A language of patterns for subterm selection

Georges Gonthier[1,2] and Enrico Tassi[2,3]

[1] Microsoft Research Cambridge
[2] INRIA - Microsoft Research joint centre
[3] INRIA, Laboratoire d'Informatique de l'Ecole Polytechnique

`gonthier@microsoft.com` `enrico.tassi@inria.fr`

Abstract. This paper describes the language of patterns that equips the SSReflect proof shell extension for the Coq system. Patterns are used to focus proof commands on subexpressions of the conjecture under analysis in a declarative manner. They are designed to ease the writing of proof scripts and to increase their readability and maintainability.
A pattern can identify the subexpression of interest approximating the subexpression itself, or its enclosing context or both. The user is free to choose the most convenient option.
Patterns are matched following an extremely precise and predictable discipline, that is carefully designed to admit an efficient implementation.
In this paper we report on the language of patterns, its matching algorithm and its usage in the formal library developed by the Mathematical Components team to support the verification of the Odd Order Theorem.

1 Introduction

In the design of proof languages many aspects have to be considered. Among them, the one that interests us the most is efficiency, both in writing proof scripts and in fixing them when they break.

Efficiency in writing and maintaining scripts is a crucial aspect for a language to be successfully adopted in a large development like the Mathematical Components library. That library comprises more than ten thousands lemmas, spread over one hundred files totalling over 113 thousand lines of code. For this development the SSReflect proof language was chosen, after its successful use in the formalization of the Four Color Theorem [10].

SSReflect is an extension of the Coq system, and inherits some of its strengths and weaknesses. The higher order logic of Coq allows to use computation as a form of proof and to enforce many invariants through its rich type system. These features were key ingredients in the formalization of Four Color Theorem, as well as in the development of many decision procedures [6,11], in interfacing Coq with external tools [18,2] and in organizing mathematical theories into higher-level packages [8,16]. The down side of this sophistication is that

The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

all basic operations, such as term comparison or term matching, have to take computation into account, thus becoming harder to predict due to the additional complexity.

SSReflect achieves *efficiency in writing* proof scripts giving the user a very precise and predictable language to assemble together carefully stated lemmas. The language is designed to be compact and compositional, with very few basic building blocks. Among them rewriting plays a special role and is indeed the most often used proof command. We describe how we improved its expressiveness while retaining, and in some circumstances even improving, its predictability.

To achieve *efficiency in maintaining* scripts the language constructs are equipped with a precise semantics that forces failures to happen early and locally. This is supplemented by support for script readability. SSReflect encourages to mix declarative steps, which assert intermediate results, and procedural statements that prove them. Declared statements are check points from which the user is likely to start replaying a broken proof, and the closer the failure is to one of these check points the easier is the fix. In this paper we describe how the `rewrite` command was made more declarative and less ambiguous.

Rewriting in particular, but also any other command that deals with subexpressions of the current conjecture, can in fact be rather ambiguous. Similar subexpressions are quite common in large conjectures and rewrite rules usually admit several different instantiations, each of which may occur multiple times. The two sources of ambiguity are thus: 1) the *instantiation* of the rewriting rule arguments to obtain a completely specified expression; 2) the *selection of the occurrences* of this expression to be affected. The standard approach to cope with the first source of ambiguity is to manually instantiate the rewriting rule. This approach requires the user to remember the nature and the order or names of the arguments of any rule, and thus hardly scales to a library with thousands of rewriting rules. Occurrences are usually selected by numbers. As we will show, this turns out to be rather inconvenient for script maintenance.

In this paper we describe the different approach adopted in the SSReflect proof language. The user is given a language of patterns to express in a concise and precise way which subterms of the current conjecture are affected by proof commands like `rewrite`.

The SSReflect Coq extension version 1.3pl2 for Coq 8.3 is available for download at the Mathematical Components web site[1]. The reader is not required to be acquainted with the specific logic of Coq or the proof language of SSReflect, but may find the reference manuals of the two tools [13,9] helpful.

In Section 2 we describe the language of patterns and give some examples on their intended use. Section 3 details the term matching algorithm. Section 4 compares our approach with the ones adopted in Coq, Matita and Isabelle/Isar.

2 A Language of Patterns

Most lines in a SSReflect proof script are procedural: they modify the current conjecture without explicitly stating the expected result. The `rewrite` command, whose argument is a list of rules, is a perfect example of this: instead of displaying a list of conjectures and expecting the system (and reader) to guess how to change one line into the next, the user quotes explicitly the *rules names* that justify the changes, and lets Coq do them sequentially. As Coq can usually reliably figure out how to apply each rule, this avoids a repetition of the parts of the conjecture that are *not* concerned by each change, and this is good for both writing and maintaining.

