How to Tame your Rewrite Rules

Jesper Cockx[1], Nicolas Tabareau[2], and Théo Winterhalter[2]

[1] Chalmers, Göteborg, Sweden
[2] Gallinette Project-Team, Inria Nantes France

Dependently typed languages such as Coq and Agda can statically guarantee the correctness of our proofs and programs. Extending these languages with new features can be done using rewrite rules [5]. For example, exceptional type theory [8] introduces two new constructions $\mathsf{raise} : \forall A.\, A$ and $\mathsf{catch} : \forall P.\, P\ \mathsf{true} \to P\ \mathsf{false} \to P\ \mathsf{raise} \to \forall b.\, P\ b$ together with new rewrite rules such as $\mathsf{catch}_P\ pt\ pf\ pr\ \mathsf{raise} \hookrightarrow pr$. However in general, adding rewrite rules is complex and error-prone. If a wrong rewrite rule is added, the language may lose any or all of its good properties like decidability of typechecking, canonicity, or even type safety. Moreover, these new rewrite rules may interact badly with other extensions. Checking all of this seems to require an almost unsurmountable amount of work...

We present a framework to add user-defined (higher-order and non-linear) rewrite rules to type theory in a *safe* and *modular* way. In particular, we provide checks to ensure type safety as well as decidability of conversion and thus type-checking, which in turn require checking the confluence and termination properties. We are currently working on extensions to both Agda and Coq to provide user-defined rewrite rules, where the user can pick their desired level of (un)safety by enabling or disabling individual checks (for confluence, termination, ...).

Ensuring subject reduction. By adding arbitrary rewrite rules to the type theory of our language, we can lose not only normalization and canonicity but also subject reduction. Of course subject reduction is trivially broken by a heterogeneous rewrite rule $u \hookrightarrow v$ where $u : A$ and $v : B$ with $A \not\equiv B$. But even homogeneous rewrite rules can break it:

- *Exploiting non-confluence:* Let $A : \mathsf{Set}$ with two rewrite rules $A \hookrightarrow (\mathbb{N} \to \mathbb{N})$ and $A \hookrightarrow (\mathbb{N} \to \mathsf{Bool})$. Then $\lambda(x : \mathbb{N}).\, x : A$, so $(\lambda(x : \mathbb{N}).\, x)\ 0 : \mathsf{Bool}$. But $(\lambda(x : \mathbb{N}).\, x)\ 0$ reduces to 0 which does not have type Bool.

- *Rewriting already defined symbols:* Let $\mathsf{Box}\ (A : \mathsf{Set}) : \mathsf{Set}$ be a datatype with a single constructor $\mathsf{box} : (x : A) \to \mathsf{Box}\ A$ and $\mathsf{unbox} : \mathsf{Box}\ A \to A$ with $\mathsf{unbox}\ (\mathsf{box}\ x) \hookrightarrow x$. If we add a rewrite rule $\mathsf{Box}\ (\mathbb{N} \to \mathbb{N}) \hookrightarrow \mathsf{Box}\ (\mathbb{N} \to \mathsf{Bool})$, then we have $\mathsf{unbox}\ (\mathsf{box}\ (\lambda(x : \mathbb{N}).\, x))\ 0 : \mathsf{Bool}$ but this term reduces to 0, which again does not have type Bool.

The second example exploits the fact that *core* reduction rules are applied regardless of possible non-linearities. Indeed, with explicit arguments, the reduction rule is actually $\mathsf{unbox}_A\ (\mathsf{box}_A\ x) \hookrightarrow x$, but it is triggered with $\mathsf{unbox}_{\mathbb{N}\to\mathsf{Bool}}\ (\mathsf{box}_{\mathbb{N}\to\mathbb{N}}\ (\lambda(x : \mathbb{N}).\, x))\ 0 : \mathsf{Bool}$. Here the Agda implementation implicitely assumes that Box is injective, enforcing the necessary conversion by typing. If we were to check for conversion in the reduction rule for unbox (as we do for non-linear user-defined rewrite rules) we would end up with a stuck term but subject reduction would be preserved.

Both of these examples break injectivity of Π types, which is a crucial lemma in most proofs of subject reduction. Hence we infer that we should only allow rewrite rules on 'fresh' (i.e. postulated) symbols, and that we should check confluence of the rewrite rules. From these natural restrictions, we can derive injectivity of Π types and hence re-establish subject reduction.

