



HAL
open science

The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant

Yann Leray, Gaëtan Gilbert, Nicolas Tabareau, Théo Winterhalter

► **To cite this version:**

Yann Leray, Gaëtan Gilbert, Nicolas Tabareau, Théo Winterhalter. The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant. 2024. hal-04511667v1

HAL Id: hal-04511667

<https://inria.hal.science/hal-04511667v1>

Preprint submitted on 19 Mar 2024 (v1), last revised 18 Sep 2024 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant

Yann Leray ✉

LS2N, Nantes Université
Inria, France

Gaëtan Gilbert

Inria, France

Nicolas Tabareau 

Inria, France

Théo Winterhalter 

Inria, France

Abstract

In dependently typed proof assistants, users can declare axioms to extend the ambient logic locally with new principles and propositional equalities governing them. Additionally, rewrite rules have recently been proposed to allow users to extend the logic with new definitional equalities, enabling them to handle new principles with a computational behaviour. While axioms can only break consistency, the addition of arbitrary rewrite rules can break other important metatheoretical properties such as type preservation. In this paper, we present an implementation of rewrite rules on top of the Coq proof assistant, together with a modular criterion to ensure that the added rewrite rules preserve typing. This criterion, based on bidirectional type checking, is formally expressed in PCUIC — the type theory of Coq recently developed in the MetaCoq project.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases type theory, dependent types, rewrite rules, type preservation, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.??

1 Introduction

Dependently typed languages are the basis of major proof assistants like Agda [11], Coq [15] and Lean [5]. In these systems, since general equality is undecidable, it is split into a decidable fragment called *definitional* and the complete but undecidable *propositional* equality. While propositional equality can be extended through axioms, definitional equality is defined upfront and limited in a way that can hinder proof developments and usability. Rewrite rules, as recently proposed by Cockx *et al.* [3], can mitigate these limitations, by allowing users to extend definitional equality in a powerful way through custom reductions. One standard example is parallel plus, an addition which can reduce on constructors from both sides, something which cannot be defined using a fixpoint.

```
Symbol pplus : ℕ → ℕ → ℕ.  
Infix "++" := pplus.  
Rewrite Rules pplus_red :=  
| 0 ++ ?n ~> ?n      | S ?m ++ ?n ~> S (?m ++ ?n)  
| ?m ++ 0 ~> ?m      | ?m ++ S ?n ~> S (?m ++ ?n).
```

The symbol `pplus` behaves as standard addition, but with additional definitional equalities, that only hold propositionally for standard addition. For instance, `n ++ 0` and `0 ++ n` are both definitionally equal to `n`. This example shows the definition of a function that could already be defined, albeit with fewer definitional properties, but it is also possible to extend



© Yann Leray, Gaëtan Gilbert, Nicolas Tabareau and Théo Winterhalter;
licensed under Creative Commons License CC-BY 4.0

Fifteenth Conference on Interactive Theorem Proving (ITP 2024).

Editors: John Q. Open and Joan R. Access; Article No. ??; pp. ??:1–??:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

???:2 The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant

46 the logic of Coq with new powerful constructions like inductive-recursive types using rewrite
47 rules. As an illustration, consider the definition of (a simple version of) a universe of types:

```
48 Axiom U : Type.  
49 Symbol El : U → Type.  
50 Symbols (N : U) (Pi : ∀ a : U, (El u → U) → U).  
51 Symbol U_rect : ∀(P : U → Type), P N → (∀ a b, (∀ A : El a, P (b A)) → P (Pi a b)) → ∀ u, P u.  
52 Rewrite Rules U_rect_rew :=  
53 | U_rect ?P ?pN ?pPi N ~> ?pN  
54 | U_rect ?P ?pN ?pPi (Pi ?a ?b) ~> ?pPi ?a ?b (fun (A : El ?a) => U_rect ?P ?pN ?pPi (?b A)).  
55 Rewrite Rules El_red :=  
56 | El N ~> N  
57 | El (Pi ?a ?b) ~> ∀ (A : El ?a), El (?b A).  
58
```

60 Without rewrites rules, this construction can be fully postulated with axioms using proposi-
61 tional equality, but in practice neither the elimination principle `U_rect` nor function `El` will
62 compute, which makes the construction much less convenient to use. From this point of
63 view, strictly positive indexed inductive types currently available in Coq can be seen as one
64 particular class of inductive constructions that have been anticipated, justified and provided
65 natively in the system. Rewrite rules allow user to go outside this fragment for instance by
66 defining a non-strictly positive inductive type.¹

67 However, contrarily to axioms which can only endanger consistency, the power of rewrite
68 rules needs to be put in check or important metatheoretical properties can be broken. The
69 work of Cockx *et al.* puts in light that the first property that rewrite rules must verify is
70 confluence, otherwise subject reduction can be broken and type checking quickly becomes
71 undecidable. They develop a criterion that can be checked to determine if a rewriting system
72 satisfies confluence, namely the triangle property [3].

73 To preserve subject reduction of the complete theory, rewrite rules must also verify a
74 second property, type preservation, that is if $t : T$ and t rewrites to t' , then $t' : T$. A
75 simple and intuitive check for type preservation is to require that the equality induced by
76 the rewrite rule should be well typed for every possible instance $(\forall \vec{x}, p = r)$. There are
77 however problems with this criterion as terms don't have a unique type in Coq because of the
78 subtyping introduced by universe cumulativity (`Type@{u}` is a subtype of `Type@{v}` when
79 $u \leq_u v$).

80 ► **Example 1.** If we consider an identity function on types, which simply returns its argument,
81 cumulativity should mandate that the domain universe be smaller than the codomain universe.
82 However, in the following example with a rewrite rule, this restriction is not enforced.

```
83 Symbol id@{u v} : Type@{u} → Type@{v}.  
84 Rewrite Rule id_rew := id ?T ~> ?T.  
85 Universe a.  
86 Check id Type@{a} : id Type@{a}.  
87
```

89 The reason is that the naive check only mandates that both universes share a common bigger
90 universe – something which is always verified. Thus, the check doesn't forbid this rule which
91 nonetheless breaks consistency by allowing the existence of a term $U : U$ [8, 9].

¹ The fact that general non-strictly positive inductive types are unsafe is due to Coquand and Paulin [4], but the argument uses impredicativity.

92 In this paper, we show that actually, having a common type is not enough, but the correct
 93 criterion checks that the principal type inferred for the pattern is a valid type for the
 94 right-hand side.

95 Another issue with this criterion is that the type of the variables \vec{x} that appear in the
 96 pattern might also not be unique, and just testing for one possibility that works is not
 97 enough.

98 ► **Example 2.** Let us consider a symbol which extracts the domains of product types, which
 99 can be defined as follows:

```
100 Symbol dom : Prop → Prop.
101 Rewrite Rule dom_rew := dom (∀ (x : ?A), ?B) ~> ?A.
102 Eval compute in dom (∀ (x : N), True). (* N : Prop *)
```

105 The problem here is that no information tells us what sort $?A$ has in the pattern in general,
 106 since the domain of a product type may have any sort quality (SProp, Prop or Type). Our
 107 naive criterion, however, fails to detect the issue because the equality is well typed when
 108 both $?A$ and $?B$ have type Prop, since then both the pattern and the replacement term will
 109 have that type, and this problematic rule can then be accepted.

110 The correct criterion will make use of typing in patterns to try to find equalities between
 111 pattern variables, so they can be used when typing the replacement term. We can however
 112 also face an issue if we use the existing unification algorithm as is on patterns, since it uses
 113 shortcut heuristics which don't always assume the worst.

114 ► **Example 3.** Consider the following:

```
115 Inductive Box (b : B) : Type := box. (* box : ∀ (b : B), Box b *)
116 Symbol I : B → B.
117 Symbol C : ∀ (b : B), Box b → Box (I b).
118 Symbol D : ∀ (b : B), Box (I b) → Box b.
119 Rewrite Rule D_C := D _ (C _ ?b) ~> ?b.
120
121 Rewrite Rule I_rew := I _ ~> false.
122
123 Definition a : Box false := (D false (C _ (box true))).
124 Eval compute in a. (* box true : Box false *)
```

127 The rewrite rule $D_ (C_ ?b) \rightsquigarrow ?b$ looks enticing, if the box is not to be considered as a
 128 truncation. However, we don't know the behaviour of I at that point, and as we add rule
 129 $I_ \rightsquigarrow \text{false}$, our first rule loses its type preservation property, so it should not in fact have
 130 been accepted in the first place.

