Pattern-based Subterm Selection in Isabelle

Lars Noschinski and Christoph Traut

Institut für Informatik, Technische Universität München, Germany

Abstract. This article presents a pattern-based language designed to select (a set of) subterms of a given term in a concise and robust way. Building on this language, we implement a single-step rewriting tactic in the Isabelle theorem prover, which removes the need for obscure "occurrence numbers" for subterm selection.
The language was inspired by the *language of patterns* of Gonthier and Tassi, but provides an elegant way of handling bound variables and a modular semantics.

1 Introduction

Proof assistants, sometimes called *interactive theorem provers*, are tools designed to assist human users in the process of writing formal mathematical proofs. They allow users to write their proofs in a formal language, where every step of the proof is verified by the assistant and checked for logical errors. Two well-known proof assistants are *Isabelle* and *Coq* [4].

A common proof method is equational reasoning, either rewriting a term into a normal form or a manual application of equations. The latter is often used for proof exploration or when automated methods do not yield the desired result. As a given equation can often be applied to different subterms of a term, the user needs a way to select the desired subterm. There are two obvious approaches to subterm selection which are implemented both in Coq and Isabelle. As an example, consider the equation

$$(a + b) + (c + d) = e + f$$

and the commutativity rule `add_commute`: $?x + ?y = ?y + ?x$ (we use ? to mark universally quantified variables). There are four positions where the commutativity rule can be applied. If we want to rewrite the subterm $c + d$ to $d + c$, current Isabelle versions offer basically three options:

```
– subst add_commute[of "c" "d"]
– subst add_commute[where x="c" and y="d"]
– subst (3) add_commute
```

The first two examples instantiate the variables in the rule, so that it is only applicable for the $c + d$ subterm. The third example instructs Isabelle to skip the first two subterms matching $?x + ?y$ and rewrite the third (which happens to be $c + d$).

Instantiation can only partially resolve the ambiguities. Also, instead of specifying directly what to rewrite, the user needs to know the names or the order in which the variables occur in the rule. On the other hand, occurrence numbers are hard to read, as it is often not obvious which subterm has which occurrence number.

As discussed by Gonthier and Tassi [2], both of these approaches are insufficient for writing maintainable proofs: While they are expressive enough to select any subterm, they lack the expressiveness for capturing the developers intent. As a result, they break easily when unrelated parts of the proof change. Gonthier and Tassi developed a *language of patterns* [2] as an alternative approach. The idea is that (for rewriting) subterm selection already is done by a pattern: The left hand side of the rule. Instead of restricting this indirectly (by instantiating the rule), the pattern is given explicitly. If this does not suffice to unambiguously select a subterm, an additional context pattern can be used.

In this article, we present a new language for subterm-selection building upon the ideas of Gonthier and Tassi. In particular, our language adds the ability to match on bound variables in a robust way and provides an easy to understand and flexible syntax with formal (though not formalized) semantics. We implemented this language in the Isabelle theorem prover[1].

This article starts by introducing basic notation in Section 2. In Section 3 we present a set of combinators for subterm selection and describe how to use them for rewriting in Section 4. Section 5 shows some improvements on real-world examples before we conclude with Section 6.

2 Preliminaries

Isabelle's term language is the language of the simply typed lambda calculus, where variables bound by lambda abstractions are represented by de Bruijn indices [1]. In this article, we use the following simplified and untyped term structure when talking about Isabelle terms. For an infinite set of variable names $\mathcal{V}$, the set of Isabelle terms $\mathcal{T}$ is inductively defined as follows:

$$\begin{array}{cc} \mathcal{V} \subseteq \mathcal{T} & x \in \mathcal{V} \wedge t \in \mathcal{T} \implies \Lambda_x t \in \mathcal{T} \\ t_1, t_2 \in \mathcal{T} \implies t_1\ t_2 \in \mathcal{T} & n \in \mathbb{N}_0 \implies \mathsf{B}_n \in \mathcal{T} \end{array}$$

$\mathcal{V}$ are the free variables and juxtaposition is application. The variable name x at the lambda abstraction $\Lambda_x t$ carries no semantics, but serves as a note to the user. Two terms differing only in the variable attached to the Λ symbols are considered equal (α-equivalence). The *bound variable* B_n refers to the $n+1$-th lambda above it, i.e. in $\Lambda_x(f(\Lambda_y \mathsf{B}_1)\mathsf{B}_0)$ both bound variables are bound by the outermost abstraction. If n is greater than the number of abstractions above it, B_n is called a *loose bound variable*.

