Introduction

Rewriting in theorem provers is the process of replacing a subterm of an expression with another term. When and if such a rewrite can happen depends on the context, i.e. the information we have about the two terms. In Lean, rewriting is possible when two terms `t` and `u` are equal `t = u` or with respect to the `propext` axiom when two propositions `p : Prop` and `q : Prop` imply each other `p <-> q`. This allows us to replace a term in a goal we want to solve or inside one of our hypothesis when doing reasoning in Lean.

This allows us to proof mathematical propositions such as the commutativity for multiplication. In the below example we can see a lean proof of the commutativity of multiplication given that additon is commutative:

```
 1  theorem mul_comm (m n : ℕ) :
 2    mul m n = mul n m := by
 3    induction m with
 4    | zero => apply mul_zero
 5    | succ m' ih =>
 6      simp [mul]
 7      rewrite [mul_succ]
 8      rewrite [add_comm]
 9      rewrite [ih]
10      rfl
```

In this example we want to proof that for any natural numbers n and m the multiplication $n \cdot m$ is equal to $m \cdot n$. In Lean we do this using structural induction[1] on the inductive type ℕ which consists only of two constructors, `zero` for constructing 0 and `succ` for constructing any successor number. After unfolding the `mul` definition in line 6 we are left with a goal $(\text{succ } m) \cdot n = n + (n \cdot m)$. The `mul_succ` theorem has the type $(\text{succ } m) \cdot n = (m \cdot n) + n$. The theorems left hand side matches the left hand side of the theorem and thus we can rewrite it (replacing the left hand side of the goal with the right hand side of our theorem). The resulting goal is $(m \cdot n) + n = n + (n \cdot m)$ which can be closed by another rewrite with an addition commutativity theorem and finally the induction hypothesis $(m \cdot n = n \cdot m)$ which also proves equality and can thus be used in a rewrite.

While this is sufficient considering the many helpful theorems and tactics Lean 4 offers, there are some cases [1] TODO where it would be helpful to consider more general rewrites that exceed equality and if and only if. When we try to solve a goal in a theorem prover we usually have a given set of hypothesis and can access theorems that we've already proven as well as tactics that can apply multiple theorems. When we want to rewrite a goal which contains a term `t` that we want to change to a term `u` we can perform a rewrite by simply showing that `u` implies `t` and thus it suffices to show `u`. The relation $(\leftrightarrow)$ is convenient because it gives us such an implication per definition. However it is possible to perform a rewrite using any relation that can lead to the desired implication.

In the Lean and Coq theorem provers relations on a type α are defined by $\alpha \to \alpha \to \text{Prop}$. When we want to proove a goal `t : Prop` and have the hypothesis of `u : Prop` as well as a proof of `h : r t u` given `r` is a relation $\text{Prop} \to \text{Prop} \to \text{Prop}$ we can proof the statement given we have the additional information that `r` implies $(\leftarrow)$, essentially $(\leftarrow)$ is a subrelation of `r`. When those hypothesis are in place the proof is straight forward for this minimal example. By Leans definition of Subrelations it suffices to whow `r t ?t` and `?t`. The question marks refer to missing values that can be filled with any given term that matches its type (Meta variables in Lean or existential variables in Coq). The second step is to instanciate `?t` with `u` and use our hypothesis that proves `u`.

[1]Structural induction means that the induction follows the structure of the inductive type.

This approach is tedious to be performed manually especially when the goal is more complicated or the term we intend to rewrite is bound by lamnda expressions or an all quantifier. When we want to prove a goal `p ∧ q` with the same context for instance and we need to rewrite `p` to `q` inside the lefthand side of the conjunction (replace `p` the without modifying the remaining term), the proof of that rewrite requires us to set a new subgoal `p ∧ q → q ∧ q`, solve that by the conjunction introduction rule leaving `t` and `u` as sub-subgoals. `u` can be proven by our hypothesis and the proof for `t` is the same proceedure as for the minimal rewrite example above. Even this approach is specific to conjunctions and can't be extended for other propositions.

A better approach for a general way of rewriting with arbitrary relations is the Morphism framework introduced by Mattheiu Sozeau [2] consisting of `respectful` and `Proper` definitions that can construct proofs for arbitrary terms using a syntax-directed algorithm. The `Proper` definition in Definition 1 merely takes a relation r and an element m in that relation demanding reflexivity. Whenever this definition holds we call m a `Proper` element of r meaning that m is a morphism for r. The `respectful` definition seen in Definition 2 denoted as $(\Longrightarrow)$ is Coqs notion for signatures. This definition can produce very general implications for a variaty of functions. For instance, the contrapositive theorem $\forall ab : \text{Prop}, (a \to b) \to (\neg b \to \neg a)$ can be stated as $((\to) \Longrightarrow (\leftarrow))(\neg)(\neg)$. We can even simplify the contrapsitive theorem by leveraging `Proper` and `respectful` with $\text{Proper } ((\to) \Longrightarrow (\leftarrow))(\neg)$. We can use the same framework to specify the above rewrite $p \land q \to q \land q$ in a more general way for instance when we create a term p of type $\text{Proper } ((=) \Longrightarrow (\leftrightarrow) \Longrightarrow (\leftarrow)) \ (\land)$ translates to $\forall xy, x = y \to \forall x'y', x' \leftrightarrow y' \to (x \land x' \leftarrow y \land y')$. When instanciating the variables in p for instance with $p, q, h : p = q, q, q, (\text{by rfl})$ we would get a proof for $(p \land q \leftarrow q \land q)$.