For example, consider these inequalities taken from Theorem 14.7 [4], that is the main result of the Local Analysis part of the Odd Order Theorem:

$$g \leq \left(1 + \frac{n}{z} - \sum_{M_i \in MX} (k_{M_i})^{-1} + \sum_{M_i \in MX} ((k_{M_i})^{-1} - (2z)^{-1})\right) \cdot g$$

$$g \leq \left(1 + \frac{n}{z} - \sum_{M_i \in MX} (k_{M_i})^{-1} + \sum_{M_i \in MX} (k_{M_i})^{-1} + (-(2z)^{-1} \cdot |MX|)\right) \cdot g$$

Rather than spelling out the second term above, we explicitly describe how to turn the fist inequality into the second one: the rightmost summation is split into two simpler ones using `big_split`; then the resulting summation of constant terms $\sum_{M_i \in MX}(-(2z)^{-1})$ is solved with `sumr_const`. This manipulation is expressed by the following command:

```
rewrite big_split sumr_const.
```

This is clearer and much more concise than the complex term it yields; indeed, in the actual proof we add more rules to carry out further simplifications.

However, sometimes guidance *is* needed, and we claim that it is best provided declaratively, using a little pattern language. Patterns are declarative in the sense that the user writes (an approximation of) the subterm that should be affected by a proof command. We will however assign a precise procedural interpretation to the pattern language in order to preserve the predictability of proof commands. It is also worth mentioning that the subterm a proof command may be focused on is often much smaller than the whole conjecture, and can usually be approximated by an even smaller pattern. So compactness is also retained.

We now give an informal presentation of patterns by some examples. We show how ambiguities in the execution of the `rewrite` proof command can be avoided thanks to a pattern and why the solution we propose is superior to existing ones.

Example 1 (Simple pattern).

This example lies in the context of the algebraic hierarchy [8] of the SSReflect library, where the infix notation for subtraction is a short hand for the addition of the opposite.

```
Infix "a - b" = (a + -b).
Lemma addrC x y : x + y = y + x.
Lemma addNKr x y : x + (- x + y) = y.

Lemma example : a + (b * c - a) + a = b * c + a.
```

The idea in the proof that will follow is to cancel the leftmost `a` with its opposite `-a`, thanks to `addNKr`. To rewrite with that lemma a preliminary step to move `-a` closer to `a` is needed: commutativity of addition must be used *on the correct subexpression*. Unluckily there are many occurrences of the addition operation. If no extra information is provided, the left hand side of the rule is the very ambiguous pattern `(_ + _)`, where `_` denotes a wild card. Its first instance encountered in pre-visit order is `(a + (b * c - a) + a)` that is not not the desired one, so additional information to disambiguate the rule is really needed.

To cope with this first form of ambiguity, *instantiation*, we have to specify the values of the quantified variables `x` and `y` to `addrC`. The standard approach adopted in many procedural proof languages, like the Coq's standard one, is to instantiate these variables manually, as in one of the following commands:

```
rewrite (addrC (b * c) (-a))          rewrite (addrC (x := b * c))
```

Both the previous commands turn the conjecture in the following one. From now on we will use a wave to underline the effect of a proof command.

$$\texttt{a + }\underset{\sim\sim\sim\sim\sim\sim}{\texttt{(-a + b * c)}}\texttt{ + a = b * c + a}$$

In the first case `x` and `y` are passed as arguments to the lemma by position. The left hand side of the rule becomes `(b * c - a)`. This expression has just one instance in the conjecture, thus the second kind of ambiguity, *occurrence selection*, does not occur. The second command passes the argument for `x` by name and leaves `y` undefined. The left hand side of the rule is thus a pattern `(b * c + _)` where `_` is a wild card. The system looks for an instance of that pattern in a prefix traversal of the conjecture, again finding the correct instance.

As anticipated in the introduction the main problem of this approach is that the user has to remember the order in which the variables of a rewrite rule are abstracted, or their names. What looks easy for the simple common lemma `addrC` quickly becomes an issue in the context of a large formalization like the one for the Odd Order Theorem, comprising over ten thousands lemmas.

The approach we propose is not only solving this usability issue but is also more compact, as shown in the following snippet.

```
rewrite [_ - a]addrC
```

The square brackets prefixing the rewrite rule `addrC` delimit the pattern `(_ - a)`. The pattern has a single, non ambiguous instance in the conjecture, namely `(b * c - a)`. Prefixing the rewriting rule name with a pattern the user substitutes the inferred pattern with a more specific one, better approximating the instance on which she wants to focus the proof command. □

A good interface design is most crucial to the usability of a theory library, and achieving one often requires several rounds of incremental refinement. When a potential improvement is identified, the statement of many lemmas is changed accordingly and their proofs are likely to break and thus require time consuming maintenance work. The general approach of the SSReflect language to lowering the cost of these refactoring activities is to detect failures as early as possible.

Example 2 (Proof script breakage). The lemma of the previous statement could be replaced (on purpose or by accident) by the following one:

$$\texttt{a + (b * c }\underline{\texttt{+ a}}\texttt{) + a = b * c + a.}$$

The user provided pattern `[_ - a]` seen before has no instance in this conjecture thus failure is immediately detected. On the contrary the command where `x` is instantiated by name with `(b * c)` would continue to produce an output, even if a different one. In that case the pattern `(b * c + _)` does have an instance occurring twice in the conjecture, namely `(b * c + a)`. Instead of signalling an error, the system changes the conjecture into the following one.

$$\texttt{a + }\underline{\texttt{(a + b * c)}}\texttt{ + a = }\underline{\texttt{a + b * c}}\texttt{.}$$

Failure will then happen at a later stage, with a conjecture that is very different from the one the author of the original proof script was seeing. Moreover the original intention of the user to move `(-a)` to the left can be recognized in the pattern `[_ - a]` but not in the instantiation `(addrC (x := b * c))`. □

As mentioned in the introduction the logic of Coq identifies terms up to conversion, i.e., unfolding of definitions and recursive functions computation. To develop a large library in a convenient way, the user often defines new concepts in terms of preexisting ones. In most cases part of the theory already developed naturally transports to the new concepts. As we see in the following example this may introduce an additional degree of ambiguity the user has to deal with.