As in Isabelle, we usually hide the de Bruijn indices in our presentation. For example, the term $\Lambda_x(f(\Lambda_y \mathsf{B}_1)\mathsf{B}_0)$ will be written as $\lambda x.\ f\ (\lambda y.\ x)\ x$. If a term

[1] https://www21.in.tum.de/~noschinl/Pattern-2014/

contains loose bounds, we index them by the number of missing abstractions:

$$f\ \mathsf{B}_1(\lambda c.\ \mathsf{B}_2)\ \mathsf{B}_0 \text{ corresponds to } f\ \mathsf{L}_2(\lambda c.\ \mathsf{L}_3)\ \mathsf{L}_1$$

By $\#\mathsf{L}(t)$ we denote the number of different loose bound variables in the term t.

A pattern is a term which might also include the special wildcard symbol _. A term *matches* a pattern, if there are terms, such that replacing the wildcards by these terms, term and pattern become equal.

A position is a word over $\Sigma = \{a, l, r\}$, describing a subterm of a term. The empty word ε refers to the term itself, a to the term under an abstraction, and l and r to the left resp. right term of an application. We write $Pos(t)$ for the set of positions of a term t, pq for the concatenation of positions p and q, and $t|_p$ for the subterm of t at position p.

3 Combinators for Subterm Selection

If we abstract from rewriting, our problem can be phrased as follows: Given a term, find a language to select the desired subterms (or rather, their positions). The language should be expressive enough to select any singleton set. It should allow for a concise description and be robust.

To this end, we describe a set of combinators, operating on sets of term/-position pairs, i.e., functions of the type $\mathcal{P}(\mathcal{T} \times \Sigma^*) \to \mathcal{P}(\mathcal{T} \times \Sigma^*)$.

When giving examples, we write for example $S \simeq \boxed{a+b} + c$ to denote that the combinator function S applied to $M = \{((a+b+c), \varepsilon)\}$ selected the subterm $a+b$ at the marked position. To denote that not only $a+b$, but also all its subterms were selected, we write $S \simeq \underline{\underline{\boxed{a+b}}} + c$ instead.

A rewrite rule can only be applied at a position which matches its left-hand side t, so any rewriting method implicitly works as a filter, selecting only the positions where the rule can be applied. This is expressed by the following combinator:

Definition 1 (Term combinator).

$$\mathcal{S}_{\mathsf{term}}(t) = M \mapsto \{(s,p) \mid (s,p) \in M \land s \textit{ matches } t\}$$

As a second combinator, we introduce a way of selecting all subterms.

Definition 2 (Subterm combinator).

$$\mathcal{S}_{\mathsf{in}} = M \mapsto \{(s|_q, pq) \mid (s,p) \in M \land q \in Pos(s)\}$$

Now, composing these combinators gives us a way of selecting all positions below a certain subterm:

Example 1.

$$\begin{aligned}
\mathcal{S}_{\mathsf{term}}(\_ * \_) &\simeq (a+b)*c+d \\
\mathcal{S}_{\mathsf{in}} &\simeq \underline{\underline{\boxed{(a+b)*c+d}}} \\
\mathcal{S}_{\mathsf{term}}(\_ * \_) \circ \mathcal{S}_{\mathsf{in}} &\simeq \boxed{(a+b)*c}+d \\
\mathcal{S}_{\mathsf{in}} \circ \mathcal{S}_{\mathsf{term}}(\_ * \_) \circ \mathcal{S}_{\mathsf{in}} &\simeq \underline{\underline{\boxed{(a+b)*c}}}+d \\
\mathcal{S}_{\mathsf{term}}(a + \_) \circ \mathcal{S}_{\mathsf{in}} \circ \mathcal{S}_{\mathsf{term}}(\_ * \_) \circ \mathcal{S}_{\mathsf{in}} &\simeq \boxed{(a+b)}*c+d
\end{aligned}$$

Unfortunately, these combinator do not yet allow us a unique selection in every case. For example, we are still missing a combinator S such that $S \simeq \boxed{a} = a$. One way to deal with this is to specify the position of the left a directly, i.e., as lr (remember that $a = a$ corresponds to $(= a)\ a$, when writing $=$ not as an infix).

