A language of patterns for subterm selection

Georges Gonthier, Enrico Tassi

A language of patterns for sub-term selection

Georges Gonthier, Enrico Tassi

INRIA - Microsoft Research joint center
gonthier@microsoft.com - enrico.tassi@inria.fr

Abstract. This paper describes the language of patterns provided by the SSReflect language to restrict the scope of a proof command to a sub expression of the current conjecture. The pattern language is designed to ease the writing of proof scripts, to increase their readability and maintainability while retaining the compactness and predictability that are typical of the SSReflect language.

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 when writing proof scripts and when dealing with their breakage.

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 summing up to 113K lines of code. For that development the SSReflect proof language was chosen, after its successful adoption for the formalization of the four color theorem [7].

SSReflect is an extension of the Coq system, and inherits some of its strengths and weaknesses. Its higher order logics allows to use computation as a form of proof and many invariants are enforced by the rich typing discipline. The bad side of this is that all basic operations, like term comparison or term matching, have to take computation into account, becoming more complex and thus harder to predict.

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 therefore the mostly 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. Readability is a companion to that. 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 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 sub expressions of the current conjecture, can in fact be rather ambiguous. Similar sub expressions are quite common in large conjectures and rewrite rules usually admit several different instantiations and each of them may occur multiple times. The two sources of ambiguity are thus: 1) the *instantiation* of the rewriting rule; 2) the *selection of occurrences* of these instances. The most standard approach to cope with the first source of ambiguity is to manually instantiate the rewriting rule. This requires the user to remember the order, or the names, of the arguments of any rule. This approach hardly scales to a library with thousands of rewriting rules. Occurrence are usually selected by numbers, that, as we will show, turn 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 1.4 is available for download at the Mathematical Components web site[1]. To understand the current article the reader is not required to be acquainted with the specific logics of Coq or the proof language of SSReflect, but may find helpful the reference manuals of the two tools [8,6].

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 describes how complex patterns are implemented in terms of the term matching procedure described in Section 3. Section 5 compares our approach with others adopted in different provers and proof languages, like Matita, Isabelle/Isar and Coq's standard proof language.

2 A Language of Patterns

The majority of lines in an SSReflect proof scripts is procedural. While the user can state intermediate goals, she mostly inputs SSReflect proof commands to modify the current conjecture without stating the expected result of the command. Even so we prefer to adopt the declarative approach of introducing a language of patterns.

Patterns are declarative, in the sense that the user writes (an approximation of) the subterm that should be affected by a proof command and not how to reach that subterm. Anyhow we are going to attach a very precise procedural interpretation to the pattern language not to loose the predictability of proof commands. It is also worth mentioning that the subterm a proof command may be focused on is in general way smaller than the whole conjecture, and can usually be approximated by an even smaller pattern. So compactness is also retained.

[1] http://www.msr-inria.inria.fr/Projects/math-components/

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 the standard ones. Intuitively the `rewrite` command takes in input an equation and substitutes occurrences of the left hand side with the corresponding instance of the right hand side.

Example 1 (Simple pattern).

This example lies in the context of the algebraic hierarchy [5] 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.
```

The idea in the proof that follows 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, namely the commutativity law for addition must be applied *on the right sub-expression*. Unluckily there are many occurrences of the addition operation.

```
Lemma example1 : a + (b * c - a) + a = b * c + a.
```

The cope with the 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.

```
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 wave to underline the effect of a proof command.

```
a + (-a + b * c) + 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 also the second kind of ambiguity, *occurrence selection*, is solved. 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 traversing the conjecture syntax tree in pre visit order, finding the same instance as before. Note that, if no extra information is provided, the left hand side of the rule is the very ambiguous pattern `(_ + _)`. Unluckily the first instance encountered in pre-visit order is `(a + (b * c - a) + a)`, that is not the desired one.

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. While this may seem easy for a trivial and pretty standard lemma like `addrC` it may not be obvious in the context of a library

of the size of the Mathematical Components one, that comprises more that 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.

For the usability of any library the design of a good API a crucial point. The quality of the interface can only be validated on a large test bed. The validation phase is likely to suggest some changes to the API. When the statement of a lemma is changed its proof and the ones using the lemma are likely to break and fixing them may be very time consuming. The general approach of the SSReflect language to ameliorate this issue is to detect failures as early as possible.

For example the previous conjecture could be replaced (on purpose or by accident) by the following one:

```
a + (b * c + a) + 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 hand 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.

```
a + (a + b * c) + a = a + b * c.
```

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 manual instantiation `(addrC (x := b * c))`.

