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

Yann Leray, Gaëtan Gilbert, Nicolas Tabareau, Théo Winterhalter

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

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

```
Axiom U : Type.
Symbol El : U → Type.
Symbols (N : U) (Pi : ∀ a : U, (El u → U) → U).
Symbol U_rect : ∀(P : U → Type), P N → (∀ a b, (∀ A : El a, P (b A)) → P (Pi a b)) → ∀ u, P u.
Rewrite Rules U_rect_rew :=
| U_rect ?P ?pN ?pPi N ⇝ ?pN
| U_rect ?P ?pN ?pPi (Pi ?a ?b) ⇝ ?pPi ?a ?b (fun (A : El ?a) ⇒ U_rect ?P ?pN ?pPi (?b A)).
Rewrite Rules El_red :=
| El N ⇝ ℕ
| El (Pi ?a ?b) ⇝ ∀ (A : El ?a), El (?b A).
```

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

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

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

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

```
Symbol id@{u v} : Type@{u} → Type@{v}.
Rewrite Rule id_rew := id ?T ⇝ ?T.
Universe a.
Check id Type@{a} : id Type@{a}.
```

The reason is that the naive check only mandates that both universes share a common bigger universe – something which is always verified. Thus, the check doesn't forbid this rule which 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.

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

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

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

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

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

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

▶ **Example 3.** Consider the following:

```
Inductive Box (b : 𝔹) : Type := box. (* box : ∀ (b : 𝔹), Box b *)
Symbol I : 𝔹 → 𝔹.
Symbol C : ∀ (b : 𝔹), Box b → Box (I b).
Symbol D : ∀ (b : 𝔹), Box (I b) → Box b.
Rewrite Rule D_C := D _ (C _ ?b) ⇝ ?b.

Rewrite Rule I_rew := I _ ⇝ false.

Definition a : Box false := (D false (C _ (box true))).
Eval compute in a. (* box true : Box false *)
```

The rewrite rule `D _ (C _ ?b) ⇝ ?b` looks enticing, if the box is not to be considered as a truncation. However, we don't know the behaviour of `I` at that point, and as we add rule `I _ ⇝ false`, our first rule loses its type preservation property, so it should not in fact have been accepted in the first place.

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

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

$$
\begin{array}{rcll}
A, B, P, T, c, t, \ldots & ::= & x & \text{variable} \\
& | & ?x\{\vec{t}\} & \text{metavariable} \\
& | & s & \text{sort} \\
& | & A & \text{axiom} \\
& | & C & \text{constructor} \\
& | & I & \text{inductive} \\
& | & t\ t' & \text{application} \\
& | & \lambda(x : A : s), t & \text{abstraction} \\
& | & \forall(x : A : s), (B : s') & \text{product} \\
& | & \mathrm{case}\, c\, \mathrm{return}\, P : s\, \mathrm{with}\, \vec{t} & \text{case destruction} \\
& | & \mathrm{fix}\, f : T := t & \text{fixpoint} \\
s, s' & ::= & \square_u^q & \text{sort} \\
q & ::= & \mathsf{SProp} \mid \mathsf{Prop} \mid \mathsf{Type} & \text{qualities}
\end{array}
$$

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

et al. [3] by providing a criterion for type preservation in presence of cumulativity. Rewrite rules as presented in this paper have been implemented in Coq and have been merged to the main branch of its repository[2] meaning that they should be available in the release 8.20 of Coq.

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

2 PCUIC with Metavariables

2.1 Syntax

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

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

[2] `https://github.com/coq/coq/pull/18038`

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

2.2 Typing

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

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

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

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

2.3 Cumulativity

Our typing judgment contains a cumulativity rule, instead of the more traditional conversion rule. Cumulativity is defined similarly to conversion, except that we allow subtyping on sorts: $\square^{q}_{u} \leqslant_s \square^{q'}_{u'}$. Formally, cumulativity is defined as the transitive closure $\leqslant$ of $\rightarrow \cup \leftarrow \cup \leqslant_\alpha$ where $\rightarrow$ is reduction, $\leftarrow$ antireduction (the relation symmetric to $\rightarrow$) and $\leqslant_\alpha$ is α-cumulativity. We also define conversion $\equiv$ by replacing α-cumulativity with α-conversion.