Definition 3 (Position combinator).

$$\mathcal{S}_{\mathsf{pos}}(q) = M \mapsto \{(s|_q, pq) \mid (s, p) \in M \wedge q \in \mathit{Pos}(s)\}$$

Specifying the offset as a position is not very readable, so we mark the position we are interested in by a special symbol in the pattern: If a pattern t contains a single hole symbol $\square$, then $\mathit{pos}_{\square}(t)$ is the position of the hole in the term, i.e., $\mathit{pos}_{\square}(\square = \_) = lr$. Similar to _, it matches every term.

Consider the following term. How can we select the subterm $f\ b$?

$$\lambda a\ b.\ [f\ a, (\lambda c.\ a), f\ b] \tag{1}$$

A possible solution would be to select the second element of this list, but this might not capture our intent (and might be very cumbersome for larger terms). The obvious combinator $\mathcal{S}_{\mathsf{term}}(f\ b)$ does not work, as a and b are free variables in the pattern and bound variables in the term. The deeper problem is that bound variables only exist as soon as we descend into the term. There are various reasons why guessing a bound variable based on its name is not a viable alternative: Different bound variables with the same name might occur either in parallel (e.g., $(\lambda x.\ x) \circ (\lambda x.\ x)$) or nested (e.g., $\lambda x.\ f\ (\lambda x.\ x)$). Moreover, the terms from which we want to select subterms are usually not input by the user directly, but are the result of some earlier proof steps. As Isabelle considers terms modulo α-equivalence, this means the names might change in unexpected ways.

Hence, to refer to bound variables, we introduce new names when we descend into a term. The $\mathcal{S}_{\mathsf{term}}$ and $\mathcal{S}_{\mathsf{pos}}$ combinators allow us to control where to descend:

Example 2.

$$\mathcal{S}_{\mathsf{pos}}(\mathit{pos}_{\square}(\lambda x\ y.\ \square)) \circ \mathcal{S}_{\mathsf{term}}(\lambda x\ y.\ \square) \simeq \lambda a\ b.\ \boxed{[f\ a, (\lambda c.\ a), f\ b]}$$

The selected term is $[f\ \mathsf{L}_2, (\lambda c.\ \mathsf{L}_2), f\ \mathsf{L}_1]$.

If we replace the bound variables L_1 and L_2 by free variables, we can refer to them in the $\mathcal{S}_{\mathsf{term}}$ combinator. The $\mathcal{S}_{\mathsf{bind}}$ combinator introduces free variables.

Definition 4 (Binding bound variables). *The bind combinator $\mathcal{S}_{\mathsf{bind}}$ will replace the innermost loose bound variables by fresh variables:*

$$\mathcal{S}_{\mathsf{bind}}(v_1, \ldots, v_n) = M \mapsto \{(s', p) \mid (s, p) \in M \wedge n \leq \#\mathsf{L}(s) \\ \wedge s' = subst_{\mathsf{L}}(s, [v_1, \ldots, v_n])\}$$

where $subst_{\mathsf{L}}$ replaces L_1 by v_n, L_2 by v_{n-1}, …, L_{n-1} by v_2 and L_n by v_1.