□

As mentioned in the introduction the logics 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, but this introduces an extra degree of ambiguity the user has to deal with.

Example 2 (Pattern forcing definition unfolding). In the context of the list library 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 extensionally 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 example2 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. Anyway 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 og)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`. □

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 on which subterms the rewrite rule could apply. 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 _ _]` 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 a real proof script part of the SSReflect library and was one of the motivating examples for contextual patterns, that will be introduced immediately after.

Example 3 (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 use 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 intention of the user is to rewrite the first occurrence of `X` with the `imgX` equation.

```
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 twice in the conjecture (the implication below the double bar). Unluckily 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 logics of Coq features explicit polymorphism, like system $\mathcal{F}$, so types occur as arguments of polymorphic functions even if some syntactic sugar hides them. The occurrence of `X` we are interested in is number twenty-nine.

The pattern we propose to unambiguously identify that occurrence mentions the context in which it occurs. 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 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 of script breakage the original intention of the user is clearly spelled out improving the readability of the script. □

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 [6, Sections 4.2 and 5.3]. Rewrite patterns are a strict superset of contextual patterns adding the possibility to identify all the subterms of a given context. They can be used as arguments of the SSReflect `rewrite` command, see [6, Section 7].

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

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.

In the third column we use the short *ao-fi* for "all the occurrences of the first instance". The instance is searched traversing the conjecture abstract syntax tree in pre-visit order.

⟨*c-pattern*⟩	redex	subterms affected
⟨*tpat*⟩	⟨*tpat*⟩	ao-fi of the redex
⟨*ident*⟩ `in` ⟨*tpat*⟩	subterm of ⟨*tpat*⟩ selected by ⟨*ident*⟩	the subterms identified by ⟨*ident*⟩ in ao-fi of ⟨*tpat*⟩
⟨*tpat*⟩$_1$ `in` ⟨*ident*⟩ `in` ⟨*tpat*⟩$_2$	⟨*tpat*⟩$_1$	ao-fi of the redex in the subterms identified by ⟨*ident*⟩ in ao-fi of ⟨*tpat*⟩$_2$
⟨*tpat*⟩$_1$ `as` ⟨*ident*⟩ `in` ⟨*tpat*⟩$_2$	⟨*tpat*⟩$_1$	the subterms identified by ⟨*ident*⟩ in ao-fi of ⟨*tpat*⟩$_2$[⟨*tpat*⟩$_1$/⟨*ident*⟩]

We have already seen in example 3 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
```

The proof command `rewrite [_ - b]andrC` changes the conjecture as follows because the first instance of the pattern `(_ - b)` is `(a - b)`, and this subterm has another occurrence in the right hand side of the conjecture. We underline the affected subterms.