$$\frac{\Sigma;\Theta \vdash \Gamma \qquad (x : T) \in \Gamma}{\Sigma;\Theta;\Gamma \vdash x : T}\ \textsc{Var} \qquad \frac{\Sigma;\Theta \vdash \Gamma \qquad (C : T) \in \Sigma}{\Sigma;\Theta;\Gamma \vdash C : T}\ \textsc{EnvVar}$$

$$\frac{\Sigma;\Theta \vdash \Gamma \qquad (\Gamma_{?x} \vdash ?x : A) \in \Theta \qquad \Sigma;\Theta;\Gamma \vdash \vec{t} : \Gamma_{?x}}{\Sigma;\Theta;\Gamma \vdash ?x\{\vec{t}\} : A}\ \textsc{Meta} \qquad \frac{\Sigma;\Theta \vdash \Gamma}{\Sigma;\Theta;\Gamma \vdash \Box^{q}_{u} : \Box^{\mathsf{Type}}_{\uparrow_q u}}\ \textsc{Sort}$$

$$\frac{\Sigma;\Theta;\Gamma \vdash f : \forall (x : A), B \qquad \Sigma;\Theta;\Gamma \vdash t : A}{\Sigma;\Theta;\Gamma \vdash f\ t : B[x := t]}\ \textsc{App}$$

$$\frac{\Sigma;\Theta;\Gamma \vdash A : \Box^{q}_{u} \qquad \Sigma;\Theta;\Gamma, x : A \vdash t : B}{\Sigma;\Theta;\Gamma \vdash \lambda(x : A : \Box^{q}_{u}), t : \forall (x : A), B}\ \textsc{Lambda}$$

$$\frac{\Sigma;\Theta;\Gamma \vdash A : \Box^{q}_{u} \qquad \Sigma;\Theta;\Gamma, x : A \vdash B : \Box^{q'}_{u'}}{\Sigma;\Theta;\Gamma \vdash \forall (x : A : \Box^{q}_{u}), (B : \Box^{q'}_{u'}) : \Box^{q'}_{u *^{q}_{q'} u'}}\ \textsc{Forall}$$

$$\frac{\begin{array}{c}(\Gamma_p, \Gamma_i, [I : \forall \Gamma_p, \forall \Gamma_i, \Box^{q_I}_{u_I}], \vec{\Gamma_k}, [C_k : \forall (\vec{p} : \Gamma_p), \forall \Gamma_k, I\ \vec{p}\ \vec{\imath_k}]_k) \in \Sigma \qquad \Sigma;\Theta;\Gamma \vdash c : I\ \vec{p}\ \vec{\imath} \\ \Sigma;\Theta;\Gamma, (\vec{\imath} : \Gamma_i), (x : I\ \vec{p}\ \vec{\imath}) \vdash P : \Box^{q}_{u} \qquad \Sigma;\Theta;\Gamma, \Gamma_k \vdash b_k : P[\Gamma_i := \vec{\imath_k}, x := C_k\ \vec{p}\ \vec{\imath_k}]_k\end{array}}{\Sigma;\Theta;\Gamma \vdash \mathrm{case}\, c\, \mathrm{return}\, P : \Box^{q}_{u}\, \mathrm{with}\, \vec{b_k} : P[\Gamma_i := \vec{\imath}, x := c]}\ \textsc{Case}$$

$$\frac{\Sigma;\Theta;\Gamma \vdash T : \Box^{q}_{u} \qquad \Sigma;\Theta;\Gamma, f : T \vdash t : T \qquad f\ \mathsf{guarded}}{\Sigma;\Theta;\Gamma \vdash \mathrm{fix}\, f : T := t : T}\ \textsc{Fix}$$

$$\frac{\Sigma;\Theta;\Gamma \vdash t : A \qquad \Sigma;\Theta;\Gamma \vdash A \leqslant B \qquad \Sigma;\Theta;\Gamma \vdash B : \Box^{q}_{u}}{\Sigma;\Theta;\Gamma \vdash t : B}\ \textsc{Conv}$$

$$\frac{}{\Sigma;\Theta;\Gamma \vdash (\lambda(x : A), t)\ a \to t[x := a]}\ \beta \qquad \frac{}{\Sigma;\Theta;\Gamma \vdash \mathrm{fix}\, f := t \to t[f := \mathrm{fix}\, f := t]}\ \textsc{FixRed}$$

