section 7.2).
  `Notation RHS := (X in _ = X)%pattern`

- Contextual patterns in `set` and ‘`:`’ (see section 7.2).
  `set t := (a + _ in RHS)`
- NO-OP intro pattern (see section 5.4).
  `move=> /eqP-H /fooP-/barP`
- `if ... isn't` notation (see section 3.2).
  `if x isn't Some y then simple else complex y`

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399