131 The last meta-theoretical property that is not endangered by adding axioms but may break
 132 with additional rewrite rules is termination. However, non-termination cannot introduce
 133 proofs of \perp ; these can only come from the declaration of symbols and rules which are
 134 inconsistent propositionally from the start [3, Section 6.4]. It does not break subject
 135 reduction either, and we can derive a type checker that is valid irrespective of termination,
 136 in the sense that if it answers, the answer is correct. Non-termination can at worst result
 137 in type checking divergence. Therefore, we can show subject reduction with our criterion
 138 without relying on termination.

139 In this paper, we present how rewrite rules can be added to PCUIC, the theory of the
 140 kernel of Coq as defined in the MetaCoq project [13, 14] Coq, and extend the work of Cockx

| | | | |
|---------------------------|-------|--|------------------|
| A, B, P, T, c, t, \dots | $::=$ | x | variable |
| | | $?x\{\vec{t}\}$ | metavariable |
| | | s | sort |
| | | A | axiom |
| | | C | constructor |
| | | I | inductive |
| | | $t\ t'$ | application |
| | | $\lambda(x : A : s), t$ | abstraction |
| | | $\forall(x : A : s), (B : s')$ | product |
| | | case c return $P : s$ with \vec{t} | case destruction |
| | | fix $f : T := t$ | fixpoint |
| s, s' | $::=$ | \Box_u^q | sort |
| q | $::=$ | SProp Prop Type | qualities |

■ **Figure 1** The (annotated) syntax of PCUIC, extended with metavariables

141 *et al.* [3] by providing a criterion for type preservation in presence of cumulativity. Rewrite
 142 rules as presented in this paper have been implemented in Coq and have been merged to the
 143 main branch of its repository² meaning that they should be available in the release 8.20 of
 144 Coq.

145 **Outline of the paper.** We begin by presenting an extension of PCUIC with metavariables
 146 in Section 2. We then define rewrite rules in Section 3 before explaining the type preservation
 147 criterion in Sections 4 and 5. Finally, we explore the implementation in Section 6 and
 148 conclude with related and future work in Sections 7 and 8.

149 2 PCUIC with Metavariables

150 2.1 Syntax

151 PCUIC, whose syntax is given in Figure 1, is a formal description of the kernel of Coq,
 152 as defined in the MetaCoq project [14]. It is a dependent type theory with inductive
 153 types, where inductive elimination is split between fixpoints and case analysis. Its sorts
 154 are of three different so-called *qualities*: the usual Type hierarchy, with universe levels u
 155 as index, and two additional sorts Prop and SProp. These two last sorts are impredicative,
 156 and SProp also has definitional uniqueness of proofs, *i.e.*, any two elements of a type in
 157 SProp are convertible. Universes are ordered with \leq_u , but when the quality of a sort is
 158 Prop or SProp, its universe becomes irrelevant. This defines an ordering \leq_s on sorts with
 159 $\Box_u^q \leq_s \Box_{u'}^{q'} \iff q = q' \wedge (q \in \{\text{SProp}, \text{Prop}\} \vee u \leq_u u')$. We also order $\Box_u^{\text{Prop}} <_s \Box_{u'}^{\text{Type}}$.
 160 One should note that λ -abstractions, products and cases are heavily annotated; the type
 161 annotation on the domains is standard for PCUIC, but the additional sort annotations are
 162 only required for the proof of our type preservation criterion and should be considered as
 163 virtual. They can be unambiguously obtained from the typing judgment on the term and
 164 will also be omitted when they aren't useful.

165 To account for higher-order rewrite rules, the syntax needs to be extended with higher-

² <https://github.com/coq/coq/pull/18038>

166 order variables—henceforth called metavariables. A metavariable $?x\{\vec{t}\}$ is meant to live
 167 in an extended context, where said context extension is to be instantiated by the context
 168 instantiation \vec{t} that the metavariable carries. While regular variable substitution will be
 169 denoted as $t[\tau]$, metasubstitution—substitution for metavariables—will be denoted as $t\{\sigma\}$.
 170 Regular substitutions do nothing to metavariables, they only propagate to the context
 171 instantiation: $(?x\{\vec{t}\})[\tau] = ?x\{\vec{t}[\tau]\}$. Dually, metasubstitutions do nothing to variables
 172 and operate solely on metavariables as $(?x\{\vec{t}\})\{\sigma\} = \sigma(?x)[\vec{t}\{\sigma\}]$, they also propagate
 173 transparently to subterms on all other grammar constructions. Note that metasubstitutions
 174 need to be defined on all metavariables of the substituted term, a condition that will be
 175 verified for metasubstitutions appearing in rewrite rules.

176 2.2 Typing

177 The typing judgment of PCUIC is described in Figure 2. Judgement $\Sigma; \Theta; \Gamma \vdash t : T$ states
 178 that term t has type T in local context Γ , metavariable context Θ and environment Σ ; its
 179 rules are standard. The type of sorts needs to take into account qualities : $\square_{-}^{\text{Prop}}$ and $\square_{-}^{\text{SProp}}$
 180 both have type \square_0^{Type} and \square_u^{Type} has type $\square_{u+1}^{\text{Type}}$; this is condensed in the universe successor
 181 function $\uparrow_q u$ which equals $u + 1$ when $q = \text{Type}$ and 0 otherwise. Similarly, the sort of a
 182 product has the quality of its codomain, but its universe level depends on both sorts: $u *_{q'}^q u'$
 183 is $\max(u, u')$ if $q = q' = \text{Type}$ and u' otherwise.

184 Environment Σ can contain axiom declarations and inductive declarations. An axiom
 185 declaration simply gives a name and a type associated with it. An inductive declaration
 186 contains a name I , a context of parameters Γ_p , a context of indices Γ_i and a list of constructors,
 187 each with its name C_k , context Γ_k and instantiation of the indices \vec{v}_k . Note that \vec{v}_k may
 188 depend on Γ_p, Γ_k . Rule (CASE) asks for a scrutinee c of type I with some instantiation of
 189 the parameters \vec{p} and some instantiation of the indices \vec{v} , a return type P which depends on
 190 a generic instance of the inductive with the given parameters and branches at the return
 191 type specialized with the indices of the associated constructor, in the extended context of the
 192 constructor. Typing of environment Σ should enforce the usual strict positivity condition on
 193 inductive type declarations in order to preserve consistency.

194 In order to define typing of metavariables, the typing judgement needs to take as input
 195 a metavariable context Θ . Its elements are of the form $(\Gamma_{?x} \vdash ?x : T)$, which reads as ‘ $?x$
 196 is of type T in context extension $\Gamma_{?x}$ ’; which means in turn that the context instantiation
 197 associated to $?x$ must instantiate $\Gamma_{?x}$. Technically, the typing rule for metavariables (Rule
 198 META) differs from the rule for variables (Rule VAR) in that it additionally checks that
 199 $\Sigma; \Theta; \Gamma \vdash \vec{t} : \Gamma_{?x}$, where typing is extended to substitution in a pointwise way.

200 Note that we remained abstract in our notion of guard condition for fixpoints. Indeed,
 201 fixpoints need to recurse on structurally smaller arguments in order to avoid inconsistencies.
 202 Moreover to ensure termination, the computation rule for fix is typically guarded so that
 203 fixpoints only unfold when they can consume a constructor. We forego this condition as we
 204 do not worry about termination in this paper.

205 2.3 Cumulativity

206 Our typing judgment contains a cumulativity rule, instead of the more traditional conversion
 207 rule. Cumulativity is defined similarly to conversion, except that we allow subtyping
 208 on sorts: $\square_u^q \leq_s \square_{u'}^{q'}$. Formally, cumulativity is defined as the transitive closure \leq of
 209 $\rightarrow \cup \leftarrow \cup \leq_\alpha$ where \rightarrow is reduction, \leftarrow antireduction (the relation symmetric to \rightarrow) and \leq_α
 210 is α -cumulativity. We also define conversion \equiv by replacing α -cumulativity with α -conversion.