$$\frac{(\Gamma_p, \Gamma_i, [I : \forall \Gamma_p, \forall \Gamma_i, \Box^{q_I}_{u_I}], \vec{\Gamma_k}, [C_k : \forall (\vec{p} : \Gamma_p), \forall \Gamma_i, I\ \vec{p}\ \vec{\imath_k}]_k) \in \Sigma}{\Sigma;\Theta;\Gamma \vdash \mathrm{case}\, C_j\ \vec{p}\ \vec{a}\, \mathrm{return}\, P\, \mathrm{with}\, \vec{b_k} \to b_j[\Gamma_j := \vec{a}]}\ \iota$$

$$\frac{\Sigma;\Theta;\Gamma \vdash \Box^{q}_{u} \leqslant_s \Box^{q'}_{u'}}{\Sigma;\Theta;\Gamma \vdash \Box^{q}_{u} \leqslant_\alpha \Box^{q'}_{u'}} \qquad \frac{\Sigma;\Theta;\Gamma \vdash T : \mathsf{SProp} \qquad \Sigma;\Theta;\Gamma \vdash t : T \qquad \Sigma;\Theta;\Gamma \vdash u : T}{\Sigma;\Theta;\Gamma \vdash t \leqslant_\alpha u}$$

$$\frac{?x \in \Theta \qquad \Sigma;\Theta;\Gamma \vdash \vec{t} \equiv_\alpha \vec{t'}}{\Sigma;\Theta;\Gamma \vdash ?x\{\vec{t}\} \leqslant_\alpha ?x\{\vec{t'}\}} \qquad \frac{\Sigma;\Theta;\Gamma \vdash f \leqslant_\alpha f' \qquad \Sigma;\Theta;\Gamma \vdash a \equiv_\alpha a'}{\Sigma;\Theta;\Gamma \vdash f\ a \leqslant_\alpha f'\ a'}$$

$$\frac{\Sigma;\Theta;\Gamma \vdash A \equiv_\alpha A \qquad \Sigma;\Theta;\Gamma \vdash t \leqslant_\alpha t'}{\Sigma;\Theta;\Gamma \vdash \lambda(x : A : \Box^{q}_{u}), t \leqslant_\alpha \lambda(x : A' : \Box^{q'}_{u'}), t'}$$

$$\frac{\Sigma;\Theta;\Gamma \vdash A \equiv_\alpha A' \qquad \Sigma;\Theta;\Gamma \vdash B \leqslant_\alpha B'}{\Sigma;\Theta;\Gamma \vdash \forall (x : A : \Box^{q}_{u}), (B : \Box^{q_1}_{u_1}) \leqslant_\alpha \forall (x : A' : \Box^{q'}_{u'}), (B' : \Box^{q'_1}_{u'_1})}$$

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

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

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

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

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

3 Rewrite Rules

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

3.1 Pattern

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

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

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

$$\frac{}{t \mid ?x\,\{[?x := t]\}} \qquad \frac{}{x \mid x\,\{[]\}} \qquad \frac{C \in \Sigma \text{ not a constant}}{C \mid C\,\{[]\}} \qquad \frac{q \text{ explicit quality}}{\square^q_u \mid \square^q_{?x_u}\,\{[?x_u := u]\}}$$

$$\frac{}{\square^q_u \mid \square^{?x_q}_{?x_u}\,\{[?x_q := q; ?x_u := u]\}} \qquad \frac{f \mid p\,\{\sigma_1\} \qquad t \mid a\,\{\sigma_2\}}{f\ t \mid p\ a\,\{\sigma_1 + \sigma_2\}}\ 7$$

$$\frac{A \mid a\,\{\sigma_1\} \qquad t \mid p\,\{\sigma_2\}}{\lambda(x : A : \square^{q'}_{u'}), t \mid \lambda(x : a : \square^q_u), p\,\{[q := q'; u := u']{+}\sigma_1 + \sigma_2\}}$$

$$\frac{A \mid a\,\{\sigma_1\} \qquad B \mid b\,\{\sigma_2\}}{\forall(x : A : \square^{q_1'}_{u_1'}), (t : \square^{q_2'}_{u_2'}) \mid \forall(x : a : \square^{q_1}_{u_1}), (b : \square^{q_2}_{u_2})\,\{[\vec{q_i} := \vec{q_i'}; \vec{u_i} := \vec{u_i'}]{+}\sigma_1 + \sigma_2\}}$$

$$\frac{c \mid p\,\{\sigma_c\} \qquad P \mid b\,\{\sigma_P\} \qquad b_i \mid b_i\,\{\sigma_i\}}{\text{case } c \text{ return } P : \square^{q'}_{u'} \text{ with } \vec{b_i} \mid \text{case } p \text{ return } a : \square^q_u \text{ with } \vec{b_i}\,\{[q := q'; u := u']{+}\sigma_c + \sigma_P + \sum \sigma_i\}}$$