Example 3 (Pattern forcing definition unfolding). In the context of the library on lists the user finds the function `map` to apply a function over a list and some of its properties. The related lemma `eq_in_map` states that a function `f` can be replaced with another function `g` if `f` and `g` are point wise equal (denoted `=1`) on the list they are mapped on. `map_comp` proves that mapping two functions in a row is the same as mapping their functional composition (denoted with `\o`). `id` is the identify function.

```
Lemma eq_in_map s f g : {in s, f =1 g} -> map f s = map g s.
Lemma map_comp f g s : map (f \o g) s = map f (map g s).
Lemma map_id s : map id s = s.
```

The `iota` function builds the list of consecutive integers given the first element and the list length. On top of `map` and `iota` the user defines the `graph` of a function over an integer interval `[0,n[` as the list of its values on that interval. An obvious property is that if a function `f` behaves as the identity on the graph of `g` on a given interval, then the graph of `(f \o g)` is equal to the graph of `g` on the same interval.

```
Definition graph f n := map f (iota 0 n).
Lemma graph_comp f g n (pf : {in graph g n, f =1 id}) :
  graph (f \o g) n = graph g n.
```

The property follows trivially from the theory of lists, but the conjecture does not mention any list operation. Nevertheless the `map_comp` rewrite rule can be used as follows:

```
rewrite [graph _ n]map_comp
```

The first instance of the pattern `[graph _ n]`, traversing the conjecture `(graph (f \o g) n = graph g n)` from left to right, is `(graph (f \o g) n)`. This is where the `map_comp` rule can apply. In fact, unfolding the definition of `graph` exposes `(map (f \o g) (iota 0 n))` that is clearly an instance of the pattern `(map (_ \o _) _)` given by the rule `map_comp`. The resulting conjecture is reported below.

```
map f (map g (iota 0 n)) = graph g n.
```

One can then complete the proof rewriting with the `eq_in_map` lemma, whose hypothesis is indeed equivalent to the `pf` assumption, and then conclude with `map_id`. □

One could argue that in the previous example the system is not "clever enough" and could exploit the fact that `graph` is defined in terms of `map` to find the subterms to be rewritten. According to our experience this would make the rewrite command less predictable. For example consider a conjecture in which both `graph` and `map` occur in that order. The `graph` occurrence may be rewritten even if the user does not know that graph is defined in terms of `map`. Moreover the user still needs a way to focus on `map` if that is what she wants.

The usual alternative approach is to manually unfold some of the occurrences of `graph` to expose `map`. This is again not only more verbose, but less informative in the script. With the pattern `[graph _ n]` the user clearly states that the whole matched expression is an instance of the left hand side of the rewriting rule. If `graph` is redefined with a different expression that strictly contains an occurrence of `map`, the script with the pattern breaks immediately, while the one just unfolding `graph` may signal an error at a later stage.

The previous examples may look a bit artificial, and in fact they were chosen to be reasonably self contained at the cost of resulting a bit simplistic. On the contrary the one below is taken from the quite involved proof of the Wielandt fixpoint [12] Theorem formalized by A. Mahboubi. It is a rather technical result required to prove the Odd Order Theorem, and was one of the motivating examples for contextual patterns, that are introduced immediately after.

Example 4 (Contextual pattern). The context of this example is group theory and the study of group morphisms. The system prints above the double bar the hypotheses accumulated by the user so far. In particular that `X` is equal to the image of the morphism `fact_g` over `X` quotiented by the kernel of `g`. The user needs to rewrite the first occurrence of `X` with the `imgX` equation in order to advance in her proof.

```
nkA : joing_group A X \subset 'N('ker g)
fact_g := factm skk nkA : coset_groupType ('ker g) -> gT
imgX : X = fact_g @* (X / 'ker g)
=================================
minnormal (fact_g @* (A / 'ker g)) X ->
  minnormal (A / 'ker g) (X / 'ker g)
```

Here the rewrite rule is fully instantiated, thus the ambiguity is given by the fact that its left hand side `X` occurs at least twice in the conjecture (the implication below the double bar). In fact, the notation system of Coq hides many other occurrences of `X`. The morphism image construction `@*` is polymorphic over the type of the morphism `fact_g`, that is itself a dependently typed construction. In particular it depends on the assumption `nkA` whose type mentions `X`. The logic of Coq features explicit polymorphism, like system $\mathcal{F}$, so types occur as arguments of polymorphic functions even if some syntactic sugar hides them. As it turns out, the occurrence of `X` we are interested in is number twenty-nine, even if it the first one displayed by the system.