Definition 0.1 (Proper):

```
class ProperProxy (r : relation α) (m : α) where
  proxy : r m m
```

Definition 1: Proper

Definition 0.2 (respectful):

```
def respectful (r : relation α) (r' : relation β) : relation (α → β) :=
  λ f g ↦ ∀ x y, r x y → r' (f x) (g y)
```

Definition 2: respectful

The Coq library for morphisms has many theorems that operate on `Proper` and `respectful` terms which helps to construct and solve theorems containing morphisms and signatures. This allows us to use the same structrue and theorems for rewrites in different terms. The proof construction for $p \land q \leftarrow q \land q$ and $p \lor q \leftarrow q \lor q$. This generalisation is the base for an algorithm proposed by Matthieu [2] that automatically produces rewrite proofs for any given `Proper` relation where the term to be rewritten can be behind binders and nested in other structures. There is one more definition that makes the proposed algorithm more powerful. When we have $\text{Proper } (A \Longrightarrow B) \ f$ and we know that B is a subrelation of C we can imply $\text{Proper } (A \Longrightarrow C) \ f$.

Definition 0.3 (Subrelation):

```
def Subrelation (q r : α → α → Prop) :=
  ∀ {x y}, q x y → r x y
```

Definition 3: Subrelation

Algorithm for Genralised Rewriting

The algorithm in TODO:algoref is an imperative translation of the declarative algorithm proposed in [2] that we implemented in Lean 4. The algorithm is syntax directed and covers every term that can be constructed in Lean. The algorithm takes an empty constraint set Ψ, a term t in that we want to rewrite and a constant proof ρ that is of the type rab where r is a relation, a is a term we want to rewrite in t and b is the value we want to replace a with. The algorithm outputs a modified set Ψ which contains all wholes in the rewrite proof that can't be determined in some of the cases of the algorithm (represented in meta variables in Lean), a carrier relation r for the rewrite, the modified term u and finally the proof for the rewrite. At the beginning we always check whether the term we want to rewrite unifies directly for the given proof ρ. In that case the proof-result for a rewrite would just be ρ. Because ρ (and any proof-result of this algorithm) is not of the type $t \leftarrow u$ we will wrap the output of the algorithm in a proof for Subrelation $r\ (\leftarrow)$.

Whenever the term does not unify directly we examine the structure and use a different approach depending on whether t is an application, lambda, dependent/non-dependent arrow, or constant. Whenever we encounter an application fe we perform a recursive call on both f and e. We use the obtained carrier relation, proof, and term to construct a proof that r_f is a subrelation of $r_e \Longrightarrow ?_T$. This is where the first holes occur that we collect in the constraint set. This generates a proof for rtu. Recall that we construct a Subrelation $r\ (\leftarrow)$ after invoking `Rew` which leads to a proof of $t \leftarrow u$.

For rewrites inside lambda terms we bind $x : \tau$ to the local context and perform a recursive rewrite on the body of the lambda. The resulting proof wrapped in a fresh lambda expression binding $x : \tau$ represents the proof for $r\ (\lambda x : \tau.b)\ (\lambda x : \tau.b')$ again progressing to $(\lambda x : \tau.b) \leftarrow (\lambda x : \tau.b')$ eventually.

All other cases leverage either the lambda or application cases by converting them slightly to fit in the scheme. The non-dependent arrow case is just transformed into a function that represents an arrow. This has the advantage that locally declared functions (`impl` in this case) are considered const in Lean and thus just reuse the already defined application case. Similarly for the case of an all quantifier that uses a local dependent function `all`.