Figure 4 Description of pattern matching

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

We also have to impose additional restrictions to prove confluence; they are listed and justified in Section 4.1. Finally, since we don't want to try all rewrite rules at all steps in reduction, we restrict patterns such that the head of their main branch of the pattern needs 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:

$$\frac{(p \rightsquigarrow r) \in \Sigma \qquad t \mid p\,\{\sigma\}}{\Sigma; \Theta; \Gamma \vdash t \to r\{\sigma\}}\ \textsc{Rew}$$

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

3.2 Pattern-Matching

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

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

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

4 A Criterion to Guarantee that Rewrite Rules are Type Preserving

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

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

4.1 Confluence and Postponement of α-cumulativity

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

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

$$\frac{(\Gamma \vdash ?x : A) \in \Theta}{\Sigma;\Theta;\Gamma \vdash_p ?x \triangleleft A}\ \textsc{PatVar} \qquad \frac{\Sigma;\Theta;\Gamma \vdash_p p \triangleright A \qquad \Sigma;\Theta;\Gamma \vdash_p A \leqslant_p B}{\Sigma;\Theta;\Gamma \vdash_p p \triangleleft B}\ \textsc{Conv}$$

$$\frac{(x : A) \in \Gamma}{\Sigma;\Theta;\Gamma \vdash_p x \triangleright A}\ \textsc{Var} \qquad \frac{(C : T) \in \Sigma}{\Sigma;\Theta;\Gamma \vdash_p C \triangleright T}\ \textsc{Ax-Symb} \qquad \frac{}{\Sigma;\Theta;\Gamma \vdash_p \square^q_u \triangleright \square^{\mathsf{Type}}_{\uparrow_q u}}\ \textsc{Sort}$$

$$\frac{\Sigma;\Theta;\Gamma \vdash_p p \triangleright \forall (x : A), B \qquad \Sigma;\Theta;\Gamma \vdash_p a \triangleleft A}{\Sigma;\Theta;\Gamma \vdash_p p\ a \triangleright B[x := a]}\ \textsc{App}$$

$$\frac{\Sigma;\Theta;\Gamma \vdash_p a \triangleleft \square^q_u \qquad \Sigma;\Theta;\Gamma, (x : a) \vdash_p p \triangleright B}{\Sigma;\Theta;\Gamma \vdash_p \lambda (x : a : \square^q_u), p \triangleright \forall (x : a), B}\ \textsc{Lambda}$$

$$\frac{\Sigma;\Theta;\Gamma \vdash_p a \triangleleft \square^q_u \qquad \Sigma;\Theta;\Gamma, (x : a) \vdash_p b \triangleleft \square^{q'}_{u'}}{\Sigma;\Theta;\Gamma \vdash_p \forall (x : a : \square^q_u), b : \square^{q'}_{u'} \triangleright \square^{q'}_{u *^q_{q'} u'}}\ \textsc{Forall}$$

$$\frac{\begin{array}{c}(\Gamma_p, \Gamma_i, [I : \forall \Gamma_p, \forall \Gamma_i, \square^{q_I}_{u_I}], \vec{\Gamma_k}, [C_k : \forall (\vec{p} : \Gamma_p), \forall \Gamma_k, I\,\vec{p}\,\vec{i_k}]_k) \in \Sigma \qquad \Sigma;\Theta;\Gamma \vdash_p p \triangleright I\ \vec{p}\ \vec{i} \\ \Sigma;\Theta;\Gamma, (\vec{i} : \Gamma_i), (x : I\ \vec{p}\ \vec{i}) \vdash_p a \triangleleft \square^q_u \qquad \Sigma;\Theta;\Gamma, \Gamma_k \vdash_p b_k \triangleleft a[\Gamma_i := \vec{i}, x := C_k\,\vec{p}\,\vec{i_k}]\end{array}}{\Sigma;\Theta;\Gamma \vdash_p \mathrm{case}\, p\, \mathrm{return}\, a : \square^q_u\, \mathrm{with}\, \vec{b_k} \triangleright a[\Gamma_i := \vec{i}, x := p]}\ \textsc{Case}$$

Figure 5 Type inference of patterns

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

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

4.2 Type Inference of Patterns

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

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

given in Figure 5.