Checking confluence and termination. Confluence and termination of (higher-order) rewrite rules are both known and well-studied problems (see for example [7] and [3]). However, we run into a problem: most termination checks require confluence, and most confluence checks require termination. To resolve this dilemma, we fix a particular deterministic *rewriting strategy* $\rightarrow_s \subseteq \rightarrow$. We require the strategy to be *complete* in the sense that whenever $u \rightarrow v$, there exists some w such that both $u \rightarrow_s^* w$ and $v \rightarrow_s^* w$. We can then check confluence and termination:

1. First, we run the termination check on $\rightarrow$. If it succeeds, we don't know yet that $\rightarrow$ is terminating (because the termination check assumes confluence), but we do know $\rightarrow_s$ is terminating (since it is included in $\rightarrow$ and confluent by construction).

2. Second, we run the confluence check on $\rightarrow$, using $\rightarrow_s$ for joining critical pairs. If this check succeeds, we know $\rightarrow$ is confluent. Conversely, if the check fails then by completeness of $\rightarrow_s$ there is at least one critical pair that is unresolvable by $\rightarrow$, hence $\rightarrow$ is non-confluent.

3. Finally, we can now prove that $\rightarrow$ is terminating. We don't actually have to re-run the termination check, but can just make use of the stronger assumption of confluence of $\rightarrow$.

Rewriting modulo an equational theory. In many proof assistants, conversion includes not just computation rules (e.g. β-reduction) but also type-directed rules (e.g. η-conversion). Hence we consider conversion up to an equational theory: $\Gamma \vdash \_ \sim \_ : A$. For instance this relation can include η-rules for functions or records, or a definitionally proof-irrelevant universe of propositions [6].

If rewrite rules do not respect the equational theory, we run into trouble. For example, let $f : \forall A.\, (A \to A) \to \mathsf{Bool}$ with a rewrite rule $f_A\ (\lambda x.\, x) \rightarrow \mathsf{true}$ and consider the term $f_\top\ (\lambda x.\, \mathsf{tt})$. The argument $\lambda x.\, \mathsf{tt}$ is not of the form $\lambda x.\, x$, yet the rewrite rule should still apply since $\mathsf{tt} \sim x : \top$! So we should take the equational theory into account when defining reduction, as well as when implementing the checks for confluence and termination.

As such we require that rewrite rules are well-behaved with respect to the equational theory: if $\Gamma \vdash a \sim a' : A$ and $a \rightarrow b$, then there must exist b' such that[1] $\Gamma \vdash b \sim b' : A$ and $a' \rightarrow^* b'$. For our implementation in Coq and Agda, this property is built into the definition of rewriting so it does not require a separate check. With this property we are able to deduce that conversion between two terms $\Gamma \vdash t = u : A$ is equivalent to having both terms reduce to *squiggly* terms: $t \rightarrow^* t'$ and $u \rightarrow^* u'$ and $\Gamma \vdash t' \sim u' : A$. Together with a natural assumption on how the equational theory interacts with Π types[2], this allows us to prove injectivity of Π types and check confluence in the presence of a non-trivial equational theory.

Conclusion. User-defined rewrite rules allow you to extend the power of a dependently typed language on a much deeper level than normally allowed. We already mentioned exceptional type theory; other potential applications include adding new equations to neutral terms [1], defining quotient types [4] or higher inductive types, and implementing guarded type theory [2]. By having access to the necessary checks you can be confident that no important properties will break by accident, while you still have the option to ignore the checks when required by your application. Soon rewrite rules and the corresponding safety checks are coming to both Agda and Coq. We cannot wait to see what other uses you will come up with!

[1] We use A for b as well, this is not implicitely assuming subject reduction, it is simply a hint to guide the equational theory.

[2] Namely that $\Pi\ A'\ B' \sim \Pi\ C'\ D'$ is equivalent to $A' \sim C'$ and $B' \sim D'$.

References

[1] Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: A sound and complete decision procedure, formalized. In *Workshop on Dependently-typed Programming*, 2013.

[2] Lars Birkedal and Rasmus Ejlers Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 213–222. IEEE Computer Society, 2013.

[3] Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency pairs termination in dependent type theory modulo rewriting. 2019.

[4] Guillaume Brunerie. quotients.agda. https://github.com/guillaumebrunerie/initiality/blob/reflection/quotients.agda.

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

[6] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages*, pages 1–28, January 2019.

[7] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical computer science*, 192(1):3–29, 1998.

[8] 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.

How to break subject reduction with homogeneous rewrite rules in Agda. Below we include a short piece of Agda code that exploits non-confluent rewrite rules to construct a term true' : ℕ that evaluates to true : Bool, thus breaking subject reduction.

```
postulate
  A : Unit → Set
  A-is-Bool : A unit ≡ Bool
{-# REWRITE A-is-Bool #-}

f : (x : Unit) → A x
f unit = true

postulate
  A-is-Nat : ∀ {x} → A x ≡ Nat
{-# REWRITE A-is-Nat #-}

true' : Nat
true' = f unit
```