??:6 The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant

$$\begin{array}{c}
\frac{\Sigma; \Theta \vdash \Gamma \quad (x : T) \in \Gamma}{\Sigma; \Theta; \Gamma \vdash x : T} \text{VAR} \qquad \frac{\Sigma; \Theta \vdash \Gamma \quad (C : T) \in \Sigma}{\Sigma; \Theta; \Gamma \vdash C : T} \text{ENVVAR} \\
\\
\frac{\Sigma; \Theta \vdash \Gamma \quad (\Gamma_{?x} \vdash ?x : A) \in \Theta \quad \Sigma; \Theta; \Gamma \vdash \vec{t} : \Gamma_{?x}}{\Sigma; \Theta; \Gamma \vdash ?x\{\vec{t}\} : A} \text{META} \qquad \frac{\Sigma; \Theta \vdash \Gamma}{\Sigma; \Theta; \Gamma \vdash \square_u^q : \square_{\uparrow q u}^{\text{Type}}} \text{SORT} \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash f : \forall(x : A), B \quad \Sigma; \Theta; \Gamma \vdash t : A}{\Sigma; \Theta; \Gamma \vdash f t : B[x := t]} \text{APP} \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash A : \square_u^q \quad \Sigma; \Theta; \Gamma, x : A \vdash t : B}{\Sigma; \Theta; \Gamma \vdash \lambda(x : A : \square_u^q), t : \forall(x : A), B} \text{LAMBDA} \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash A : \square_u^q \quad \Sigma; \Theta; \Gamma, x : A \vdash B : \square_{u'}^{q'}}{\Sigma; \Theta; \Gamma \vdash \forall(x : A : \square_u^q), (B : \square_{u'}^{q'}) : \square_{u *_{q'} u'}^{q'}} \text{FORALL} \\
\\
\frac{(\Gamma_p, \Gamma_i, [I : \forall \Gamma_p, \forall \Gamma_i, \square_{u_i}^{q_i}], \vec{\Gamma}_k, [C_k : \forall(\vec{p} : \Gamma_p), \forall \Gamma_k, I \vec{p} \vec{i}_k]_k) \in \Sigma \quad \Sigma; \Theta; \Gamma \vdash c : I \vec{p} \vec{i} \quad \Sigma; \Theta; \Gamma, (\vec{v} : \Gamma_i), (x : I \vec{p} \vec{v}) \vdash P : \square_u^q \quad \Sigma; \Theta; \Gamma, \Gamma_k \vdash b_k : P[\Gamma_i := \vec{i}_k, x := C_k \vec{p} \vec{i}_k]_k}{\Sigma; \Theta; \Gamma \vdash \text{case } c \text{ return } P : \square_u^q \text{ with } \vec{b}_k : P[\Gamma_i := \vec{i}, x := c]} \text{CASE} \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash T : \square_u^q \quad \Sigma; \Theta; \Gamma, f : T \vdash t : T \quad f \text{ guarded}}{\Sigma; \Theta; \Gamma \vdash \text{fix } f : T := t : T} \text{FIX} \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash t : A \quad \Sigma; \Theta; \Gamma \vdash A \leq B \quad \Sigma; \Theta; \Gamma \vdash B : \square_u^q}{\Sigma; \Theta; \Gamma \vdash t : B} \text{CONV} \\
\\
\frac{}{\Sigma; \Theta; \Gamma \vdash (\lambda(x : A), t) a \rightarrow t[x := a]} \beta \qquad \frac{}{\Sigma; \Theta; \Gamma \vdash \text{fix } f := t \rightarrow t[f := \text{fix } f := t]} \text{FIXRED} \\
\\
\frac{(\Gamma_p, \Gamma_i, [I : \forall \Gamma_p, \forall \Gamma_i, \square_{u_i}^{q_i}], \vec{\Gamma}_k, [C_k : \forall(\vec{p} : \Gamma_p), \forall \Gamma_i, I \vec{p} \vec{i}_k]_k) \in \Sigma}{\Sigma; \Theta; \Gamma \vdash \text{case } C_j \vec{p} \vec{a} \text{ return } P \text{ with } \vec{b}_k \rightarrow b_j[\Gamma_j := \vec{a}]} \iota \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash \square_u^q \leq_s \square_{u'}^{q'} \quad \Sigma; \Theta; \Gamma \vdash T : \text{SProp} \quad \Sigma; \Theta; \Gamma \vdash t : T \quad \Sigma; \Theta; \Gamma \vdash u : T}{\Sigma; \Theta; \Gamma \vdash \square_u^q \leq_\alpha \square_{u'}^{q'} \quad \Sigma; \Theta; \Gamma \vdash t \leq_\alpha u} \\
\\
\frac{?x \in \Theta \quad \Sigma; \Theta; \Gamma \vdash \vec{t} \equiv_\alpha \vec{t}'}{\Sigma; \Theta; \Gamma \vdash ?x\{\vec{t}\} \leq_\alpha ?x\{\vec{t}'\}} \qquad \frac{\Sigma; \Theta; \Gamma \vdash f \leq_\alpha f' \quad \Sigma; \Theta; \Gamma \vdash a \equiv_\alpha a'}{\Sigma; \Theta; \Gamma \vdash f a \leq_\alpha f' a'} \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash A \equiv_\alpha A \quad \Sigma; \Theta; \Gamma \vdash t \leq_\alpha t'}{\Sigma; \Theta; \Gamma \vdash \lambda(x : A : \square_u^q), t \leq_\alpha \lambda(x : A' : \square_{u'}^{q'}), t'} \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash A \equiv_\alpha A' \quad \Sigma; \Theta; \Gamma \vdash B \leq_\alpha B'}{\Sigma; \Theta; \Gamma \vdash \forall(x : A : \square_u^q), (B : \square_{u_1}^{q_1}) \leq_\alpha \forall(x : A' : \square_{u'}^{q'}), (B' : \square_{u'_1}^{q'_1})}
\end{array}$$

■ **Figure 2** Typing, reduction and cumulativity judgment in PCUIC with metavariables

| | | | |
|---------|-------|--|-------------------------------------|
| p | $::=$ | x | variable |
| | | $\square_{?x}^q$ | sort |
| | | C | constructor |
| | | I | inductive |
| | | S | symbol |
| | | $p a$ | application |
| | | $\lambda(x : a : s), p$ | abstraction |
| | | $\forall(x : a : s), (b : s')$ | product |
| | | case p return $a : s$ with \vec{b} | case destruction |
| a, b | $::=$ | $p \mid ?x$ | pattern / pattern variable |
| q | $::=$ | $\text{SProp} \mid \text{Prop} \mid \text{Type} \mid ?x$ | explicit quality / quality variable |
| s, s' | $::=$ | $\square_{?x}^y$ | pattern sort annotation |

■ **Figure 3** Syntax of patterns p and argument patterns a, b

211 The reduction is generated by β and ι -reductions, unfolding of fixpoints (Rule `FIXRED`)
 212 and is allowed to happen in any subterm. α -cumulativity is roughly α -equality with the
 213 cumulativity rule on sorts explained above, closed by congruence. However, most subterms
 214 need to be convertible and not only cumulative in the congruence. Formally, cumulativity \leq_α
 215 is defined with the rules described in Figure 2 and the other congruences where all subterms
 216 need to be related with \equiv_α , where $t \equiv_\alpha t'$ is defined as $t \leq_\alpha t' \wedge t' \leq_\alpha t$.

217 PCUIC also supports `SProp`'s proof irrelevance in α -cumulativity. The rule given is more
 218 idealistic than the real presentation with relevance marks [7] but the differences won't matter
 219 here.

220 3 Rewrite Rules

221 A rewrite rule is composed of a left-hand side, called *pattern*, and a right-hand side, called
 222 *replacement term*, and written $p \rightsquigarrow r$. When it is applied, a term t is *matched* against the
 223 pattern p to extract subterms at the *holes* of the pattern to form a metasubstitution σ , which
 224 is then substituted in the replacement term r , resulting in reduction $t \rightarrow r\{\sigma\}$. As explained
 225 in the introduction, restrictions are needed so that the metatheory of PCUIC does not break.

226 3.1 Pattern

227 The syntax of patterns is described in Figure 3. As a separation from axioms, that never
 228 reduce, we introduce in PCUIC a new class of constants named symbols, which is a specific
 229 subclass of axioms on which rewrite rules operate.

230 Patterns corresponds to a subset of terms that need to be *rigid*. Their holes are denoted
 231 as metavariables ($?x$, also called pattern variables) in the pattern. Note that pattern variables
 232 don't carry context instantiations contrarily to metavariables. Thus to see a pattern variable
 233 as a term, we have to add the identity instantiation $\{|\Delta| := \Delta\}$ for Δ the context in which
 234 the pattern variable lives. This gives us an injection $p \mapsto p$ from patterns to terms. From
 235 now on, we will simply use the change of color to represent this injection.