Example 3. For the term selected in Example 2, this works as follows:

$$\begin{aligned} \mathcal{S}_{\mathsf{bind}}(x, y) &\text{ selects } [f\ x, (\lambda c.\ x), f\ y] \\ \mathcal{S}_{\mathsf{bind}}(x) &\text{ selects } [f\ \mathsf{L}_2, (\lambda c.\ \mathsf{L}_2), f\ x] \\ \mathcal{S}_{\mathsf{bind}}(x, y, z) &\text{ selects nothing} \end{aligned}$$

So the following function lets us finally select the desired subterm $f\ b$ of Equation 1:

$$\mathcal{S}_{\mathsf{term}}(f\ y) \circ \mathcal{S}_{\mathsf{bind}}(x, y) \circ \mathcal{S}_{\mathsf{pos}}(pos_{\square}(\lambda x\ y.\ \square)) \circ \mathcal{S}_{\mathsf{term}}(\lambda x\ y.\ \square)$$

Note that we used the same pattern both for $\mathcal{S}_{\mathsf{pos}}$ and $\mathcal{S}_{\mathsf{term}}$ and that the use of $\mathcal{S}_{\mathsf{bind}}$ only gives predictable results if we know where in the term we actually are. Usually, this is the case after selecting with both $\mathcal{S}_{\mathsf{term}}$ and $\mathcal{S}_{\mathsf{pos}}$ first. So we combine the three combinators into a single one.

Definition 5 (Matching combinator). *The matching combinator selects the subterms according to a pattern t. If t contains a single hole, it descends to the position of the hole and replaces all bound variables, which become loose in the descent, by free variables named like the corresponding bound variables in t.*

$$\mathcal{S}_{\mathsf{match}}(t) = \mathcal{S}_{\mathsf{bind}}(binds(t, pos_{\square}(t))) \circ \mathcal{S}_{\mathsf{pos}}(pos_{\square}(t)) \circ \mathcal{S}_{\mathsf{term}}(t)$$

By $binds(t, p)$ we denote the list of variables which are bound between the root and p in t. E.g., for $t = \lambda a\ b.\ f\ a\ b$ and $p = pos_{\square}(\lambda a\ b.\ \square)$, we have $binds(t, p) = (a, b)$.

The matching combinator takes the variables it introduces directly from the lambda abstractions. As this term was explicitly entered by the user, the variables were not subject to renamings and produce the expected result. When using the bind combinator, one should take care of always using fresh variables (otherwise, different variables in the term might be identified). With the usual Isabelle naming mechanisms we will be able to hide this detail from the user.

We have described a basic set of combinators for navigating in a term. In Isabelle, the subgoals we are rewriting usually have a special structure:

$$\bigwedge x_1\ x_2 \ldots x_m.\ P_1 \implies P_2 \implies \ldots \implies P_n \implies Q$$

Here, the x_i are universally quantified variables, the P_i are premises and Q is the conclusion. When new universally quantified variables are introduced during a proof, they are added at the end of the list. This motivates the introduction of combinators for selecting these parts of a goal.

Definition 6 (Combinators for the goal structure). *For a term t of the structure given above, $\mathcal{S}_{\mathsf{concl}}$ will select the term Q and $\mathcal{S}_{\mathsf{asm}}$ will select the terms $P_1, \ldots, P_n$ (with the appropriate positions). For matching the universally quantified variables, it is important to remember that the list is extended from the right, so usually one wants to talk about a suffix of this list (cf. Isabelle's `rename_tac`). So, if $k \leq m$, $\mathcal{S}_{\mathsf{for}}(y_1, \cdots, y_k)$ binds (as in $\mathcal{S}_{\mathsf{bind}}$) x_m to y_k, x_{m-1} to y_{k-1}, and x_{m-k+1} to y_1 and selects the term $P_1 \implies P_2 \implies \ldots \implies P_n \implies Q$.*

Example 4. We demonstrate the $\mathcal{S}_{\mathsf{for}}$ combinator:

$$\mathcal{S}_{\mathsf{for}}(a, b) \simeq \bigwedge x_1 x_2 x_3. \boxed{x_1 \leq x_2 \implies x_2 \leq x_3 \implies x_1 \leq x_3}$$

In the matched term, the bound variables x_2 and x_3 are replaced by free variables a and b. x_1 remains a loose bound variable:

$$\mathsf{L}_3 \leq a \implies a \leq b \implies \mathsf{L}_3 \leq b$$

The bound variables a and b can then be used with the $\mathcal{S}_{\mathsf{match}}$ combinator:

$$\begin{aligned} &\mathcal{S}_{\mathsf{term}}(a \leq b) \circ \mathcal{S}_{\mathsf{in}} \circ \mathcal{S}_{\mathsf{for}}(a, b) \\ \simeq &\bigwedge x_1\ x_2\ x_3.\ x_1 \leq x_2 \implies \boxed{x_2 \leq x_3} \implies x_1 \leq x_3 \end{aligned}$$