The pattern we propose to unambiguously identify that desired occurrence uses its enclosing context. In the following snippet, `R` is a name bound in the expression following the `in` keyword.

```
rewrite [R in minnormal _ R]imgX
```

The intended meaning is to focus the `rewrite` command on the subterm identified by `R` in the first occurrence of the context `(minnormal _ R)`. While being more verbose than the occurrence number `{29}`, it is way easier to write, since no guessing is needed. Moreover in case the script breaks the original intent of the user is clearly spelled out. □

2.1 Syntax and Semantics

The syntax is defined by two grammar entries: ⟨*c-pattern*⟩ for contextual patterns and ⟨*r-pattern*⟩ for their superset rewrite patterns. Contextual patterns are meant to identify a specific subterm, and can be used as arguments of the SSReflect commands `set`, `elim` and `:` (colon), see [9, Sections 4.2 and 5.3], respectively used to declare abbreviations, perform induction or generalize the current conjecture. Rewrite patterns are a strict superset of contextual patterns adding the possibility of identifying all the subterms under a given context. They can be used as arguments of the SSReflect `rewrite` command, see [9, Section 7].

⟨*c-pattern*⟩ ::= [⟨*tpat*⟩ `as` | ⟨*tpat*⟩ `in`] ⟨*ident*⟩ `in` ⟨*tpat*⟩ | ⟨*tpat*⟩
⟨*r-pattern*⟩ ::= ⟨*c-pattern*⟩ | `in` [⟨*ident*⟩ `in`] ⟨*tpat*⟩

Here ⟨*tpat*⟩ denotes a term pattern, that is a generic Coq term, possibly containing wild cards, denoted with `_` (underscore).

We now summarize the semantics of both categories of patterns. We shall call *redex* the pattern designed to identify the subterm on which the proof command will have effect. We shall also use the word *match* in an informal way recalling the reader's intuition to pattern matching. The precise meaning of matching will be described in Section 3.

Contextual patterns ⟨*c-pattern*⟩ For every possible pattern we identify the *redex* and define which subterms are affected by a proof command that uses such pattern. We then point out the main subtleties with some examples.

⟨*tpat*⟩ is the degenerate form of a contextual pattern, where the context is indeed empty. The redex is thus the whole ⟨*tpat*⟩. The subterms affected by this simple form of pattern are all the occurrences of the first instance of the redex. See Example 5.

⟨*ident*⟩ in ⟨*tpat*⟩ is the simplest form of contextual pattern. The redex is the subterm of the context ⟨*tpat*⟩ bound by ⟨*ident*⟩. The subterm affected are all the subterms identified by the redex ⟨*ident*⟩ in all the occurrences of the first instance of ⟨*tpat*⟩. See Example 6.

⟨*tpat*⟩$_1$ as ⟨*ident*⟩ in ⟨*tpat*⟩$_2$ is a form of contextual pattern where the redex is explicitly given as ⟨*tpat*⟩$_1$. It refines the previous pattern by specifying a pattern for the context hole named by ⟨*ident*⟩. The subterms affected are thus the ones bound by ⟨*ident*⟩ in all the occurrences of the first instance of ⟨*tpat*⟩$_2$[⟨*tpat*⟩$_1$/⟨*ident*⟩], i.e., ⟨*tpat*⟩$_2$ where ⟨*ident*⟩ is replaced by ⟨*tpat*⟩$_1$. See Example 8.

⟨*tpat*⟩$_1$ in ⟨*ident*⟩ in ⟨*tpat*⟩$_2$ is the last form of contextual pattern and is meant to identify deeper contexts in two steps. The redex is given as ⟨*tpat*⟩$_1$ and the subterms affected are all the occurrences of its first instance inside the subterms bound by ⟨*ident*⟩ in all the occurrences of the first instance of ⟨*tpat*⟩$_2$. The context described by this pattern is thus made of two parts: an explicit one given by ⟨*tpat*⟩$_2$, and an implicit one given by the matching of the redex ⟨*tpat*⟩$_1$ that could occur deep inside the term identified by ⟨*ident*⟩. See Example 7.

Example 5. We have already seen in Example 4 the first form of pattern. Here we give another example to stress that *all* the occurrences of the first instance of the pattern are affected. Take the conjecture:

```
(a - b) + (b - c) = (a - b) + (d - b)
```

The proof command `rewrite [_ - b]addrC` changes the conjecture as follows because the first instance of the pattern `(_ - b)` is `(a - b)`, and not `(d - b)` since the conjecture is traversed in pre visit order.

```
(-b + a) + (b - c) = (-b + a) + (d - b)
```

The subterm `(a - b)` has another occurrence in the right hand side of the conjecture that is affected too. □

The second form of contextual pattern comes handy when the subterm of interest occurs immediately under a context that is easy to describe.