236 The rigidity requirements manifests as limitations on where holes may appear, so patterns
 237 are split into bona fide patterns, where holes will *not* be allowed, and *argument patterns*
 238 where they will be allowed. Since term reduction happens in a stack machine, the reduction

$$\begin{array}{c}
 \frac{}{t \mid ?x \{[?x := t]\}} \quad \frac{}{x \mid x \{\emptyset\}} \quad \frac{C \in \Sigma \text{ not a constant}}{C \mid C \{\emptyset\}} \quad \frac{q \text{ explicit quality}}{\square_u^q \mid \square_{?x_u}^q \{[?x_u := u]\}} \\
 \\
 \frac{}{\square_u^q \mid \square_{?x_u}^q \{[?x_q := q; ?x_u := u]\}} \quad \frac{f \mid p \{\sigma_1\} \quad t \mid a \{\sigma_2\}}{f t \mid p a \{\sigma_1 + \sigma_2\}} \quad 7 \\
 \\
 \frac{A \mid a \{\sigma_1\} \quad t \mid p \{\sigma_2\}}{\lambda(x : A : \square_{u'}^{q'}), t \mid \lambda(x : a : \square_u^q), p \{[q := q'; u := u'] + \sigma_1 + \sigma_2\}} \\
 \\
 \frac{A \mid a \{\sigma_1\} \quad B \mid b \{\sigma_2\}}{\forall(x : A : \square_{u_1}^{q_1}), (t : \square_{u_2}^{q_2}) \mid \forall(x : a : \square_{u_1}^{q_1}), (b : \square_{u_2}^{q_2}) \{[\vec{q}_i := \vec{q}'_i; \vec{u}_i := \vec{u}'_i] + \sigma_1 + \sigma_2\}} \\
 \\
 \frac{c \mid p \{\sigma_c\} \quad P \mid b \{\sigma_P\} \quad b_i \mid b_i \{\sigma_i\}}{\text{case } c \text{ return } P : \square_{u'}^{q'} \text{ with } \vec{b}_i \mid \text{case } p \text{ return } a : \square_u^q \text{ with } \vec{b}_i \{[q := q'; u := u'] + \sigma_c + \sigma_P + \sum \sigma_i\}}
 \end{array}$$

■ **Figure 4** Description of pattern matching

239 of rewrite rules has to work on the machine representation of terms which have already been
 240 reduced to weak head normal form. For instance, having a hole at the head of eliminations
 241 (e.g., pattern $?f \ ?a$ against term $(\lambda x f, f x) 0 \ S$) would be unstable (does $?a$ match with 0
 242 or S ?) and would not commute with other reductions ($?f$ can match with S after reductions),
 243 so it must be forbidden. This means that argument patterns may be anywhere but at the
 244 main subterm of an elimination construction, that is an application or a case destruction.

245 We also have to impose additional restrictions to prove confluence; they are listed and
 246 justified in Section 4.1. Finally, since we don't want to try all rewrite rules at all steps in
 247 reduction, we restrict patterns such that the head of their main branch of the pattern needs
 248 to be a symbol, which will be the starting point to pattern matching during reduction.

In the end, each rewrite rule induces the following reduction rule:

$$\frac{(p \rightsquigarrow r) \in \Sigma \quad t \mid p \{\sigma\}}{\Sigma; \Theta; \Gamma \vdash t \rightarrow r \{\sigma\}} \text{REW}$$

249 where $t \mid p \{\sigma\}$ denotes the matching of p against a term t as described in the following
 250 section.

251 3.2 Pattern-Matching

252 Pattern matching is the operation that tries to match a term t against a pattern p , resulting
 253 in a metasubstitution σ when it succeeds. Each pattern constructor can match against the
 254 associated term constructor, trying to match recursively, and pattern variable match against
 255 any term to fill the metasubstitution. It is denoted as $t \mid p \{\sigma\}$ and is formally defined in
 256 Figure 4. Actually, pattern matching can also match universes against universe variables and
 257 sort qualities against quality variables, so the resulting σ contains in fact a metasubstitution,
 258 a universe substitution and a sort quality substitution. These last two substitutions are
 259 defined transparently on terms, only having effect on the quality and universe variables they
 260 carry. Pattern matching verifies one crucial property: $\forall p t \sigma, t \mid p \{\sigma\} \implies t \equiv_\alpha p \{\sigma\}$. This

261 α -conversion would be an equality if not for the fact that some technical information is lost
 262 by pattern matching, such as names of binders.

263 Note that patterns are as heavily annotated as terms, but in this case it is required in
 264 the normal course of operations. Indeed, during the typing pass, we need to give a type to
 265 argument patterns so they can be provided to all pattern variables. This means that all type
 266 fields need an annotation to name the quality and universe of said type. These annotations
 267 can however not be used during pattern matching, as they contain no computational content;
 268 the quality and universe that are matched from the virtual annotation of terms can be
 269 considered virtual as well. These type annotations are used to avoid the issue presented
 270 in Example 2 where the type of a pattern variable may vary depending on the term being
 271 matched.

272 **4 A Criterion to Guarantee that Rewrite Rules are Type Preserving**

273 This section presents the criterion to ensure that rewrite rules preserve typing. To be
 274 able to prove any property surrounding subject reduction, we first have to ensure crucial
 275 properties like injectivity of product types, which are the consequences of the completeness
 276 of algorithmic cumulativity (defined as \rightarrow^* ; \leq_α ; \leftarrow^*). Informally, this means that for a given
 277 proof of cumulativity, we have to build a standardization of it that starts with repeated \rightarrow ,
 278 then repeated \leq_α , then repeated \leftarrow (that is, we reduce both sides before applying \leq_α). To
 279 achieve this, we prove in Section 4.1 that we can move the different relations around, which
 280 needs confluence and postponement of α -cumulativity after reduction. This idea follows
 281 exactly the proof performed in MetaCoq [14].

282 Then, we define in Section 4.2 (bidirectional) type inference of patterns which is at the
 283 heart of the definition of our criterion for subject reduction. Section 4.3 is devoted to the
 284 proof that the criterion indeed ensures subject reduction.

285 **4.1 Confluence and Postponement of α -cumulativity**

286 We first turn to the proof of confluence and postponement of α -cumulativity after reduction.
 287 To ensure confluence, we want to use the triangle criterion [3]. The criterion only applies
 288 with left-linear rewrite rules, which means that patterns variable will be required to appear
 289 at most once in the pattern. The triangle criterion on rewrite rules says that when $t \mid p \{ \sigma \}$,
 290 with $(p \rightsquigarrow r) \in \Sigma$ and $t' \mid p' \{ \sigma' \}$ with $(p' \rightsquigarrow r') \in \Sigma$ for t' a subterm of t ($t = C[t']$) not
 291 in σ (so inspected by pattern p), then $C[r' \{ \sigma' \}] \Rightarrow r \{ \sigma \}$ where \Rightarrow is the parallel reduction
 292 (reduction of several redexes at the same time).

293 However, since PCUIC and the grammar of patterns are richer than the theory of Cockx
 294 *et al.*, there are additional concerns we have to consider. First, the pattern syntax is rich
 295 enough to allow for β - and ι -redexes, which introduce critical pairs with the rewrite rule
 296 being created. One way to fix the issue would be to include these reductions in the triangle
 297 criterion, but since the reducts do not fit as patterns, this would be as restrictive as outright
 298 forbidding them, the approach we chose. Second, our system includes the sort **SProp** and
 299 its conversion rule, so rewriting needs to account for it. This means that patterns whose
 300 type is in **SProp** (or equivalently, patterns which are syntactically irrelevant) can never be
 301 inspected reliably since the term could always be swapped by any other term of the same
 302 type. Therefore, irrelevant patterns need to be forbidden as well. Note that pattern holes
 303 may appear at irrelevant positions, since they do not inspect the term and thus do not
 304 conflict with the conversion rule, so the ban is specifically on patterns and not on argument
 305 patterns.