Most of the rules follow the standard bidirectional discipline. For instance, the type of a variable (Var) is directly inferred from the local context, and the type of an application (App) is inferred by inferring the type for the function and checking the type of the argument pattern against the domain of the function.

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

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

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

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

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

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

4.3 Subject Reduction

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

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

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

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

$$\forall t, T, \sigma,\ \Sigma; []; \Gamma \vdash t : T \wedge t \,|\, p\{\sigma\} \implies \Sigma; []; \Gamma \vdash \sigma : \Theta \wedge \Sigma; []; \Gamma \vdash T_p\{\sigma\} \leqslant T.$$

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

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

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

$$\Sigma; \Theta; \Delta_p \vdash_p p \triangleright T_p \implies$$
$$\forall \Delta, t, T, \sigma_0, \sigma,\ \Sigma; []; \Gamma, \Delta \vdash t : T \wedge t \mid p\{\sigma\} \wedge \Delta_p\{\sigma_0\} \equiv \Delta \implies$$
$$\Sigma; []; \Gamma \vdash \sigma : \Theta\{\sigma_0\} \wedge T_p\{\sigma_0, \sigma\} \leqslant T$$

and the corresponding predicate on argument patterns:

$$\Sigma; \Theta; \Delta_p \vdash_p p \triangleleft T_p \implies$$
$$\forall \Delta, t, T, \sigma_0, \sigma,\ \Sigma; []; \Gamma, \Delta \vdash t : T \wedge t \mid p\{\sigma\} \wedge T_p\{\sigma_0, \sigma\} \equiv T \wedge \Delta_p\{\sigma_0\} \equiv \Delta \implies$$
$$\Sigma; []; \Gamma \vdash \sigma : \Theta\{\sigma_0\}$$

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

- Case $?x$:
  Hypotheses: $\Sigma; []; \Gamma, \Delta \vdash t : T$, $\sigma = [?x := t]$, $T_p\{\sigma_0\} \equiv T$, $\Delta_p\{\sigma_0\} \equiv \Delta$
  Goal: $\Sigma; []; \Gamma \vdash (\Delta_p \vdash ?x : T_p)\{\sigma_0, \sigma\}$ which simplifies to $\Sigma; []; \Gamma, \Delta_p\{\sigma_0, \sigma\} \vdash t : T_p\{\sigma_0, \sigma\}$
  Proof: Neither Δ_p nor T_p mention $?x$, so they are respectively convertible to Δ and T by hypothesis, which proves our goal.
- Case $p\ a$
  Hypotheses: $\Sigma; \Theta; \Delta \vdash_p p \triangleright \forall (x : A_p), B_p$, $\Sigma; \Theta; \Delta \vdash_p a \triangleleft A_p$, $T_p = B_p[x := a]$, $\Sigma; []; \Gamma, \Delta \vdash f\ t : T$, $f \mid p\{\sigma_1\}$, $t \mid a\{\sigma_2\}$, $\Delta_p\{\sigma_0\} \equiv \Delta$.
  By inversion of typing on an application, there exists A and B such that $\forall (x : A), B \leqslant T$, $\Sigma; []; \Gamma, \Delta \vdash f : \forall x : A, B$ and $\Sigma; []; \Gamma, \Delta \vdash t : A$.
  Then, by induction hypothesis on p, we get $(\forall x : A_p, B_p)\{\sigma_0, \sigma_1\} \leqslant \forall x : A, B$ and $\Sigma; []; \Gamma \vdash \sigma_1 : \Theta\{\sigma_0\}$.
  By the definition of substitution and injectivity of products for $\leqslant$, $A_p\{\sigma_0, \sigma_1\} \equiv A$ and $B_p\{\sigma_0, \sigma_1\} \leqslant B$. Since Δ_p does not contain any metavariable in $|\sigma_1|$, $\Delta_p\{\sigma_0, \sigma_1\} \equiv \Delta$.
  This means we can use the induction hypothesis on a and get $\Sigma; []; \Gamma \vdash \sigma_2 : \Theta\{\sigma_0, \sigma_1\}$
  Goal: $B_p[x := a]\{\sigma\} \leqslant B[x := t]$ and $\Sigma; []; \Gamma \vdash \sigma_1, \sigma_2 : \Theta\{\sigma_0\}$. The second goal is immediately proved by concatenating the induction hypotheses.
  Proof: by commuting the substitution and metasubstitution,