Example 5. The difference between $\mathcal{S}_{\mathsf{for}}(a, b)$ and $\mathcal{S}_{\mathsf{match}}(\bigwedge a\ b.\ \square)$ is that the former matches the rightmost and the latter the leftmost of the universally quantified variables (note that $\bigwedge a\ b.\ t$ is a shorthand for $\bigwedge a.\ \bigwedge b.\ t$).

$$\mathcal{S}_{\mathsf{for}}(a, b) \simeq \bigwedge x\ y\ z. \boxed{t\ x\ y\ z}$$

$$\mathcal{S}_{\mathsf{match}}(\bigwedge a\ b.\ \square) \simeq \bigwedge x\ y \boxed{\bigwedge z.\ t\ x\ y\ z}$$

where the selected term for the former is $t\ \mathsf{L}_3\ a\ b$ and the selected for the latter is $\bigwedge z.\ t\ a\ b\ z$.

4 Rewriting

In the last section we introduced a combinator language for subterm selection. We now apply this language to the implementation of a proof method `pat-subst` for single-step rewriting. We start by introducing a user-facing syntax for subterm selection, building on the previously introduced combinators.

```
<atom>    ::= <term> | concl | asm | prop
<pattern> ::= (in <atom> | at <atom> | for <names>)
                [<pattern>]
```

The atom `<term>` is parsed into a term t. We define the semantics of atoms as follows:

$$[\![t]\!] = \mathcal{S}_{\mathsf{match}}(t) \qquad [\![\texttt{concl}]\!] = \mathcal{S}_{\mathsf{concl}} \qquad [\![\texttt{asm}]\!] = \mathcal{S}_{\mathsf{asm}} \qquad [\![\texttt{prop}]\!] = I$$

Based on this, the semantics of patterns are:

$$\begin{aligned} [\![\texttt{in}\ a]\!] &= \mathcal{S}_{\mathsf{in}} \circ [\![a]\!] \\ [\![\texttt{at}\ a]\!] &= [\![a]\!] \\ [\![\texttt{for}\quad (i_1 \ldots i_n)]\!] &= \mathcal{S}_{\mathsf{for}}(i_1, \ldots, i_n) \\ [\![\mathit{pat}_1\ \mathit{pat}_2]\!] &= [\![\mathit{pat}_1]\!] \circ [\![\mathit{pat}_2]\!] \end{aligned}$$

To select subterms of some term t with a pattern p, $[\![p]\!]$ is applied to an initial set $M(t)$ to obtain the set of selected subterms $[\![p]\!](M(t))$. As in our example syntax, we set $M(t) = \{(t, \varepsilon)\}$.

This allows full control over the subterm selection. From our experience with the existing `subst` tactic for single-step rewriting in Isabelle, rewriting in the conclusion is the most common case. For this reason, our implementation automatically appends `in concl` if a pattern ends with a term atom.

Example 6.

$$\begin{array}{ll}
\texttt{at "a + \_"} & \simeq (\boxed{a+b}) + c \\
\texttt{at "a + \_" at prop} & \simeq (a+b) + c \\
\texttt{in "\_ * c"} & \simeq \underline{\boxed{(a+b) * c}} + d \\
\texttt{at "\_ + \_" in "\_ * c"} & \simeq (\boxed{a+b}) * c + d \\
\texttt{at "x + 1" in "f \_"} & \simeq f\ (\boxed{x+1}) = g\ (x+1) \\
\texttt{at "}\square\texttt{ = \_"} & \simeq \boxed{x} = x \\
\texttt{at "a + \_" in "f }\square\texttt{ + f \_"} & \simeq f\ ((\boxed{a+b}) + c) + f\ ((a+b) + c) \\
\texttt{at "v+1" in "(}\lambda\texttt{v. }\square\texttt{) }\equiv\texttt{ \_"} & \simeq (\lambda x.\ (\boxed{x+1}) + c) \equiv (\lambda y.\ (y+1) + c)
\end{array}$$