$$\begin{array}{c}
\frac{(\Gamma \vdash ?x : A) \in \Theta}{\Sigma; \Theta; \Gamma \vdash_p ?x \triangleleft A} \text{PATVAR} \qquad \frac{\Sigma; \Theta; \Gamma \vdash_p p \triangleright A \quad \Sigma; \Theta; \Gamma \vdash_p A \leq_p B}{\Sigma; \Theta; \Gamma \vdash_p p \triangleleft B} \text{CONV} \\
\\
\frac{(x : A) \in \Gamma}{\Sigma; \Theta; \Gamma \vdash_p x \triangleright A} \text{VAR} \qquad \frac{(C : T) \in \Sigma}{\Sigma; \Theta; \Gamma \vdash_p C \triangleright T} \text{AX-SYMB} \qquad \frac{}{\Sigma; \Theta; \Gamma \vdash_p \square_u^q \triangleright \square_{\uparrow_q u}^{\text{Type}}} \text{SORT} \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash_p p \triangleright \forall(x : A), B \quad \Sigma; \Theta; \Gamma \vdash_p a \triangleleft A}{\Sigma; \Theta; \Gamma \vdash_p p a \triangleright B[x := a]} \text{APP} \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash_p a \triangleleft \square_u^q \quad \Sigma; \Theta; \Gamma, (x : a) \vdash_p p \triangleright B}{\Sigma; \Theta; \Gamma \vdash_p \lambda(x : a : \square_u^q), p \triangleright \forall(x : a), B} \text{LAMBDA} \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash_p a \triangleleft \square_u^q \quad \Sigma; \Theta; \Gamma, (x : a) \vdash_p b \triangleleft \square_{u'}^{q'}}{\Sigma; \Theta; \Gamma \vdash_p \forall(x : a : \square_u^q), b : \square_{u'}^{q'} \triangleright \square_{u *_{q'} u'}^{q'}} \text{FORALL} \\
\\
\frac{(\Gamma_p, \Gamma_i, [I : \forall \Gamma_p, \forall \Gamma_i, \square_{u_I}^{q_I}], \vec{\Gamma}_k, [C_k : \forall(\vec{p} : \Gamma_p), \forall \Gamma_k, I \vec{p} i_k]_k) \in \Sigma \quad \Sigma; \Theta; \Gamma \vdash_p p \triangleright I \vec{p} \vec{i}}{\Sigma; \Theta; \Gamma, (\vec{i} : \Gamma_i), (x : I \vec{p} \vec{i}) \vdash_p a \triangleleft \square_u^q \quad \Sigma; \Theta; \Gamma, \Gamma_k \vdash_p b_k \triangleleft a[\Gamma_i := \vec{i}, x := C_k \vec{p} i_k]} \text{CASE} \\
\Sigma; \Theta; \Gamma \vdash_p \text{case } p \text{ return } a : \square_u^q \text{ with } \vec{b}_k \triangleright a[\Gamma_i := \vec{i}, x := p]
\end{array}$$

■ Figure 5 Type inference of patterns

306 Once this is done, the proof of confluence follows exactly what Cockx *et al.* did, which
307 was already a variation on the proof in MetaCoq for PCUIC [14].

308 Let us now prove postponement of α -cumulativity after reduction. The use of α -
309 cumulativity needs to be moved after all reductions, so we need the following property:
310 $t \rightarrow t' \wedge t \leq_\alpha u \implies \exists u', u \rightarrow^* u' \wedge t' \leq_\alpha u'$ (and the same one if we change the direction of
311 \leq_α). Starting from the proof for regular PCUIC which does an induction on $t \rightarrow t'$, the only
312 additional case of reduction is the application of a rewrite rule. So long as \leq_α consists of
313 congruence rules and cumulativity, there is no issue for reapplying the rewrite rule on u (\leq_α
314 can be postponed after pattern matching and is stable under metasubstitutions). However,
315 when the irrelevance rule of **SProp** is used, we are only able to reapply the rewrite rule if the
316 pattern doesn't inspect the irrelevant subterm. With the restriction introduced above, we
317 ensure that the rewrite rule can be reapplied to u once more.

318 4.2 Type Inference of Patterns

319 As mentioned in Section 1, the typing criterion on rewrite rules to ensure subject reduction
320 cannot be solely based on the typing judgment of PCUIC. Actually, what needs to be checked
321 is that the most general type that can be given to a term matching a pattern is a valid type
322 for the right-hand side of the rewrite rule, when its variables also have the most general type
323 allowed by the pattern.

324 To define this formally, we introduce the notion of type inference in a pattern. This
325 notion is based on bidirectional typing as presented for instance in [10]. Technically, the
326 type of a pattern will be inferred, denoted as $\Sigma; \Theta; \Gamma \vdash_p p \triangleright T$, while the type of an argument
327 pattern will only be checked, denoted as $\Sigma; \Theta; \Gamma \vdash_p a \triangleleft T$. The rules for pattern inference are

328 given in Figure 5.

329 Most of the rules follow the standard bidirectional discipline. For instance, the type of a
330 variable (**VAR**) is directly inferred from the local context, and the type of an application (**APP**)
331 is inferred by inferring the type for the function and checking the type of the argument
332 pattern against the domain of the function.

333 Let us remark once again that the sort annotations which correspond to the virtual
334 annotations of terms (*e.g.*, the domain in rule (**FORALL**)) are needed to check the associated
335 type fields, whereas the usual discipline would have been to infer them and coerce them to
336 sorts.

337 The main specificity with respect of standard bidirectional typing lies in the rule (**PATVAR**)
338 for checking a pattern variable. In our presentation, we consider that Θ is given and mentions
339 exactly the pattern variables occurring in the pattern. The rule then checks that the type
340 mentioned in Θ is exactly the one that is checked. Since our criterion needs for pattern
341 variables to have the most general type they can have, we need this rule to be in checking
342 mode. In an algorithmic presentation, Θ can in fact be constructed solely from the typing
343 of the pattern and can then be considered an output of the typing procedure, along with the
344 inferred type of the whole pattern.

345 One last crucial rule is (**CONV**), which allows changing bidirectional modes. It makes use
346 of the pattern cumulativity relation, which will be characterised and defined in Section 5, but
347 can for the moment be approximated with regular term cumulativity (they both compare
348 regular terms).

349 Using this notion of type inference of patterns, we define the following criterion for
350 subject reduction.

351 ► **Definition 4** (Type-preservation criterion). *A rewrite rule $p \rightsquigarrow r$ is said to be type-preserving*
352 *in the global environment Σ when there exists a metavariable context Θ and type T_p such*
353 *that $\Sigma; \Theta; [] \vdash_p p \triangleright T_p$ and $\Sigma; \Theta; [] \vdash r : T_p$.*

354 We now turn to the proof that this criterion is enough to deduce subject reduction.

355 4.3 Subject Reduction

356 In this section, we consider a fixed rewrite rule $p \rightsquigarrow r$ that satisfies the type-preservation
357 criterion in environment Σ_0 for the metavariable environment Θ at type T_p . To show that the
358 type system has subject reduction, we have to show that the rewrite rule is type preserving
359 in all extended environments and all local contexts, which means:

$$360 \quad \forall \Sigma \supseteq \Sigma_0, \Gamma, t, T, \sigma, \Sigma; []; \Gamma \vdash t : T \wedge t \mid p\{\sigma\} \implies \Sigma; []; \Gamma \vdash r\{\sigma\} : T.$$

361 We first determine the properties that type inference \vdash_p needs to satisfy before proving
362 the implication. Let us fix an environment $\Sigma \supseteq \Sigma_0$ and a local context Γ .

- 363 1. We need to work under environment Σ instead of Σ_0 . Fortunately, \vdash satisfies environment
364 weakening and we can verify that \vdash_p also satisfies it, so we can really work under
365 assumptions $\Sigma; \Theta; [] \vdash_p p \triangleright T_p$ and $\Sigma; \Theta; [] \vdash r : T_p$.
- 366 2. We can use the substitutivity property of \vdash and the fact that types are well typed to remove
367 all mentions of r in our implication: since $\forall \sigma, \Sigma; []; \Gamma \vdash \sigma : \Theta \implies \Sigma; []; \Gamma \vdash r\{\sigma\} : T_p\{\sigma\}$
368 and $\Sigma; []; \Gamma \vdash T : \Box_u^q$ for some q and u , implying that $(\Sigma; []; \Gamma \vdash r\{\sigma\} : T_p\{\sigma\} \wedge$
369 $T_p\{\sigma\} \leq T) \implies \Sigma; []; \Gamma \vdash r\{\sigma\} : T$, it only remains to prove the following anti-
370 substitution lemma:

$$371 \quad \forall t, T, \sigma, \Sigma; []; \Gamma \vdash t : T \wedge t \mid p\{\sigma\} \implies \Sigma; []; \Gamma \vdash \sigma : \Theta \wedge \Sigma; []; \Gamma \vdash T_p\{\sigma\} \leq T.$$

??:12 The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant

- 372 3. We need σ to completely instantiate Θ , so we make use of the fact that Θ contains exactly
 373 one entry per pattern variable of p .
- 374 4. Since the types in Θ and T_p can refer to the additional sort annotations in patterns, we
 375 need to consider the virtual parts of t and σ in the following.