Example 6. Take the following conjecture:

```
0 = snd (0 * c, 0 * (a + b))
```

To prove this conjecture it is enough to use the annihilating property of `0` on `(a + b)` and compute away the `snd` projection. Unfortunately that property also applies to `(0 * c)`. We can easily identify `(0 * (a + b))` with the second form of contextual pattern, mentioning the context symbol `snd` and marking with `X` the argument we are interested in. The resulting command is thus `rewrite [X in snd (_, X)]mul0n`. □

A typical example of the last form is with the `set` command, that creates a local definition grabbing instances of the definendum in the conjecture.

Example 7. Take the following conjecture:

```
a + b = f (a^2 + b) - c
```

To make it more readable one may want to abbreviate with `n` the expression `(a^2 + b)`. The command `set n := (_ + b in X in _ = X)` binds to `n` all the occurrences of the first instance of the pattern `(_ + b)` in the right hand side only of the conjecture.

$\texttt{a + b = }\underline{\texttt{f }\uwave{\texttt{n}}\texttt{ - c}}$

Note that the pattern `(_ + b)` could also match `(a + b)` in the left hand side of the conjecture, but the `(in X in _ = X)` part of the contextual pattern focuses the right hand side only. From now on we will always underline with a straight line the subterm selected by the context part of a pattern (i.e., the subterm identified by the bound variable `X` in the previous example). □

In Section 2.3 we describe how the user can define shortcuts for commonly used contexts, and thus write the previous pattern as: `set n := (_ + b in RHS)`.

We give an example of the third ⟨*c-pattern*⟩ form together with the examples for ⟨*r-pattern*⟩s.

Rewrite patterns ⟨*r-pattern*⟩ The `rewrite` command supports two more patterns obtained by prefixing the first two ⟨*c-pattern*⟩s with the `in` keyword. The intended meaning is that the pattern identifies all subterms of the specified context. Note that the `rewrite` command can always infer a redex from the shape of the rewrite rule. For example the `addrC` rule of Example 1 gives the redex pattern `(_ + _)`.

in ⟨*tpat*⟩ is the simplest form of rewrite pattern. The redex is inferred from the rewriting rule. The subterms affected are all the occurrences of the first instance of the redex inside all the occurrences of the first instance of ⟨*tpat*⟩.

in ⟨*ident*⟩ in ⟨*tpat*⟩ is quite similar to the last form of contextual pattern seen above, but the redex is not explicitly given but instead inferred from the rewriting rule. The subterms affected are all the occurrences of the first instance of the redex inside the subterms identified by ⟨*ident*⟩ in all the occurrences of the first instance of ⟨*tpat*⟩.

Example 8. The first form of ⟨*r-pattern*⟩ is handy when we want to focus on the subterms of a given context. Take for example the following conjecture:

```
f (a + b) (2 * (a + c)) + (c + d) + f a (c + d) = 0
```

The command `rewrite [in f _ _]addrC` focuses the matching of the redex inferred from the `addrC` lemma, `(_ + _)`, to the subterms of the first instance of the pattern `(f _ _)`. Thus the conjecture is changed into

```
f (b + a) (2 * (a + c)) + (c + d) + f a (c + d) = 0
```

If the user had in mind to exchange `a` with `c` instead, she could have used a pattern like `[in X in f _ X]addrC`, to focus the matching of the redex on the second argument of `f`, obtaining:

```
f (a + b) (2 * (c + a)) + (c + d) + f a (c + d) = 0
```

The last form of ⟨*c-pattern*⟩ could be used to focus on the last occurrence of `(c + d)`. The pattern `[_ + d as X in f _ X]` would first match the context substituting `(_ + d)` for `X`. The pattern `(f _ (_ + d))` focuses on the second occurrence of `f`, then the `X` identifier selects only its second argument that is exactly where the rewriting rule `addrC` is applied.

```
f (a + b) (2 * (a + c)) + (c + d) + f a (d + c) = 0
```

It is important to note that even if the rewrite proof command always infers a redex from the rewrite rule, a different redex can be specified using a ⟨*c-pattern*⟩. This is especially convenient when the inferred redex is masked by a definition, as in Example 3 .

2.2 Matching order

In the previous examples we implicitly followed a precise order when matching the various ⟨*tpat*⟩s part of a ⟨*c-pattern*⟩ or ⟨*r-pattern*⟩. For example we always matched the context part first. We now make this order explicit.

⟨*tpat*⟩, ⟨*ident*⟩ in ⟨*tpat*⟩ All the subterms of the conjecture are matched against ⟨*tpat*⟩.

⟨*tpat*⟩$_1$ as ⟨*ident*⟩ in ⟨*tpat*⟩$_2$ All the subterms of the conjecture are matched against ⟨*tpat*⟩$_2$[⟨*tpat*⟩$_1$/⟨*ident*⟩].

⟨*tpat*⟩$_1$ in ⟨*ident*⟩ in ⟨*tpat*⟩$_2$ First, subterms of the conjecture are matched against ⟨*tpat*⟩$_2$. Then the subterms of the instantiation of ⟨*tpat*⟩$_2$ identified by ⟨*ident*⟩ are matched against ⟨*tpat*⟩$_1$.

in ⟨*ident*⟩ in ⟨*tpat*⟩ First, subterms of the conjecture are matched against ⟨*tpat*⟩. Then the subterms of the instantiation of ⟨*tpat*⟩ identified by ⟨*ident*⟩ are matched against the inferred redex (that is always present since this pattern has to be used with the `rewrite` proof command).

in ⟨*tpat*⟩ First, subterms of the conjecture are matched against ⟨*tpat*⟩. Then the instantiation of ⟨*tpat*⟩ is matched against the inferred redex.