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

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

```
a + b = f n - 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 only the right hand side only. From now on we will always underline with a straight line the sub-term focused by the context part of a pattern.

As we will describe later in Section 2.3 shortcuts for recurrent contexts can be defined by the user, making the previous pattern more compact: `set n := (_ + b in RHS)`.

We will give an example of the latter form together with the examples for ⟨*r-pattern*⟩.

The `rewrite` command supports two more patterns obtained prefixing the first two ⟨*c-patterns*⟩ with `in`. The intended meaning is that the pattern identifies all subterms of the specified context. Note that the `rewrite` command infers a redex looking at the rewrite rule. For example the `addrC` rule of example 1 gives the redex pattern `(_ + _)`.

⟨*r-pattern*⟩	redex	subterms affected
`in` ⟨*tpat*⟩	inferred from rule	ao-fi of the redex in ao-fi of ⟨*tpat*⟩
`in` ⟨*ident*⟩ `in` ⟨*tpat*⟩	inferred from rule	ao-fi of the redex in the subterms identified by ⟨*ident*⟩ in ao-fi of ⟨*tpat*⟩

The first ⟨*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 notice 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*⟩. As in example 2 this is convenient as soon as a definition hides the inferred redex.

2.2 Matching order for contextual patterns

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.

pattern	instantiation order and place for $\langle tpat \rangle_i$ and redex
$\langle tpat \rangle$	$\langle tpat \rangle$ is matched against the goal, redex is matched with the instantiation of $\langle tpat \rangle$
$\langle ident \rangle$ `in` $\langle tpat \rangle$	$\langle tpat \rangle$ is matched against the goal, then the redex is matched with the subterm of the instantiation of $\langle tpat \rangle$ identified by $\langle ident \rangle$
$\langle tpat \rangle_1$ `in` $\langle ident \rangle$ `in` $\langle tpat \rangle_2$	$\langle tpat \rangle_2$ is matched against the goal, then $\langle tpat \rangle_1$ is matched against the subterms of the instantiation of $\langle tpat \rangle_2$ identified by $\langle ident \rangle$, finally redex is matched with the instantiation of $\langle tpat \rangle_1$
$\langle tpat \rangle_1$ `as` $\langle ident \rangle$ `in` $\langle tpat \rangle_2$	$\langle tpat \rangle_2[\langle tpat \rangle_1/\langle ident \rangle]$ is matched against the goal, then redex is matched with the instantiation of $\langle tpat \rangle_1$
`in` $\langle ident \rangle$ `in` $\langle tpat \rangle$	$\langle tpat \rangle$ is matched against the goal, then the redex is matched against the subterms of the instantiation of $\langle tpat \rangle$ identified by $\langle ident \rangle$
`in` $\langle tpat \rangle$	$\langle tpat \rangle$ is matched against the goal, then the redex is matched against the instantiation of $\langle tpat \rangle$

The matching order is very relevant for predicting the instantiation of patterns. For example when matching the pattern `((_ + _) in X in (_ * X))`, the matching of the sub pattern `(_ + _)` is restricted to the sub-term identified by `X`. Take the following conjecture:

```
a + b + (a * ((a + b) * d)) = 0
```

The dash underlined sub-term would be a valid instance of `(_ + _)` but is skipped since id does not occur in the right context. In fact `(_ * X)` is matched first. The subterm corresponding to `X` is `((a + b) * d)`. Then `(_ + _)` is matched against its sub-terms and its first, and only, occurrence is underlined with a wave.

2.3 Recurrent contexts

Whilst being quite expressive, contextual patterns tends 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 allow the user to define abbreviations for common contexts, corresponding to the $\langle ident \rangle$ `in` $\langle tpat \rangle$ 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 $\langle tpat \rangle$. Then, when the system parses `[in RHS]` as an instance of `in` $\langle tpat \rangle$ it recognizes the context encoded in $\langle tpat \rangle$ and outputs the abstract syntax tree for `in` $\langle ident \rangle$ `in` $\langle tpat \rangle$.

To achieve a non ambiguous encoding of `(X in _ = X)` into a $\langle tpat \rangle$, the notation `(X in t)%pattern` generates an ill-typed $\langle tpat \rangle$. In particular it generates `(_ : fun X => t)`, that corresponds to a type cast toward an ill-formed type (types cannot begin with a λ-abstraction in the logics of Coq). Thus the output of the encoding cannot clash with a $\langle tpat \rangle$ input by the user, that must be well typed and well formed.

3 Term matching

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

Predictability has already been discussed in relation to example 2. A lemma that talks about the `map` function should affect occurrences of the `map` function only, unless specified otherwise. Nevertheless the most characterizing feature of the logics 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 a computable definition of addition.

Performance is a main concern when one deals with large conjectures. To take advantage of the computational notion of comparison the logics offer, one could be tempted to try to match the pattern against any subterm, even if the subterm shares no similarity with the pattern itself. An higher order matching procedure could find that the pattern actually matches up to computation a subterm, and instantiate the pattern variables accordingly. 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 verbatim equal to the head constant (the *key*) of the pattern. Arguments of the key are matched using the standard higher order matching algorithm of Coq, thus taking computation into account.

Take for example the following 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 sub-terms 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. Note that ⟨*c-pattern*⟩s can override the redex inferred looking at the rewrite rule. In that case the term matched may look arbitrarily different from the inferred redex, but will be higher order matched against it. Thus the user can, on demand, take advantage of the powerful notion of equality up to computation offered by the logics of Coq.

3.1 Gadgets

To adhere to the keyed matching discipline, that is different from the standard one implemented in Coq, the SSReflect extension implements its own stricter matching algorithm that eventually piggy backs on the standard Coq's one. This gave us the opportunity to tune it towards our needs, adding some exceptions to ease the handling of unit types and abstract algebraic structures.

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 redex inferred from the left hand side 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 less 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, only subterms of type `'I_1` are matched by `(unkeyed _)`.