376 It is well known that the typing judgment of PCUIC satisfies environment weakening [14].
 377 In fact, our extension with metavariables changes nothing to the proof (the rule on metavariables
 378 doesn't mention the environment), so the property still stands in our system. Even
 379 better, since our pattern typing judgment resembles regular typing, the same proof can be
 380 adapted to work on it, proving that pattern typing also verifies environment weakening. Let
 381 us now prove the anti-substitution lemma.

382 We first need to introduce more elements and strengthen our property so that the
 383 induction can go through. We first split Γ into Γ, Δ and σ into σ_0, σ such that Γ is fixed
 384 while Δ is the context extension which corresponds to Δ_p as we go into the pattern and σ_0
 385 corresponds to the already matched portion of the pattern, which will affect Δ and Θ while
 386 σ corresponds to the currently matched portion of the pattern. The proof goes by induction
 387 on the typing derivation of p , with induction predicate:

$$\begin{aligned}
 388 \quad & \Sigma; \Theta; \Delta_p \vdash_p p \triangleright T_p \implies \\
 389 \quad & \forall \Delta, t, T, \sigma_0, \sigma, \Sigma; []; \Gamma, \Delta \vdash t : T \wedge t \mid p\{\sigma\} \wedge \Delta_p\{\sigma_0\} \equiv \Delta \implies \\
 390 \quad & \Sigma; []; \Gamma \vdash \sigma : \Theta\{\sigma_0\} \wedge T_p\{\sigma_0, \sigma\} \leq T \\
 391 \\
 392
 \end{aligned}$$

393 and the corresponding predicate on argument patterns:

$$\begin{aligned}
 394 \quad & \Sigma; \Theta; \Delta_p \vdash_p p \triangleleft T_p \implies \\
 395 \quad & \forall \Delta, t, T, \sigma_0, \sigma, \Sigma; []; \Gamma, \Delta \vdash t : T \wedge t \mid p\{\sigma\} \wedge T_p\{\sigma_0, \sigma\} \equiv T \wedge \Delta_p\{\sigma_0\} \equiv \Delta \implies \\
 396 \quad & \Sigma; []; \Gamma \vdash \sigma : \Theta\{\sigma_0\} \\
 397 \\
 398
 \end{aligned}$$

399 Let us give a representative subset of all cases needed to prove the induction.

400 ■ Case $?x$:

401 Hypotheses: $\Sigma; []; \Gamma, \Delta \vdash t : T, \sigma = [?x := t], T_p\{\sigma_0\} \equiv T, \Delta_p\{\sigma_0\} \equiv \Delta$
 402 Goal: $\Sigma; []; \Gamma \vdash (\Delta_p \vdash ?x : T_p)\{\sigma_0, \sigma\}$ which simplifies to $\Sigma; []; \Gamma, \Delta_p\{\sigma_0, \sigma\} \vdash t : T_p\{\sigma_0, \sigma\}$
 403 Proof: Neither Δ_p nor T_p mention $?x$, so they are respectively convertible to Δ and T by
 404 hypothesis, which proves our goal.

405 ■ Case $p a$

406 Hypotheses: $\Sigma; \Theta; \Delta \vdash_p p \triangleright \forall(x : A_p), B_p, \quad \Sigma; \Theta; \Delta \vdash_p a \triangleleft A_p, \quad T_p = B_p[x := a],$
 407 $\Sigma; []; \Gamma, \Delta \vdash f t : T, f \mid p\{\sigma_1\}, t \mid a\{\sigma_2\}, \Delta_p\{\sigma_0\} \equiv \Delta.$

408 By inversion of typing on an application, there exists A and B such that $\forall(x : A), B \leq T,$
 409 $\Sigma; []; \Gamma, \Delta \vdash f : \forall x : A, B$ and $\Sigma; []; \Gamma, \Delta \vdash t : A.$

410 Then, by induction hypothesis on p , we get $(\forall x : A_p, B_p)\{\sigma_0, \sigma_1\} \leq \forall x : A, B$ and
 411 $\Sigma; []; \Gamma \vdash \sigma_1 : \Theta\{\sigma_0\}.$

412 By the definition of substitution and injectivity of products for \leq , $A_p\{\sigma_0, \sigma_1\} \equiv A$ and
 413 $B_p\{\sigma_0, \sigma_1\} \leq B.$ Since Δ_p does not contain any metavariable in $|\sigma_1|$, $\Delta_p\{\sigma_0, \sigma_1\} \equiv \Delta.$

414 This means we can use the induction hypothesis on a and get $\Sigma; []; \Gamma \vdash \sigma_2 : \Theta\{\sigma_0, \sigma_1\}$

415 Goal: $B_p[x := a]\{\sigma\} \leq B[x := t]$ and $\Sigma; []; \Gamma \vdash \sigma_1, \sigma_2 : \Theta\{\sigma_0\}.$ The second goal is
 416 immediately proved by concatenating the induction hypotheses.

Proof: by commuting the substitution and metasubstitution,

$$B_p[x := a]\{\sigma\} = B_p\{\sigma\}[x := a\{\sigma\}] \equiv B_p\{\sigma\}[x := t] \leq B[x := t]$$

417 (a contains only metavariables in $|\sigma_2|$ and $t \mid a\{\sigma_2\}$, so $a\{\sigma\} = a\{\sigma_2\} \equiv t$)

418 ■ Case $\forall(x : a : \square_{?u}^{?q}), b : \square_{?u'}^{?q'}$
 419 Hypotheses: $\Sigma; \Theta; \Delta \vdash_p a \triangleleft \square_{?u}^{?q}, \Sigma; \Theta; \Delta \vdash_p b \triangleleft \square_{?u'}^{?q'}, T_p = \square_{?u * ?q, ?u'}^{?q'}$,
 420 $\Sigma; []; \Gamma, \Delta \vdash \forall(x : A : \square_u^q), (B : \square_{u'}^{q'}) : T, A^+ \mid a \{\sigma_1\}, B^+ \mid b \{\sigma_2\}, \Delta_p \{\sigma_0\} \equiv \Delta$ with
 421 $\sigma = [?q := q; ?q' := q'; ?u := u; ?u' := u'] + \sigma_1 + \sigma_2$.
 422 By inversion of typing on a product, we get $\square_{u * ?q, u'}^{q'} \leq T, \Sigma; []; \Gamma, \Delta \vdash A : \square_u^q$ and
 423 $\Sigma; []; \Gamma, \Delta, (x : A) \vdash B : \square_{u'}^{q'}$.
 424 Since Δ_p does not contain any metavariable in $|\sigma_1|$, $(\Delta_p, x : a) \{\sigma_0, \sigma_1\} \equiv \Delta, (x : A)$.
 425 This means we can use both induction hypotheses to get $\Sigma; []; \Gamma \vdash \sigma_1 : \Theta \{\sigma_0\}$ and
 426 $\Sigma; []; \Gamma \vdash \sigma_2 : \Theta \{\sigma_0, \sigma_1\}$
 427 Goal: $\square_{?u * ?q, ?u'}^{?q'} \{\sigma\} \leq \square_{u * ?q, u'}^{q'} \{\sigma\}$ and $\Sigma; []; \Gamma \vdash \sigma_1, \sigma_2 : \Theta \{\sigma_0\}$. The second goal is immedi-
 428 ately proved by concatenating the induction hypotheses.
 429 Proof: We didn't define yet how $*$ should behave when its quality arguments are variables.
 430 Evidently, they need to return the minimal universe, so they must behave as **Prop** to
 431 satisfy the required inequality.
 432 ■ Case **CONV**:
 433 Hypotheses: $\Sigma; []; \Gamma, \Delta \vdash t : T, t \mid p \{\sigma\}, T'_p \{\sigma_0\} \equiv T, \Delta_p \{\sigma_0\} \equiv \Delta, T_p \{\sigma\} \leq T, T_p \leq_p T'_p$,
 434 $\Sigma; []; \Gamma \vdash \sigma : \Theta \{\sigma_0\}$.
 435 Goal: $\Sigma; []; \Gamma \vdash \sigma : \Theta \{\sigma_0\}$. It is one of our hypotheses, our cumulativity hypotheses are
 436 useless for now. In Section 5 however, we present the definition of pattern cumulativity
 437 to extract some information from these unused hypotheses. Let us simply note that, by
 438 symmetry of \equiv and transitivity, $T_p \{\sigma\} \leq T'_p \{\sigma\}$, which we will use alongside $T_p \leq_p T'_p$.