If one of the first four patterns is used in conjunction with `rewrite`, the instance of the redex is then matched against the pattern inferred from the rewriting rule. The matching order is very relevant to predict the instantiation of patterns.

Example 9. For example in the pattern `((_ + _) in X in (_ * X))`, the matching of the sub pattern `(_ + _)` is restricted to the subterm identified by `X`. Take the following conjecture:

$$\dashuline{\texttt{a + b}}\texttt{ + (a * }\underline{\texttt{(}(\uwave{\texttt{a + b}})\texttt{ * d)}}\texttt{) = 0}$$

The dash underlined subterm would be a valid instance of `(_ + _)` but is skipped since it does not occur in the right context. In fact `(_ * X)` is matched first. The subterm corresponding to `X` is `((a + b) * d)`. Then its subterms are matched against `(_ + _)` and the first, and only, occurrence is $\texttt{(}\uwave{\texttt{a + b}}\texttt{)}$. □

2.3 Recurring contexts

Whilst being quite expressive, contextual patterns tend to be a bit verbose and quite repetitive. For example to focus on the right hand side of an equational conjecture, one may have to specify the pattern `(in X in _ = X)`.

With a careful use of the notational mechanism of Coq we let the user define abbreviations for common contexts, corresponding to the ⟨*ident*⟩ `in` ⟨*tpat*⟩ part of the pattern. The definition of the abbreviation `RHS` is as follows.

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

There the notational scope `%pattern` interprets the infix `in` notation in a peculiar way, encoding in a non ambiguous way the context `(X in _ = X)` in a simple ⟨*tpat*⟩. Then, when the system parses `(in RHS)` as an instance of `in` ⟨*tpat*⟩ it recognizes the context encoded in ⟨*tpat*⟩ and outputs the abstract syntax tree for `in` ⟨*ident*⟩ `in` ⟨*tpat*⟩.

3 Term matching

We now give a precise description of the matching operation for ⟨*tpat*⟩. The main concerns are performances and predictability.

Predictability has already been discussed in relation to Example 3. A lemma that talks about the `map` function should affect occurrences of the `map` function only, even if other subterms are *defined* in terms of `map`, unless the user really means that. Indeed the most characterizing feature of the logic of Coq is to identify terms up to definition unfolding and computation. That allows to completely omit proof steps that are pure computations, for example `(0 + x)` and `x` are just equal (not only provably equal) for the standard definition of addition.

Performance is a main concern when one deals with large conjectures. To take advantage of the computational notion of comparison the logic offers, one could be tempted to try to match the pattern against any subterm, even if the subterm shares no similarity with the pattern itself. A higher order matching procedure could find that the pattern actually matches up to computation. Nevertheless, this matching operation could be expensive. Especially because it is expected to fail on most of the subterms and failure is certain only after both the pattern and the subterm are reduced to normal forms.

The considerations about performances and predictability lead to the idea of *keyed matching*. The matching operation is attempted only on subterms whose head constant is equal to the head constant (the *key*) of the pattern, *verbatim*. Arguments of the key are matched using the standard higher order matching algorithm of Coq, which takes computation into account.

Take for example the conjecture `(`*huge* `+ x * (1 - 1) = 0)` and the rewrite rule `muln0` that gives the redex `(_ * 0)`. The key of the redex `*` is compared with the head of all the subterms of the conjecture. This traversal is linear in size of the conjecture. The higher order matching algorithm of Coq is thus run on the candidate subterms identified by the keyed filtering phase, like `(x * (1 - 1))`. In that case the second argument of the pattern, `0`, matches `(1 - 1)` up to reduction. The *huge* subterm, assuming it contains no `*`, is quickly skipped, and the expensive but computation aware matching algorithm of Coq is never triggered on its subterms.

3.1 Gadgets

To adhere to the keyed matching discipline, that is different from the standard one implemented in Coq, we had to implement our own stricter matching algorithm inside the SSReflect extension, piggybacking on Coq's general unification procedure. This gave us the opportunity to tune it towards our needs, adding some exceptions for unit types, abstract algebraic structures, etc.

Unit types are types that have only one canonical inhabitant. In a library of the extent of the SSReflect's one, there are many of them. For example there is only one matrix of 0 rows or 0 columns, there is only one natural number in the interval subtype `[0,1[`, there is only one 0-tuple, etc.