```
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 logics of Coq can directly represent dependently typed records [11], 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 represent abstract algebraic structures [12,5,13,3]. For example one can define a Monoid as a record with three fields: a type `mT`, a binary operation `mop` on `mT` and the associative property for `mop`.

```
Structure Monoid := {
  mT : Type;
  mop : mT -> mT -> mT;
  massoc : associative mop }.
```

```
mT (M : Monoid) : Type
mop (M : Monoid) : mt M -> mt M -> mt M
massoc (M : Monoid) (x y z : mt M) :
   mop M x (mop M y z) = mop M (mop M x y) z
```

Constants `mT`, `mop` and `massoc` are defined as projections for the corresponding record fields.

If we look at the statement of any lemma equating Monoid expressions we notice 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 any result proved in the abstract setting applies 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 Coq system can be instrumented to exploit the fact that there exists a canonical Monoid over lists and is thus able to higher order 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 are verbatim different.

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.

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 in general inefficient, but for small conjectures it is anyway convenient, in particular 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.

Example 4 (Motivating example).

```
Lemma example3 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 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 head constant is on the left hand side of the conjecture. Since the definition of multiplication for natural numbers is computable Coq compares as equal `(2 * m)` and `(m + (1 * m))`. Thus the redex is matched with `(n + (m + (1 * m)))` 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 ameliorate this issue, the term pattern matching algorithm performs two phases. In the first one, called first order matching, no reduction takes place. Moreover 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 one is driven by the head constant of the pattern as explained before and allows full reduction on its arguments.

The search of the occurrences of the instantiated pattern 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)`.

Allowing conversion on the arguments is necessary to consider as occurrences subterms that have different invisible and automatically inferred arguments. XXX not sure it's interesting

4 Contextual and Rewrite Pattern Implementation

XXX cut this part The implementation amounts to 1300 lines of OCaml code. In the following we show the main implementation tricks in pseudo code. The actual code is forced to fold around many more accumulators, to honor some flags and to properly report a wide variety of errors. Nevertheless the following code takes into account occurrence numbers. Although not encouraged, SSReflect supports them for experimentation, debugging purposes and as the last resort when dealing with inductive family elimination. XXX should this be mentioned earlier?

```
type pat =
  | T of term
  | In_T of term
  | X_In_T of term * term | (*...*)
```

```
type subst = term -> int -> term
type chain = term -> int -> k:subst -> term
type conclude = unit -> term
```

In contrast with the grammar presented in Section 2.1 the syntax tree `pat` merges ⟨*c-pattern*⟩ and ⟨*r-pattern*⟩.

The function `mk_tpat_matcher` compiles a term pattern `p` into a continuation passing style substitution `chain` and an assertion function `conclude`, to be called at the end of the matching process. The substitution maps an input term eventually calling the `k` continuation on subterms matching the pattern. The substitution function is stateful and shares with the assertion function the set of occurrences to be substituted as well as the instance of the pattern `instance`. This design choice makes it easy to chain continuations, that may be called multiple times, without threading around an accumulator for each of them. The pattern is instantiated only once, when the substitution function is called the first time, using the function given to `do_once` (line 11).

```
1  let mk_tpat_matcher (occ : int set) (p : term) : chain * conclude =
2    let n_occ = ref 0 and occ = ref occ in
3    let subst_occ () = incr n_occ;
4      if mem !n_occ !occ then (occ := !occ - !n_occ; true) else false in
5    let instance = ref None in
6  (fun (c : term) (h : int) ~k ->
7    do_once instance (fun () -> try match_FO p c with _ -> match_HO p c);
8    let key, k_args = splay_app (assert_done instance) in
9    let rec subst_loop h c = if !occ = empty then c else
10     let f, f_args = splay_app c in
11     let ok, args, rest = is_instance (key, f) (k_args, f_args) in
12     if ok then
13       let f' = mkApp (f, args) in
14       mkApp ((if subst_occ () then k f' h else f'), map (subst_loop h) rest)
15     else
16       mkApp (map_term ((+) 1) subst_loop h f, map (subst_loop h) f_args) in
17   subst_loop h c),
18 (fun () ->
19   if !occ = empty || !occ = all_occ then assert_done instance
20   else error "Not all specified occurrences were substituted")
```

The `match_FO` and `match_HO` functions were detailed in the previous section and their code is not reported. The `subst_loop` calls the continuation `k` on any occurrence of the term matched by `p` in the occurrence set (see `subst_occ` line 3). The search for occurrences is performed using the head constant, `key` of the instance of `p` (line 8). Line 11 checks if the current subterm head constant matched verbatim the key, and eventually splits the arguments into the ones corresponding to the arguments of the key and the remaining ones (line 6). Note that arguments are compared allowing conversion, while `f` and `key` are compared verbatim. If the sub-term is not an occurrence the `map_term` iterator continues the conjecture traversal (line 16). The `h` argument threaded around is incremented when traversing binders and is useful to implement the `rewrite` command (see the last code snippet). If the current subterm is an instance and its number is in the occurrence set specified by the user it is replaced by the term returned by the continuation.

On top of the term pattern matcher, the `eval_pat` function applies the substitution `s` on the sub-terms of `c` identified by `p`. If no pattern is given, the substitution is fired immediately on the whole term. If a simple pattern is given, then a substitution function `find_t` is created and called passing `s` as the continuation. At last, `end_t` is called to check that all the occurrences specified were substituted to detect errors where the user specifies more occurrences than the ones actually present. In this particular case the returned term, the instance of the pattern, is discarded.

