The Rewster: The Coq Proof Assistant with Rewrite Rules

Gaëtan Gilbert, Yann Leray, Nicolas Tabareau, Théo Winterhalter

The Rewster: The Coq Proof Assistant with Rewrite Rules

Gaëtan Gilbert[1], Yann Leray[1,2], Nicolas Tabareau[1], and Théo Winterhalter[1]

[1] Inria, France
[2] École Normale Supérieure, France

Dependently typed languages such as Coq or Agda are very convenient tools to program with strong invariants and develop mathematical proofs. However, a user might be inconvenienced by things such as the fact that `n` and `n+0` are not considered *definitionally* equal, or the inability to postulate one's own constructs with computation rules such as exceptions [PT18]. Coq modulo theory [Str10] solves the first of the two problems by extending Coq's conversion with decision procedures, *e.g.*, for linear integer arithmetic. Rewrite rules can be used to deal with directed equalities for natural numbers, but also to implement exceptions that compute. They were introduced in Agda [CA16] a few years ago, and later extended to provide more guarantees with a modular confluence checker [CTW19, CTW21].

We present a work-in-progress extension[1] of Coq which supports user-defined rewrite rules. While we mostly follow in the footsteps of the Agda implementation, we also have to face new issues due to the differences in the implementation and meta-theory of Coq and Agda. The most prominent one being the different treatment of universes as Coq supports cumulativity but no first-class universe levels. We will take advantage of this talk to expose our ideas on how to solve the different issues that arise when adding user-defined rewrite rules to a proof assistant by integrating[2] rewrite rules in MetaCoq [SAB+20, SBF+20], building on previous work [CTW19, CTW21].

Rewrite Rules in Coq. We only support rewrite rules whose head symbol is declared as such with the `Symbol` command, which essentially declares an axiom for which we can postulate computation rules. Rules are then declared using the `Rewrite Rule` command. For instance:

```
Symbol pplus : ℕ → ℕ → ℕ.
Rewrite Rule [ n ] ⊢ pplus 0 n ⟹ n.
Rewrite Rule [ n ] ⊢ pplus n 0 ⟹ n.
Rewrite Rule [ n m ] ⊢ pplus (S n) m ⟹ S (pplus n m).
Rewrite Rule [ n m ] ⊢ pplus n (S m) ⟹ S (pplus n m).
```

will declare the parallel plus that computes on both its arguments. On the left of the turnstile (⊢) variables are quantified and can furthermore be annotated with their type. We will illustrate the features and specificities of our implementation below, while exposing the challenges that come with them.

Non-Linearity. For now, we restrict our rewrite rules to be left-linear: each variable can only appear once on the left-hand side of the rule. This appears like a very strong limitation as certain rules are only well typed in presence of non-linearity: *e.g.*, the computation rule for the J eliminator. This can be circumvented by *forcing* certain variables to be equal to an expression:

```
Rewrite Rule [ A u P t (v := u) (B := A) (w := u) ] ⊢ J A u P t v (eq_refl B w) ⟹ t.
```

[1] Available at https://github.com/yannl35133/coq/tree/rewrite-rules-TYPES, examples can be found in the test-suite/success/rewrule.v file.
[2] Available at https://github.com/yannl35133/metacoq/tree/rewrite-rules-TYPES

These equalities are only used to help elaboration figure out implicits in the right-hand side of the rule. They bear no meaning on the rewrite rule itself. For J, this is not a problem, because all well-typed instances of the left-hand side will actually verify these identities, but it is not a guarantee in general. Take for instance the following symbol and rule:

```
Symbol f : ∀ b, if b then ℕ else 𝔹.
Rewrite Rule [ b := true ] ⊢ f b ⟹ 23.
```

The rule will even trigger for the term `f false` meaning that we have something of type $\mathbb{B}$ which reduces to 23, a violation of subject reduction: the rule is simply not type preserving. We will now see that with universes we can run into more subtle problems with respect to type preservation.

Universes and Cumulativity. As exposed before, one major difference between Coq and Agda is the treatment of universes. In Coq, cumulativity means that sharing a common type for the left-hand side and the right-hand side (*i.e.*, the requirement to postulate their propositional equality) is not sufficient to ensure type preservation, even when the rules are confluent.[3] A counter-example would be the following :

```
Symbol id : Type@{v} → Type@{u}.
Rewrite Rule [ ] ⊢ id Type ⟹ Type@{u}.
```

Both sides share the super type `Type@{1 + max u v}`, but no constraints are inferred from the definition of the rule: it works for any level `v` and `u`. In particular it can be used to map terms of type `Type@{u+1}` to `Type@{u}`. This allows the user to create a term `U : U` when they might have thought assuming `Type → Type` was harmless. Not only does this break subject reduction, but also consistency. This raises two challenges to overcome: (1) theoretically we have to come up with a modular criterion to ensure type preservation of the typing rules; (2) in practice we need to be able to collect universe constraints not only on the symbol declaration but also in the rewrite rules themselves.

Reduction Strategies. Coq also features several reduction strategies such as call-by-value (`cbv`), call-by-name (`cbn`) or call-by-need (`lazy`), which are furthermore highly parametrisable (*e.g.*, they can unfold constants or not). Naively grafting a function that matches left-hand sides of rules on top of one of these reduction machines will often lead to incompleteness issues: (1) one has to ensure that subterms (function arguments) are in weak-head-normal form before matching them, which is not the case by default with `cbn` or `lazy`; (2) a function argument brought to normal form, when substituted into another term, may create a deep redex if the pattern is itself deep, which may cause problems with `cbv`. This means that we need to be careful when adding rewrite rules to those, otherwise users may face the frustration of having to type `cbn` several times to fully evaluate redices in a term.

In fact, we believe that proving such properties about reduction strategies would be nice additions to the MetaCoq project. One final challenge we have to face when dealing with rewrite rules in MetaCoq is that we do not know a priori that the rules do not break strong normalisation, while the current implementation and verification in MetaCoq relies on this assumption [SBF+20]. We plan to solve this problem simply by no longer relying on this assumption. Instead, we believe that we can first define the reduction machine and the type checker as partial functions so that we may prove correctness on all terminating inputs, using ideas similar to the Braga method [LWM21].

[3] When the system has uniqueness of type, like Agda, confluence is sufficient.

References

[CA16] Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. In *Types for Proofs and Programs*, TYPES, 2016.

[CTW19] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. How to tame your rewrite rules. In *Types for Proofs and Programs*, TYPES, 2019.

[CTW21] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The Taming of the Rew: A Type Theory with Computational Assumptions. *Proceedings of the ACM on Programming Languages*, 2021.

[LWM21] Dominique Larchey-Wendling and Jean-François Monin. The Braga Method: Extracting Certified Algorithms from Complex Recursive Schemes in Coq. In *Proof and Computation II*, pages 305–386. WORLD SCIENTIFIC, August 2021.

[PT18] Pierre-Marie Pédrot and Nicolas Tabareau. Failure is Not an Option An Exceptional Type Theory. In *ESOP 2018 - 27th European Symposium on Programming*, volume 10801 of *LNCS*, pages 245–271, Thessaloniki, Greece, April 2018. Springer.

[SAB+20] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. To appear in Journal of Automated Reasoning, 2020.

[SBF+20] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *PACMPL*, 4(POPL), 2020.

[Str10] Pierre-Yves Strub. Coq modulo theory. In *Computer Science Logic: 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings 24*, pages 529–543. Springer, 2010.