In the statement of the canonicity lemma for these types, the inferred redex is just a wild card, i.e., there is no key. In the following example the type `'I_n` denotes the subtype of natural numbers strictly smaller than `n`.

```
Lemma ord1 (x : 'I_1) : x = 0.
```

A pattern with no key results in a error message in SSReflect. Nevertheless SSReflect supports a special construction to mark wild cards meant to act as a key. In that case the search is driven by the type of the wild card. In the following example the `(unkeyed x)` notation denotes any subterm of type `'I_1`.

```
Notation unkeyed x := (let flex := x in flex).
Lemma ord1 (x : 'I_1) : unkeyed x = 0.
```

Another notable exception is the case in which the key is a projection. The logic of Coq can represent dependently typed records [15], that are non homogeneous n-tuples where the type of the i-th element can depend on the values of the previous $i-1$ elements. This is a key device to model abstract algebraic structures [16,8,17,6], like a `Monoid` as a record with three fields: a type `mT`, a binary operation mop on mT and the associative property for mop[5].

```
Structure Monoid := {      mT (M : Monoid) : Type
  mT : Type;               mop (M : Monoid) : mt M -> mt M -> mt M
  mop : mT -> mT -> mT;    massoc (M : Monoid) (x y z : mt M) :
  massoc : assoc mop }       mop M x (mop M y z) = mop M (mop M x y) z
```

Constants mT, mop and massoc are projections for the corresponding record fields. Their types are reported on the right.

If we look at the statement of any lemma equating Monoid expressions we note that the key for the operation is mop, as in the statement of massoc that leads to the pattern (mop _ _ (mop _ _ _)).
Algebraic reasoning is interesting because all the results proved in the abstract setting apply to any instance of the algebraic structure. For example lists and concatenation form a Monoid. Nevertheless, any conjecture about lists is going to use the concrete concatenation operation cat. The higher order matching algorithm of Coq can be instrumented to exploit the fact that there exists a canonical Monoid over lists and is thus able to match (mop _ _ _) against (cat s1 s2) assigning to the first wild card this canonical Monoid structure. Unfortunately, our matching algorithm would fail to match any occurrence of cat against the key mop, because they not equal verbatim.
The exception to the keyed matching discipline we considered is to compare as verbatim equal keys that happen to be projections with any of their canonical values. For example the key mop will match list concatenation, but also integer addition etc. and any other operation that is declared to form a Monoid. Note that this matching requires to correctly align the pattern with the term to be matched. In case of the term (cat s1 s2), the pattern (mop _ _ _) has to be matched as follows: the cat term has to be matched against the initial part of the pattern (mop _), that corresponds to the projection applied to the unknown Monoid structure. Then the following two arguments s1 and s2 have to be matched against the two wild cards left.

The last exception is for patterns with a flexible key but some arguments, like (_ a b). The intended meaning is that the focus is on the application of a function whose last two arguments are a and b. This kind of pattern lacks a key and its match is attempted on any subterms. This is very convenient when the head constant of the expression to be focused is harder to write than the arguments. For example the expression ([predI predU A B & C] x) represents the application of a composite predicate to x. This expression can be easily identified with the pattern (_ x).

3.2 Verbatim matching of the pattern

There is an important exception to the keyed matching discipline worth explaining in more details. We begin with a motivating example, showing a situation in which the keyed matching prevents the user from freely normalizing associativity.

[5] We omit the unit for reasons of space

Example 10 (Motivating example).

```
Lemma example n m : n + 2 * m = m + (m + n)
by rewrite addnA addnC !mulSn addn0.
```

Without the verbatim matching phase, the application of the first rewrite rule, `addnA`, would turn the conjecture into:

```
n + m + (1 * m) = m + (m + n)
```

In fact, the redex inferred from `addnA` is `(_ + (_ + _))`, and the first occurrence of its key `+` is in the left hand side of the conjecture. Since the definition of multiplication for natural numbers is computable Coq compares `(2 * m)` and `(m + (1 * m))` as equal. Thus `(n + (m + (1 * m)))` is successfully matched against the redex failing the user expectations. Thus the user is forced to add a pattern like `(m + _)` to `addnA` that is somewhat unnatural, since there is only one visible occurrence of the redex `(_ + (_ + _))`. □

To address this issue, the term pattern matching algorithm performs two phases. In the first no reduction takes place, and syntactic sugar, like hidden type arguments or explicit type casts, is taken into account, essentially erasing invisible arguments from the pattern and the conjecture. The second phase is driven by the pattern key and allows full reduction on its arguments.

Once the pattern is instantiated the search for its occurrences is again keyed, and arguments are compared pairwise allowing conversion. For example consider the pattern `(_ + _)` and its first instance `(1 + m)`. Other occurrences are searched using `+` as the key, and arguments are compared pairwise. Thus a term like `(0 + (1 + m))`, that is indeed computationally equal to `(1 + m)`, is not an occurrence of `(1 + m)`, since `1` does not match `0` and `m` does not match `(1 + m)`. On the contrary `(1 + (0 + m))` is an occurrence of `(1 + m)`.

4 Related works

In order to compare the approach proposed in this paper with other approaches we consider the following conjecture `f (a + 0) = a + 0 * b` where we want to replace `(a + 0 * b)` with `a` using the equation `forall x, x + 0 = x` named `addn0`. Note that the logic of Coq compares `(0 * b)` and `0` as equal.