4.1 Conversions

For a long time, Isabelle has provided the `subst` proof method for single-step rewriting. This proof method uses the rule

$$\frac{f \equiv g \qquad x \equiv y}{f\ x \equiv g\ y}$$

to rewrite a term in a single derivation step: x is chosen as a subterm matching the left hand side of the applied equation, f as the surrounding context. Then the equation is applied to x, resulting in y, and g is the same as f. Unfortunately, this approach is incompatible with our handling of bound variables. Instead, our implementation builds upon Isabelle's *conversions*.

A conversion is a function that takes a term t and produces a theorem of form $t \equiv t'$. Apart from the basic conversion which rewrites a term at its root, the combinators `abs_conv`, `arg_conv`, and `fun_conv` are of prime interest to us: These combinators take a conversion and apply it to the term under an abstraction, the argument part of an application, and the function part of an application, respectively. Figure 1 depicts the rules implemented by these conversions. When descending below the abstraction, the `abs_conv` conversion automatically invents a fresh free variable, based on the name of the bound variable. A small modification to this function and delayed parsing of the pattern terms ensure that the name presented to the user is exactly the name of the bound variable (without introducing variable clashes).

$$\frac{t_1[x/s] \equiv t_2[x/s]}{(\lambda s.\ t_1) \equiv (\lambda s.\ t_2)}$$

(a) `abs_conv`

$$\frac{s \equiv t}{f\,s \equiv f\,t}$$

(b) `arg_conv`

$$\frac{f \equiv g}{f\,s \equiv g\,s}$$

(c) `fun_conv`

Fig. 1: Rules implemented by the conversion combinators. By $t[x/s]$ we denote substitution of x for s in t . In (1a), x must not be free in t_1 and t_2.

Note that the conversions from Figure 1 functions directly relate to subterm positions: `fun_conv` corresponds to l, `arg_conv` to r, and `abs_conv` to a. The position ϵ can be represented by the identity function I. Also, concatenation of positions corresponds to function composition.

In our semantics, we represent subterms by pairs of term and position. In our implementation, we represent each of these positions by a conversion combinator, i.e., a function of type $conv \rightarrow conv$, where $conv$ is the type of conversions. During subterm selection, we descend into terms, while updating the current position by composing the appropriate function with it. This implicit representation of the position makes for a straight-forward implementation and avoids having to descend into the term multiple times.

In contrast to the old `subst` implementation, Isabelle's conversions cannot deal with conditional equations. Support for conditional equations is an important feature, in particular for proof exploration. Fortunately, this problem is not inherent to our approach and can be rectified by using a more general variant of conversions.

Manual instantiation There are rewrite rules that introduce a new variable, one simple example being $0 = 0 * ?a$. Another example would be rules for adding an invariant annotation to a loop in program verification. In Example 7, we will show a simple example of such a rule.

Such a variable will not be instantiated automatically (as there is no indication what its value should be), so usually we need to instantiate it manually. If its value does not depend on any bound variable, the variables can be instantiated before rewriting. If the value depends on a bound variable, this is not possible; the complications being the same as for subterm selection. Currently, Isabelle provides no easy way around this problem.

However, in our implementation, the use of abs_conv ensures that all bound variables under which we descended were replaced by free variables. So, at the point where we apply the rewriting, we will never need to instantiate the term with any bound variables, but with the associated free variables instead. The same technique that allows us to match bound variables now allows us to instantiate the theorem with bound variables.

Our syntax for instantiation during term rewriting mirrors the syntax of the where attribute, which can be used to instantiate theorems directly. The where keyword is followed by a list of instantiations, which are pairs of variable name and term.

We demonstrate the use of this feature on the while combinator from Isabelle's library.