5 Pattern Cumulativity and Equality Extraction

Currently, our criterion is more limited than the one we seek. For instance, it doesn't allow us to infer the type of slightly complex rewrite rules on vectors of natural numbers like

$$\mathbf{vtail} \ ?n \ (\mathbf{vcons} \ ?n' \ ?a \ ?v) \rightsquigarrow \ ?v$$

440 since the pattern has type **vector** $?n$ while the replacement term has type **vector** $?n'$.
 441 However, typing ensures that we always have $?n = ?n'$, so we could extract this equality
 442 during pattern typing, in the same way that unification is used to collect constraints during
 443 the elaboration of a **Coq** term [16]. This way, we can modify the subject reduction criterion
 444 to take advantage of these additional conversion hypotheses when checking the type of the
 445 right-hand side of the rewrite rule.

446 We proceed by defining a notion of pattern conversion and cumulativity (\equiv_p and \leq_p) that
 447 collect the set of equalities \mathcal{E} necessarily satisfied by substituted terms: we replace the binary
 448 relation $t \leq_p u$ with the ternary relation $t \leq_p u \triangleright \mathcal{E}$, which satisfies the following property:

$$\begin{aligned} \forall t, u, \mathcal{E}, \quad \Sigma_0; \Theta; \Delta_p \vdash_p t \leq_p u \triangleright \mathcal{E} &\implies \\ \forall \Sigma \supseteq \Sigma_0, \Xi, \Gamma, \sigma, \quad \Sigma; \Xi; \Gamma \vdash \sigma : \Theta \wedge \Sigma; \Xi; \Gamma, \Delta_p \{\sigma\} \vdash t \{\sigma\} \leq u \{\sigma\} & \\ \implies \forall (\Delta_i \vdash t_i \equiv u_i) \in \mathcal{E}, \quad \Sigma; \Xi; \Gamma, \Delta \{\sigma\} \vdash t_i \{\sigma\} \equiv u_i \{\sigma\}. & \end{aligned}$$

450 Note that we are still very free in the implementation of \leq_p because, as long as we
 451 don't claim any equality ($\mathcal{E} = []$), even the full relation would be sound. However, we must
 452 remember that our primary goal is to extract as many enforced equalities as possible.

453 Our main objective is thus to apply safe inversions of conversion, to make sure that our
 454 equalities necessarily hold:

??:14 The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant

$$\begin{array}{c}
\frac{\Sigma; \Theta; \Gamma \vdash t \rightarrow^* t' \quad \Sigma; \Theta; \Gamma \vdash u \rightarrow^* u' \quad \Sigma; \Theta; \Gamma \vdash_p t' \leq_p u' \triangleright \mathcal{E}}{\Sigma; \Theta; \Gamma \vdash_p t \leq_p u \triangleright \mathcal{E}} \text{RED} \\
\\
\frac{C \text{ is injective}}{\Sigma; \Theta; \Gamma \vdash_p C \leq_p C \triangleright []} \text{INJ} \qquad \frac{x \in \Gamma}{\Sigma; \Theta; \Gamma \vdash_p x \leq_p x \triangleright []} \text{REL} \\
\\
\frac{\Sigma \vdash q =_q q' \wedge u \leq_u u' \text{ possible}}{\Sigma; \Theta; \Gamma \vdash_p \square_u^q \leq_p \square_{u'}^{q'} \triangleright [u \leq_u u'; q =_q q']} \text{SORT} \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash_p A \equiv_p B \triangleright \mathcal{E} \quad \Sigma; \Theta; \Gamma, (x : A) \vdash_p t \leq_p u \triangleright \mathcal{E}' \quad \mathcal{B} \in \{\lambda, \forall\}}{\Sigma; \Theta; \Gamma \vdash_p \mathcal{B}(x : A), t \leq_p \mathcal{B}(x : B), u \triangleright \mathcal{E} \cup \mathcal{E}'} \text{LAMBDA/FORALL} \\
\\
\frac{\Sigma; \Theta; \Gamma \vdash_p f \leq_p f' \triangleright \mathcal{E} \quad \Sigma; \Theta; \Gamma \vdash_p a \equiv_p a' \triangleright \mathcal{E}' \quad \Sigma; \Gamma \vdash f, f' \text{ whne}}{\Sigma; \Theta; \Gamma \vdash_p f a \leq_p f' a' \triangleright \mathcal{E} \cup \mathcal{E}'} \text{APP} \\
\\
\frac{\Sigma; \Theta; \Gamma, \Gamma_i \vdash_p P \equiv_p P' \triangleright \mathcal{E}_r \quad \begin{array}{c} \Sigma; \Theta; \Gamma \vdash_p c \equiv_p c' \triangleright \mathcal{E}_c \\ [\Sigma; \Theta; \Gamma, \Gamma_k \vdash_p b_k \equiv_p b'_k \triangleright \mathcal{E}_{b,k}] \end{array} \quad \Sigma; \Gamma \vdash c, c' \text{ whne}}{\Sigma; \Theta; \Gamma \vdash_p \text{case } c \text{ return } P \text{ with } \vec{b} \leq_p \text{case } c' \text{ return } P' \text{ with } \vec{b}' \triangleright \mathcal{E}_c \cup \mathcal{E}_r \cup \bigcup_i \mathcal{E}_{b,i}} \text{CASE} \\
\\
\frac{}{\Sigma; \Theta; \Gamma \vdash_p ?x \equiv_p t \triangleright [\Delta \vdash ?x := t]} \text{PATVAR} \\
\\
\frac{t \text{ or } u \text{ contains a pattern variable or a symbol in head position}}{\Sigma; \Theta; \Gamma \vdash_p t \leq_p u \triangleright []} \text{FAILSAFE} \\
\\
\frac{x \in \Gamma}{\Sigma; \Gamma \vdash x \text{ whne}} \quad \frac{\Sigma; \Gamma \vdash f \text{ whne}}{\Sigma; \Gamma \vdash f a \text{ whne}} \quad \frac{\Sigma; \Gamma \vdash c \text{ whne}}{\Sigma; \Gamma \vdash \text{case } c \text{ return } P \text{ with } \vec{b} \text{ whne}}
\end{array}$$

■ **Figure 6** The rules for conversion judgments \leq_p and \equiv_p (replace all \leq_p with \equiv_p in the rules)

- 455 ■ If $t_1\{\sigma\} \equiv u$ and $t_1 \rightarrow^* t_2$, then by reflexivity of \equiv and substitutivity $t_1\{\sigma\} \equiv t_2\{\sigma\}$ and
- 456 thus $t_2\{\sigma\} \equiv u$. This means that we are allowed to reduce in \equiv_p .
- 457 ■ When a term is in weak head normal form, its congruence rule is invertible. This includes
- 458 products, sorts, λ -abstractions, inductives, constructors and all weak head neutral terms
- 459 (whne). Therefore, the general strategy for pattern conversion will be to reduce our
- 460 arguments to weak head normal form and then to invert the congruence rule of the head
- 461 constructor.
- 462 ■ Non-injective symbols may be equipped with any rule in the future, so the congruence
- 463 rules of their elimination context (as would-be neutral terms) are not invertible. This
- 464 means we cannot extract information in this case and should simply accept with no
- 465 equality. This approach is a way to avoid the failure of subject reduction described in
- 466 Example 3.
- 467 ■ Metavariables may take any (well-typed) value, so the congruence rules of their elimination
- 468 context (as would-be neutral terms) are not invertible. However, the conversion can

469 be extracted as an equality. To keep the procedure easily decidable, we only keep the
 470 equality when the elimination context is empty ($?x\{\Delta := \Delta\} \equiv t$). Also, if we extract
 471 more than one equality for a pattern variable, we will discard all but one.

472 The definition of $t \leq_p u \triangleright \mathcal{E}$ is formally given in Figure 6. We use this notion of
 473 cumulativity on patterns producing equality constraints to extend inference of patterns to
 474 also produce a set of equalities: $\Sigma; \Theta; [] \vdash_p p \triangleright T_p, \mathcal{E}$.