Finally we will take a look the last case is triggered whenever none of the above cases match. This is the case for constants such as `all`, `impl`, or simply for atoms that don't unify at the beginning of the `Rew` function. In this case we construct another meta variable of type Proper $\tau\ ?_r\ t'$ that is treated as a hole at the bottom of the proof tree and essentially represents and identity rewrite from t to t.

```
$\mathrm{Rew}_\rho(\Psi, t)$:
$(\Psi', r', u', \text{unifyable}) := \text{unify}_\rho(\Psi, t)$
**if** unifyable **then**:
    **return** $(\Psi', r', u', \rho)$
**match** $t$ **with**
    | $f\ e \Rightarrow$
        $(\Psi', r_f, u_f, p_f) := \mathrm{Rew}_\rho(\Psi, f)$
        $(\Psi'', r_e, u_e, p_e) := \mathrm{Rew}_\rho(\Psi', e)$
        $\Psi''' := \{?_T : \text{relation } \mathbf{type}(\text{e}), ?_{\text{sub}} : \text{subrelation } r_f(r_e \Longrightarrow ?_T)\}$
        **return** $(\Psi'' \cup \Psi''', ?_T, \text{app } u_f u_e, ?_{\text{sub}}\ f u_f p_f e u_e p_e)$
    | $\lambda$ x : $\tau$. b $\Rightarrow$
        $(\Psi', r, u, p) := \mathrm{Rew}_\rho(\Psi, b)$
        **return** $(\Psi', \text{pointwiseRelation } \tau\ r, \lambda\ \text{x} : \tau.\ \text{u}, \lambda\ \text{x} : \tau.\ \text{p})$
    | $\forall x : \tau, b \Rightarrow$
        $(\Psi', r, u, \text{unifyable}) := \text{unify*}_\rho(\Psi, b)$
        **if** unifyable **then**:
            **return** $(\Psi', r, u, \rho)$
        $(\Psi', r', \text{all } (\lambda x : \tau.b'), p) := \mathrm{Rew}_\rho(\Psi, \text{all } (\lambda x : \tau.b))$
        **return** $(\Psi', r', \forall\ \text{x} : \tau, b', p)$
    | $\sigma \to \tau \Rightarrow$
        $(\Psi', r, \text{impl } \sigma'\ \tau', p) := \mathrm{Rew}_\rho(\Psi, \text{impl } \sigma\ \tau)$
        **return** $(\Psi', r, \sigma' \to \tau', p)$
    | t' $\Rightarrow$
        **return** $(\Psi \cup \{?_r : \text{relation } \mathbf{type}(t), ?_m : \text{Proper } \tau\ ?_r\ \ t'\}, ?_r, t', ?_m)$
```

Optimisations

```
Subterm$_\rho(\Psi, t)$:
$(\Psi', r', u',$ unifyable) := unify$_\rho(\Psi, t)$
if unifyable then:
    return $(\Psi', r', u', \rho)$
match $t$ with
    | $e_0 ... e_n \Rightarrow$
        respectful := {}
        prefixIsId := true
        fn, u := $e_0$
        for $e : \tau$ in $e_0 ... e_n$ do
            $(\Psi$, result) := Subterm$_\rho(\Psi, e)$
            if prefixIsId then
                if result = identity
                    fn := fn e
                    u := u e
                    continue
                else
                    prefixIsId := false
            match result with
                | identity $\Rightarrow$
                    $\Psi := \Psi \cup \left\{?_r : \text{relation } \tau, ?_p : \text{ProperProxy } \tau ?_r\, t\right\}$
                    respectful := respectful ++ $\{?_r\}$
                    u := u e
                | success $\Rightarrow$

                return $(\Psi'' \cup \Psi''', ?_T, \text{app } u_f u_e, f u_f p_f e u_e p_e)$
            | $\lambda$ x : $\tau$. b $\Rightarrow$
                $(\Psi', r, u, p)$ := Rew$_\rho(\Psi, b)$
                return $(\Psi'$, pointwiseRelation $\tau\ r$, $\lambda$ x : $\tau$. u, $\lambda$ x : $\tau$. p)
            | $\forall x : \tau, b \Rightarrow$
                $(\Psi', r, u,$ unifyable) := unify*$_\rho(\Psi, b)$
                if unifyable then:
                    return $(\Psi', r, u, \rho)$
                $(\Psi', r', \text{all } (\lambda x : \tau . b'), p)$ := Rew$_\rho(\Psi, \text{all } (\lambda x : \tau . b))$
                return $(\Psi', r', \forall$ x : $\tau, b', p)$
            | $\sigma \to \tau \Rightarrow$
                $(\Psi', r, \text{impl } \sigma'\ \tau', p)$ := Rew$_\rho(\Psi, \text{impl } \sigma\ \tau)$
                return $(\Psi', r, \sigma' \to \tau', p)$
            | t' $\Rightarrow$
                return $(\Psi \cup \{?_r :$ relation type($t$), $?_m :$ Proper $\tau\ ?_r\ \ t'\}, ?_r, t', ?_m)$
```

Updated Algorithm

Equality of the Generated Proofs

Related Work

Conclusion

References

[1] R. JUNG, R. KREBBERS, J.-H. JOURDAN, A. BIZJAK, L. BIRKEDAL, and D. DREYER, “Iris from the ground up: A modular foundation for higher-order concurrent separation logic,” *Journal of Functional Programming*, vol. 28, p. e20, 2018, doi: 10.1017/S0956796818000151.

[2] M. Sozeau, “A New Look at Generalized Rewriting in Type Theory,” *Journal of Formalized Reasoning*, vol. 2, no. 1, pp. 41–62, Dec. 2009, [Online]. Available: https://inria.hal.science/inria-00628904