Example 7. The following program implements multiplications by addition, which in turn is implemented as repeated increment.

```
while (λ(i, x). i < a)
  (λ(i::nat, x::nat).
    let
      (_, x') = while (λ(j, _). j < b)
        (λ(j::nat, x). (j + 1, x + 1))
        (0, x)
    in (i + 1, x'))
  (0, 0)
```

Proofs about the while combinator usually use the relevant introduction rule, which requires specifying the loop invariant by hand. So a goal involving while needs to be decomposed by hand, before automated tools can be applied. By using an annotated variant of while, one can write an introduction rule picking up the invariant from the annotation.

```
definition
  "while_inv (I :: 'a ⇒ bool) c b s :: 'a ≡ while c b
     s"
```

Now, assume we wanted to add the invariant λ(j, x).x = j + a * i to the inner while loop, where the variable i is the variable bound in the body of the outer while loop. For this, we need to substitute while_inv for while and instantiate the schematic variable ?I with the invariant, using the following command:

```
apply (
  patsubst in "while _ (λ(i, _). □) _"
  while_inv_def[symmetric]
  where ?I = "λ(j, x). x = j + a*i"
  )
```

We use a hole pattern to select the inner while loop for rewriting and also associate the identifier `i` to the bound variable that should appear in our invariant. Then we instantiate the rewrite rule with our invariant. Since we associated the identifier `i` with a bound variable, any occurrence of `i` in the instantiation will be replaced with this particular bound variable.

The `where` option we just described requires the user to know the names of the variables in the rewrite rule, which is one of the objections we mentioned in the introduction. As an alternative, `patsubst` could take a second pattern, describing the expected term after rewriting.

As an additional benefit, such a change would allow selecting the correct rule from a set of rules: For example, we could state our intent of rewriting $b + a$ to $a + b$, together with a set of algebraic rules. Knowing both the left and the right hand side, the proof method can then select the symmetry rule and apply it.

5 Practical applications

In this section, we look at some uses of `subst` in Isabelle's Archive of Formal Proofs [2] and demonstrate that our new `patsubst` method indeed allows for writing much nicer proofs.

5.1 Real world usage of subst

To gain some insight into the usage of `subst`, we decided to look at the *Archive of Formal Proofs* (AFP), a large repository of formal proofs for Isabelle. We used the most recent version of the AFP, which, at the time of this writing, was from the 14th of January 2014.

It is difficult to make any general statement about the usage of `subst`, and the usefulness of our pattern-based implementation, from the data we collected, because there are several factors that may distort our results:

- `subst` is often used during proof exploration. This is, of course, not visible in the final proof, since those exploratory statements naturally are removed as soon as a suitable proof is found.
- It is also possibility that some proof authors avoid using `subst` exactly because of the problems associated with subterm selection that this work intends to address. Even though these authors might benefit from our new `subst` implementation, we will not be able to see any evidence of this in the AFP.
- We searched the AFP using regular expressions to find usages of `subst`. Our results are not completely accurate, since we are limited by the expressiveness of regular languages. Since some of our results are only approximate, we will, in these cases, only give approximate figures.

[2] `http://afp.sourceforge.net`

Usage of `subst` in the AFP Our search revealed that the AFP contains an total of 1654 theory files (i.e. files with the extension **.thy*). Of those, only about 380 contain usages of `subst`, which is a little more than 20%. In total, there are about 4000 distinct usages of `subst` in the AFP.

We found 135 usages of `subst` that also used the `where` attribute to instantiate the applied rule. Those usages where spread across 45 files. The `of` attribute appears to be the more popular choice when rewriting with `subst`, we found over 700 usages in about 60 files. We also counted 177 instances in about 50 files where `subst` was used with at least one occurrence number.

This makes for a total of roughly 1000 usages of `subst` in the AFP where our pattern-based approach could potentially add value.

5.2 Examples

We will now examine some real world usages of `subst`, and show how they can benefit from our pattern based implementation. All of these examples are taken from the AFP. They therefore were embedded inside relatively big theories, sometimes several thousand lines long. In this section, we only want to focus on `subst` itself, so we present these examples without any context.

We chose the following examples because they each use a different method for subterm selection and they all benefit from our pattern approach. But they most certainly are not isolated instances where our approach works. Due to the sheer volume of potential examples available, we easily could have chosen other examples that would have worked just as well.