475 In the setting we consider, all equations are of the form $\Delta \vdash ?x := a$ and can then be used in
 476 the regular typing judgment by adding reduction $(\Delta \vdash ?x := a) \in \mathcal{E} \implies ?x\{\bar{t}\} \rightarrow a[\Delta := \bar{t}]$
 477 in the definition of cumulativity. We note $\vdash_{\mathcal{E}}$ the typing judgment where cumulativity is
 478 extended with the equalities in \mathcal{E} .

479 ► **Definition 5** (Extended type-preservation criterion). *A rewrite rule $p \rightsquigarrow r$ is said to be*
 480 *type-preserving in the global environment Σ when there exists a metavariable context Θ and*
 481 *type T_p and set of equalities \mathcal{E} such that $\Sigma; \Theta; [] \vdash_p p \triangleright T_p, \mathcal{E}$ and $\Sigma; \Theta; [] \vdash_{\mathcal{E}} r : T_p$.*

482 The proof of subject reduction can be extended easily to this setting.

483 6 Implementation

484 Rewrite rules have been implemented and integrated into the Coq proof assistant with [pull](#)
 485 [request #18038](#). The criterion for type preservation has been implemented but is yet to be
 486 integrated to the Coq proof assistant. A few extensions were made from the theory exposed
 487 in this paper.

488 The implementation supports the full grammar of Coq terms, including notably primitive
 489 projections, universe polymorphism and sort polymorphism which can be matched against.

490 An extension has been made so that users can make some symbols unfold fixpoints. In
 491 Coq, fixpoints are guarded to prevent infinite loops and the guard condition operates on
 492 inductive values. This means that fixpoint unfolding may only happen when the guarded
 493 argument has a head constructor. However, a user may want to add new values of an
 494 inductive type that should behave as a constructor in this regard, usually along with a rule
 495 to describe the reduction on a case. To this end, a flag called `unfold_fix` can be given when
 496 declaring a symbol, as for instance in the following example declaring an exception in the
 497 type of natural numbers [12].

```
498 #[unfold_fix] Symbol error : N.  
499 Rewrite Rule error_case :=  
500   match error return N with 0 => _ | S _ => _ end ~> error.  
501 Eval compute in error + 5. (* error *)  
502  
503
```

504 In practice, metavariables can be used to bind terms on the left-hand side and mention
 505 them in the right-hand side, but they are also convenient to avoid describing explicitly
 506 sub-patterns that are ‘forced’ and will be found by the extraction of equalities mechanism
 507 described in Section 5. To this end, we have added the `_` to produce metavariables that are
 508 not relevant to the right-hand side, as the two branches of the example above.

In this setting, another extension has been added to the type preservation criterion to
 mitigate the inference requirements for application and cases: while the theory requires that
 the pattern infers respectively a product and an inductive, it is in fact safe to allow inferring a
 pattern variable. This way, the user can still put `_` at places where a product or an inductive
 is expected, and we automatically add an equality between this implicit pattern variable and
 the product/inductive with ‘fresh’ pattern variables (respectively for the domain/codomain

and for the arguments to the inductive). Since we cannot generate fresh variables during type checking, these additional pattern variables have to be added to the pattern from the start, giving for application a virtual annotation $(p : \forall(x : ?A), ?B) a$, with typing rule

$$\frac{\Sigma; \Theta; \Gamma \vdash p \triangleleft \forall(x : ?A), ?B \quad \Sigma; \Theta; \Gamma \vdash a \triangleleft ?A}{\Sigma; \Theta; \Gamma \vdash (p : \forall(x : ?A), ?B) a \triangleright ?B\{x := a\}}$$

7 Related Work

In Andromeda 2, users manually describe definitional equality by providing equality judgments, which must either be extensional equalities or get interpreted as rewrite rules [1]. During this transformation, the system checks type preservation in a manner similar to ours, but since their system is more permissive, it has less guarantees (such as substitutivity which always holds for us) and thus needs to check equalities before applying a rewrite rule.

Felicissimo introduces a bidirectional system with rewriting [6], where rewrite rules are also checked to preserve typing. Both systems are close, but our pattern grammar is more expressive, we have fewer constraints on erased arguments (*e.g.*, our virtual sort annotations can't be recovered from the type of the products) and we extract equalities from the typing of the pattern. The bidirectional approach has some interesting advantages such as the possibility to implement rewrite rules that would typically be non-left-linear. In the case of Coq it would require a redesign of the syntax and thus it's not clear how applicable it would be to our case.

Blanqui's decision procedure for type safety in rewrite rules [2] also checks type preservation and extracts equalities in a simpler theory, the $\lambda\Pi$ -calculus, thus without a (cumulative) hierarchy of universes nor inductive types. In particular, there is no notion of subtyping and thus no need for bidirectional inference. However, this simpler setting makes it possible to extract more enforced equalities from the pattern.

8 Future Work

From a practical point of view, our most immediate goal is to integrate the criterion for type preservation to the Coq proof assistant, as well as port to Coq the confluence checker that has been written by Jesper Cockx for Agda, to greatly improve the safety of rewrite rules.

Another meaningful extension is to define a notion of justified rewrite rules, which corresponds to the mapping of existing terms to the declaration of symbols and a proof of propositional equality on those terms to rewrite rules on those symbols. For instance, parallel `pplus` is justified with the standard notion of addition. This way, justified rewrite rules can be considered as a safe extension instead of their current status of additional axioms.

References

- 1 Andrej Bauer and Anja Petković Komel. An extensible equality checking algorithm for dependent type theories. *Logical Methods in Computer Science*, Volume 18, Issue 1, January 2022. doi:10.46298/lmcs-18(1:17)2022.
- 2 Frédéric Blanqui. Type Safety of Rewrite Rules in Dependent Types. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSCD.2020.13.
- 3 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*, 5(POPL):60:1–60:29, January 2021. doi:10.1145/3434341.

- 547 4 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and
548 Grigori Mints, editors, *COLOG-88*, pages 50–66, Berlin, Heidelberg, 1990. Springer. doi:
549 10.1007/3-540-52335-9_47.
- 550 5 Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Lan-
551 guage. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages
552 625–635, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-79876-5_
553 37.
- 554 6 Thiago Felicissimo. Generic bidirectional typing for dependent type theories, October 2023.
555 arXiv:2307.08523.
- 556 7 Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-
557 irrelevance without K. *Proceedings of the ACM on Programming Languages*, 3(POPL):3:1–3:28,
558 January 2019. doi:10.1145/3290316.
- 559 8 Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination Des Coupures de l’arithmétique*
560 *d’ordre Supérieur*. PhD thesis, Université Paris VII, 1972.
- 561 9 Antonius J. C. Hurkens. A simplification of Girard’s paradox. In Mariangiola Dezani-Ciancaglini
562 and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 266–278, Berlin,
563 Heidelberg, 1995. Springer. doi:10.1007/BFb0014058.
- 564 10 Meven Lennon-Bertrand. Complete Bidirectional Typing for the Calculus of Inductive Con-
565 structions. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ITP.2021.24*. Schloss Dagstuhl –
566 Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.24.
- 567 11 Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*,
568 volume 32. Citeseer, 2007.
- 569 12 Pierre-Marie Pédro and Nicolas Tabareau. Failure is not an option an exceptional type theory.
570 In *ESOP 2018 - 27th European Symposium on Programming*, volume 10801 of *LNCS*, pages
571 245–271, Thessaloniki, Greece, April 2018. Springer. doi:10.1007/978-3-319-89884-1_9.
- 572 13 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian
573 Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project.
574 *Journal of Automated Reasoning*, 2020. doi:10.1007/s10817-019-09540-0.
- 575 14 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter.
576 Coq Coq correct! Verification of type checking and erasure for Coq, in Coq. *PACMPL*,
577 4(POPL), 2020. doi:10.1145/3371076.
- 578 15 The Coq Development Team. The Coq Proof Assistant. Zenodo, June 2023. doi:10.5281/
579 zenodo.8161141.
- 580 16 Beta Ziliani and Matthieu Sozeau. A unification algorithm for Coq featuring universe poly-
581 morphism and overloading. In *Proceedings of the 20th ACM SIGPLAN International Conference*
582 *on Functional Programming*, ICFP 2015, pages 179–191, New York, NY, USA, August 2015.
583 Association for Computing Machinery. doi:10.1145/2784731.2784751.