```
1  let eval_pat (c : term) (occ : int set) (p : pat option) (s : subst) : term =
2    match p with
3    | None -> s c 1
4    | Some (T t) ->
5      let find_t, end_t = mk_tpat_matcher occ t in
6      let c = find_t c 1 ~k:s in let _ = end_t () in c
7    | Some (X_In_T (x, t)) ->
```

```
8       let find_t, end_t = mk_tpat_matcher all_occ t in
9       let find_x, end_x = mk_tpat_matcher occ x in
10      let c = find_t c 1 (fun t_occ h ->
11        let carved_t_occ, body_x = carve x t_occ p in
12        find_x carved_t_occ h (fun _ -> s body_x)) in
13      let _ = end_x () in let _ = end_t () in c
14    (*...*)
```

The more complex pattern of the form `(x in t)` creates two substitutions, for `t` and for `x`, passing the latter as a continuation to the former. `x` here is just a matching variable occurring in `t`, and the pattern for `x` matches only occurrences of that variable. Finding an occurrence for `t`, called `t_occ` also finds an instantiation for `x` (line 10). The `carve` function puts back the matching variable `x` in its place, saving in `body_x` its instantiation (line 11). Then `find_x` calls the continuation `s` only on the occurrences of `x` but passing to it the save instantiation for `x`. Note that this is the expected behaviour. A pattern like `(X in f _ X)` must focus on the second argument of `f`, and skip the first one even if the two arguments are accidentally the same. The evaluation of other patterns is very similar and thus omitted.

The last code snippet shows how `rewrite` calls the pattern matching facility. After having examined the rewrite rule and having inferred a term pattern for the left hand side (line 2), it examines the user provided pattern to generate a substitution function and its conclusion counterpart. If the pattern is absent or gives no redex, the one inferred from the rule is used for the substitution (line 6). Otherwise the substitution just checks that `lhs` higher order matches the instance of the redex specified by the pattern (line 8).