Example 8 (Cauchy/CauchysMeanTheorem.thy). Even though our previous examples where all constructed, they were not totally removed from reality, as this example shows. Consider the following subgoal:

```
(len + 1) * Σ:xs / (len * (1 + len)) = Σ:xs / len
```

To this, we want to apply the rule `mult_commute`:

```
?a * ?b = ?b * ?a
```

For this, the original author used the following command:

```
apply (subst mult_commute [where a="len"])
```

Using our pattern language, we can replace this with the following command:

```
apply (patsubst at "len * _" mult_commute)
```

This admittedly is only a relatively minor improvement. It arguably improves readability, but most importantly, it removes the variable name `a` from the command. The proof author no longer has to know the names of the variables in the `mult_commute` rule to apply it at the correct location.

Example 9 (Group-Ring-Module/Algebra1.thy). Here, the user applies the definition `segment_def`:

```
segment ?D ?a =
(if ?a ∉ carrier ?D then carrier ?D
 else {x. x ≺ ?D ?a ∧ x ∈ carrier ?D})
```

To this proof state:

```
a ∈ carrier D ⟹
b ∈ carrier D ⟹
a ≺ b ⟹ segment (Iod D (segment D a)) b = segment D a
```

This is the command the user chose to accomplish this:

```
apply (subst segment_def[of "Iod D (segment D a)"])
```

Again, we can easily resolve the ambiguity without instantiating our rule:

```
apply (patsubst at "□ = _" segment_def)
```

This command is not only far less verbose, it also explicitly states the author's intention to rewrite the outermost occurrence on the left hand side of the equation.

Example 10 (Coinductive/Coinductive_List.thy). In this example, `subst` is used with an occurrence number to apply the rule `lmap_ident[symmetric]`:

```
?t = lmap (λx. x) ?t
```

This rule is highly ambiguous. It can be applied to any subterm of the right type. There are eight such subterms in the following term:

```
lzip (lmap f xs) ys =
  lmap (λ(x, y). (f x, y)) (lzip xs ys)
```

To disambiguate the application of the rule, the user chose to state an occurrence number.

```
apply (subst (4) lmap_ident[symmetric])
```

This command is completely unreadable. It is not obvious at all which subterm the user wishes to rewrite. In this case, the user wanted to apply the rule to the variable `ys` on the left hand side of the equation. To make this intention clear, we propose using the following command:

```
apply (patsubst at ys in "□ = _" lmap_ident[symmetric])
```

6 Conclusion

In this article, we presented a combinator language for subterm selection and, building on this, implemented a single-step rewrite tactic for the Isabelle theorem prover. The purpose of single-step rewriting is to give the user full control over the rewriting process. In contrast to the existing `subst` proof method, our approach to subterm selection does support this spirit of explicitness. Patterns express the user's intention during rewriting; containing information both about the targeted subterm and its context. In addition to this, they enable us to explicitly instantiate with bound variables during rewriting. Previously, this was not possible with any rewriting tool. Compared with the language of patterns by Gonthier and Tassi we contribute handling of bound variables and a more regular and modular semantics.

A very useful addition to the rewriting implementation presented here would be support for *congruence rules*. Congruence rules provide additional assumptions when rewriting under certain contexts. As an example, when rewriting the branches of an conditional expression, we may assume the condition to hold respectively not hold:

$$\frac{b = c \qquad c \implies x = u \qquad \neg c \implies y = v}{(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)}$$

These kind of rules are heavily used by the Isabelle's automatic simplifier.

It might also be interesting to support "rewriting" for weaker relations. Window inference[3] is a proof technique for program refinement with respect to transitive and reflexive relations. This technique allows "opening a window" at any position in the term and refine it according to the given relation. Our subterm selection language would allow a comfortable selection of this position.

References

1. De Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In: Indagationes Mathematicae (Proceedings). vol. 75, pp. 381–392. Elsevier (1972)
2. Gonthier, G., Tassi, E.: A language of patterns for subterm selection. In: ITP 2012, pp. 361–376. Springer
3. Grundy, J.: Window inference in the HOL system. In: TPHOLs 1991. pp. 177–189. IEEE (1991)
4. Wiedijk, F., Scott, D.S.: The seventeen provers of the world. Springer (2006)