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. We can extend these languages with new features 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 $\mathsf{catch}_P\ pt\ pf\ pr\ \mathsf{raise} \rightarrow pr$. However in general, by adding a wrong rewrite rule, 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.

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. In the current implementation of rewrite rules in Agda, 'bad' rewrite rules can destroy not only normalization and canonicity but also subject reduction:

- *Exploiting non-confluence:* Let $A : \mathsf{Set}$ with rewrite rules $A \rightarrow (\mathbb{N} \to \mathbb{N})$ and $A \rightarrow (\mathbb{N} \to \mathsf{Bool})$, then $(\lambda(x : \mathbb{N}).\, x)\ 0 : \mathsf{Bool}$. But this 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 $\mathsf{unbox} : \mathsf{Box}\ A \to A$ defined by $\mathsf{unbox}\ (\mathsf{box}\ x) = x$. If we add a rewrite rule $\mathsf{Box}\ (\mathbb{N} \to \mathbb{N}) \rightarrow \mathsf{Box}\ (\mathbb{N} \to \mathsf{Bool})$, then we have $\mathsf{unbox}\ (\mathsf{box}\ (\lambda(x : \mathbb{N}).\, x))\ 0 : \mathsf{Bool}$ but this term again reduces to 0, which does not have type Bool.

The second example exploits the fact that $\mathsf{unbox}_A\ (\mathsf{box}_B\ x) \rightarrow x$ even when $A \neq B$. Here Agda 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 user-defined rewrite rules) evaluation would be stuck but subject reduction would be preserved.

Both these examples break injectivity of Π types, which is a crucial lemma in most proofs of subject reduction. Hence we infer that we should check confluence of the rewrite rules, and we should only allow rewrite rules on 'fresh' (i.e. postulated) symbols. 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, checking termination usually requires confluence, and checking confluence usually requires termination. We propose to resolve this dilemma by fixing a deterministic *rewriting strategy* $\rightarrow_s \subseteq \rightarrow$, which is *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 check termination of $\rightarrow$. 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 check confluence of $\rightarrowtriangle$, using $\rightarrowtriangle_s$ for joining critical pairs. Because $\rightarrowtriangle_s \subseteq \rightarrowtriangle$ and $\rightarrowtriangle_s$ is complete, this check succeeds iff $\rightarrowtriangle$ is confluent.
3. Finally, we conclude that $\rightarrowtriangle$ is terminating, using point 1. and the confluence of $\rightarrowtriangle$.

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 $\sim$. E.g. 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 $\sim$, we run into trouble. For example, let $f : \forall A.\, (A \to A) \to \mathsf{Bool}$ with a rewrite rule $f_A\ (\lambda x.\, x) \rightarrowtriangle \mathsf{true}$ 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 $\mathsf{tt} \sim x : \top$!

As such we must ensure that rewrite rules are well-behaved with respect to the equational theory: if $a \sim a' : A$ and $a \rightarrowtriangle b$, then there must exist b' such that[1] $b \sim b' : A$ and $a' \rightarrowtriangle^* b'$. With this property we are able to deduce that conversion between two terms $\Gamma \vdash t = u : A$ is equivalent to $t \rightarrowtriangle^* t'$ and $u \rightarrowtriangle^* u'$ and $\Gamma \vdash t' \sim u' : A$. This allows us to prove[2] 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!

References

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

[2] Lars Birkedal and Rasmus Ejlers Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *LICS '13*.

[3] Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency pairs termination in dependent type theory modulo rewriting. unpublished, 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 '16*.

[6] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof-Irrelevance without K. *POPL '19*.

[7] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical computer science*, 1998.

[8] Pierre-Marie Pédrot and Nicolas Tabareau. Failure is Not an Option An Exceptional Type Theory. In *ESOP '18*.

[1] This does not assume that $b : A$, the type A is simply a hint to guide the equational theory.

[2] Further assuming $\Pi\ A'\ B' \sim \Pi\ C'\ D'$ is equivalent to $A' \sim C'$ and $B' \sim D'$