```
1   let rewrite dir occ pat rule c =
2     let lhs, lhs_ty = examine_rule dir rule in
3     let subst, conclude = match pat with
4     | Some (In_T _ | In_X_In_T _) | None ->
5         let find_lhs, end_lhs = mk_tpat_matcher occ lhs in
6         find_lhs ~k:(fun _ h -> mkRel h), end_lhs
7     | Some(T _ | X_In_T _ | E_As_X_In_T _ | E_In_X_In_T _) ->
8         (fun c h -> assert (eq_conv c lhs); mkRel h), (fun () -> lhs) in
9     let rew_predicate = eval_pat c occ pat subst in
10    let _ = conclude () in
11    elim (mkLam ("_match_", lhs_ty, rew_predicate)) rule
```

Note that both substitutions replace the term to be rewritten with the h-th De Bruijn index (`mkRel h`). Thus the conjecture `c` is used to generate a predicate for the higher order elimination rule for equality binding to a variable called `"_match_"` the subterms of the conjecture to be rewritten. Note that no lifting (in De Bruijn sense) has to be performed on `rew_predicate` since Coq uses a locally nameless encoding of bound variables.

5 Related works

The first comparison that has to be made is with the standard Coq mechanism to identify sub expressions. As of version 8.3, the Coq main mechanism is the `pattern` command ([2, Section 6.3.3]) that β-expands the conjecture with respect to a term the user has to spell out completely. The future 8.4 release will add support for patterns, but the discipline that will be followed to match these patterns is still unclear. Coq's `rewrite` command employs a particular algorithm to find the first instance of the pattern inferred looking at the rewrite rule, but even this search is not key driven.

Matita is a lightweight Coq clone that integrates some technologies born in the context of MoWGLI project that aimed at putting the Coq library on the web. It features a MathML rendering widget (see [9, Section 5.1.1]) that supports the visual selection of meaningful sub expressions. The user can thus focus any proof command using the mouse (see [1, Section 3.2]). This action is recorded in the proof script with a rather verbose representation of the paths from the root of the conjecture to the selected subterms. In addition to visual selection the user can specify a pattern that is higher order matched, following an unkeyed discipline. In the development version a keyed matching discipline similar to the one of SSReflect has been adopted at least for the `rewrite` and `elim` proof commands. In the following example the `rewrite` proof command is focused on some subterms of the hypothesis `H` and of the conclusion. In the pattern `(? + ?)` question marks correspond to wild cards, and the following patterns `%` identifies the root of a focused subterm. Thus the pattern `(? + ?)` is searched in the second argument of `H` and in the last one of the conclusion. While the pattern `(? + ?)` is input by the user, the rest is generated in response to a mouse click.

```
rewrite E in match (? + ?) in H : ??%? |- ???%
```

The Isabelle prover [10] implements a framework on top of which different logics and proof languages are developed. The most used combination is higher order logics and the declarative proof language Isar. In this setting some of the complexity introduced by the logics of Coq disappear. 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. Nevertheless the lower level of the system deals with what is called "term surgering", see for example the `subst` command ([14, Section 9.2.2]). In this setting occurrences are identified by a term pattern `tpat` and a list of occurrence numbers. The unification engine is asked to unify `(_ tpat)` with the current conjecture following Huet's higher order unification algorithm. The algorithm synthesizes a λ-expression for the unknown head symbol, and in this process it emits some occurrences of the bound variable in a precise order, standing for subterms unified with `tpat`. Occurrence numbers supplied to the `subst` command refer to occurrences of the bound variable, following their creation order. In practice the algorithm searches occurrences of `tpat` in the conjecture up to $\alpha\eta\beta$-equivalence.

6 Conclusion

This paper presents the language of patterns adopted by the SSReflect proof shell extension for the Coq system. The language was first introduced in SSReflect version 1.3 in March 2011 mainly to improve the effectiveness of the rewrite proof command. In version 1.4 the pattern language was improved and made available to all language constructs that have to identify sub expressions 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 project during the last four years, and the implementation has been validated by roughly one year of intense usage. As of today the Mathematical Components library comprises 113.384 lines of code, of which 34.077 comprise a rewrite statements. Of these 2.095 contain at least one pattern.

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. This will allow proof commands available in other Coq extensions to benefit from the same pattern language. A possible application would be the AAC Coq extension, allowing to automate proofs dealing with associativity and commutativity. In fact in [4] Braibant and Pous identify the need for a linguistic construct to select sub expressions other than occurrence numbers.

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

References

1. 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.
2. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions*. Springer-Verlag, 2004.
3. Thomas Braibant and Damien Pous. An efficient coq tactic for deciding kleene algebras. In *ITP*, pages 163–178, 2010.
4. Thomas Braibant and Damien Pous. Tactics for reasoning modulo AC in Coq. In *Proceedins of CPP'11*, LNCS, 2011.
5. François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.
6. Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A small scale reflection extension for the Coq system*. INRIA Technical report, http://hal.inria.fr/inria-00258384.
7. Geroges Gonthier. Formal proof – the four color theorem. *Notices of the American Mathematical Society*, 55:1382–1394, 2008.
8. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
9. Luca Padovani. *MathML Formatting*. PhD thesis, University of Bologna, February 2003. Technical Report UBLCS 2003-03.
10. Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.
11. Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.
12. Claudio Sacerdoti Coen and Enrico Tassi. Working with mathematical structures in type theory. In *Proceedins of TYPES 2007*, volume 4941/2008 of *LNCS*, pages 157–172. Springer-Verlag, 2007.
13. Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'*, 21:1–31, 2011.
14. M. Wenzel. The isabelle/isar reference manual.