The first comparison that has to be made is with the standard Coq mechanism to focus on subexpressions. The `pattern` command ([5, Section 6.3.3]) pre-processes the conjecture putting in evidence the sub term of interest. The user has to spell out completely this subexpression, and if it occurs multiple times she can specify occurrence numbers. This leads to the proof script: `pattern (a + 0 * b); rewrite addn0`. Note that the two expressions `(a + 0)` and `(a + 0 * b)` are not consider as equal by the matching algorithm used by `pattern` and by Coq's `rewrite`. For example `pattern (a + 0) at 2` fails as well as `rewrite (addn0 a) at 2`. We believe our approach is superior because the intent of the user is not obfuscated by commands to prepare the conjecture and because it takes computation into account when comparing terms. In SSReflect one can perform the desired manipulation with `rewrite [RHS]addn0` where `RHS` is a pattern notation as in Section 2.3.

Matita [3] is an ITP based on the same logic of Coq with a modern graphical interface. Its proof language is similar to the one of Coq but (parts) of proof commands can be generated by mouse gestures. In particular the user can focus on a sub term selecting it with the mouse. The selection of the right hand side of the running example results in the following snippet `rewrite add0n in |- ???%` where `%` marks the subterm of interest, while the three `?` respectively stand for the head symbol (`=`), the invisible type parameter `nat` and the left hand side, all of which have to be ignored by the `rewrite` tactic. While this approach is intuitive and effective in writing proof scripts, it does not ease their maintenance, since the textual representation of the visual selection is not very informative for the reader, especially when selection happens deep inside the conjecture.

The Isabelle prover [14] implements a framework on top of which different logics and proof languages are built. The most used combination is higher order logic and the declarative proof language Isar [19]. In this setting some of the complexity introduced by the logic of Coq disappears. For example terms are not considered to be equal taking definitions into account. Moreover in the declarative style proposed by Isar language the user spells out the conjecture more frequently, and some automation tries to prove it, finding for example which occurrences need to be rewritten. To avoid repeating large expressions the Isar language provides the `(is` ⟨*pattern*⟩`)` construct [19, Section 3.2.6] to bind expressions to schematic variables that can be reused later on.

5 Conclusion

This paper presents the language of patterns adopted by the SSReflect proof shell extension for the Coq system. The language was introduced in SSReflect version 1.3 in March 2011 mainly to improve the effectiveness of the `rewrite` proof command. Version 1.4 makes the pattern language consistently available to all language constructs that have to identify subexpressions of the conjecture.

The choices made in the design of the pattern language and its semantics are based on the experience gathered in the Mathematical Components team on the formal proof of the Odd Order Theorem during the last five years, and the implementation has been validated by roughly one year of intense use. As of today this formalization comprises 113,384 lines of code, of which 34,077 contain a `rewrite` statement. Of these 2,280 have been changed to take advantage of the pattern language, and some other 2,005 lines (of which 1,705 contain a `rewrite` command) still make use of occurrence numbers and could be modified too.

A line of ongoing development is to separate the code implementing the pattern matching algorithm and the parsing of patterns concrete syntax from the rest of the SSReflect extension, making a separate extension. This will allow proof commands provided by other Coq extensions to benefit from the same pattern language. A possible application is in the AAC Coq extension that automates proofs dealing with associativity and commutativity. In fact in [7] Braibant and Pous express the need for a linguistic construct to select subexpressions other than occurrence numbers.

We thank Frédéric Chyzak for some very productive discussions on the topic.

References

1. Mathematical Components website. http://www.msr-inria.inria.fr/Projects/math-components/, http://coqfinitgroup.gforge.inria.fr/ssreflect-1.3/.
2. Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with imperative features and its application to SAT verification. In *ITP*, volume 6172 of *LNCS*, pages 83–98, 2010.
3. Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007.
4. Helmut Bender and Georges Glauberman. *Local analysis for the Odd Order Theorem.* Number 188 in London Mathematical Society Lecture Note Series. Cambridge University Press, 1994.
5. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions.* Springer-Verlag, 2004.
6. Thomas Braibant and Damien Pous. An efficient Coq tactic for deciding Kleene algebras. In *ITP*, pages 163–178, 2010.
7. Thomas Braibant and Damien Pous. Tactics for reasoning modulo AC in Coq. In *CPP*, pages 167–182, 2011.
8. François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *TPHOLs*, LNCS, pages 327–342, 2009.
9. Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A Small Scale Reflection extension for the Coq system.* INRIA Technical report, 00258384.
10. Geroges Gonthier. Formal proof – the four color theorem. *Notices of the American Mathematical Society*, 55:1382–1394, 2008.
11. Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In *TPHOLs*, pages 98–113, 2005.
12. B. Huppert and N. Blackburn. *Finite groups II.* Number vol. 2 in Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete. Springer-Verlag, 1982.
13. The Coq development team. *The Coq proof assistant reference manual*, 2011. Version 8.3.
14. Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.
15. Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.
16. Claudio Sacerdoti Coen and Enrico Tassi. Working with mathematical structures in type theory. In *TYPES*, volume 4941 of *LNCS*, pages 157–172, 2007.
17. Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures In Computer Science*, 21:1–31, 2011.
18. Laurent Théry and Guillaume Hanrot. Primality proving with elliptic curves. In *TPHOLs*, pages 319–333, 2007.
19. Markus Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In *TPHOLs*, pages 167–184, 1999.