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

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

- Case $\forall(x : a : \square^{?q}_{?u}), b : \square^{?q'}_{?u'}$
  Hypotheses: $\Sigma; \Theta; \Delta \vdash_p a \triangleleft \square^{?q}_{?u}$, $\Sigma; \Theta; \Delta \vdash_p b \triangleleft \square^{?q'}_{?u'}$, $T_p = \square^{?q'}_{?u *^{?q}_{?q'} ?u'}$,
  $\Sigma; []; \Gamma, \Delta \vdash \forall(x : A : \square^q_u), (B : \square^{q'}_{u'}) : T,\ A^+ \mid a\{\sigma_1\},\ B^+ \mid b\{\sigma_2\},\ \Delta_p\{\sigma_0\} \equiv \Delta$ with $\sigma = [?q := q; ?q' := q'; ?u := u; ?u' := u'] + \sigma_1 + \sigma_2$.
  By inversion of typing on a product, we get $\square^{q'}_{u *^q_{q'} u'} \leqslant T$, $\Sigma; []; \Gamma, \Delta \vdash A : \square^q_u$ and $\Sigma; []; \Gamma, \Delta, (x : A) \vdash B : \square^{q'}_{u'}$.
  Since Δ_p does not contain any metavariable in $|\sigma_1|$, $(\Delta_p, x : a)\{\sigma_0, \sigma_1\} \equiv \Delta, (x : A)$. This means we can use both induction hypotheses to get $\Sigma; []; \Gamma \vdash \sigma_1 : \Theta\{\sigma_0\}$ and $\Sigma; []; \Gamma \vdash \sigma_2 : \Theta\{\sigma_0, \sigma_1\}$
  Goal: $\square^{?q'}_{?u *^{?q}_{?q'} ?u'}\{\sigma\} \leqslant \square^{q'}_{u *^q_{q'} u'}$ and $\Sigma; []; \Gamma \vdash \sigma_1, \sigma_2 : \Theta\{\sigma_0\}$. The second goal is immediately proved by concatenating the induction hypotheses.
  Proof: We didn't define yet how $*$ should behave when its quality arguments are variables. Evidently, they need to return the minimal universe, so they must behave as Prop to satisfy the required inequality.
- Case Conv:
  Hypotheses: $\Sigma; []; \Gamma, \Delta \vdash t : T$, $t \mid p\{\sigma\}$, $T'_p\{\sigma_0\} \equiv T$, $\Delta_p\{\sigma_0\} \equiv \Delta$, $T_p\{\sigma\} \leqslant T$, $T_p \leqslant_p T'_p$, $\Sigma; []; \Gamma \vdash \sigma : \Theta\{\sigma_0\}$.
  Goal: $\Sigma; []; \Gamma \vdash \sigma : \Theta\{\sigma_0\}$. It is one of our hypotheses, our cumulativity hypotheses are useless for now. In Section 5 however, we present the definition of pattern cumulativity to extract some information from these unused hypotheses. Let us simply note that, by symmetry of $\equiv$ and transitivity, $T_p\{\sigma\} \leqslant T'_p\{\sigma\}$, which we will use alongside $T_p \leqslant_p T_{p'}$.

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

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

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

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

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

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

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

$$\frac{\Sigma;\Theta;\Gamma \vdash t \to^* t' \qquad \Sigma;\Theta;\Gamma \vdash u \to^* u' \qquad \Sigma;\Theta;\Gamma \vdash_p t' \leqslant_p u' \triangleright \mathcal{E}}{\Sigma;\Theta;\Gamma \vdash_p t \leqslant_p u \triangleright \mathcal{E}}\ \textsc{Red}$$

$$\frac{C \text{ is injective}}{\Sigma;\Theta;\Gamma \vdash_p C \leqslant_p C \triangleright []}\ \textsc{Inj} \qquad \frac{x \in \Gamma}{\Sigma;\Theta;\Gamma \vdash_p x \leqslant_p x \triangleright []}\ \textsc{Rel}$$

$$\frac{\Sigma \vdash q =_q q' \wedge u \leqslant_u u' \text{ possible}}{\Sigma;\Theta;\Gamma \vdash_p \square_u^q \leqslant_p \square_{u'}^{q'} \triangleright [u \leqslant_u u'; q =_q q']}\ \textsc{Sort}$$

$$\frac{\Sigma;\Theta;\Gamma \vdash_p A \equiv_p B \triangleright \mathcal{E} \qquad \Sigma;\Theta;\Gamma,(x:A) \vdash_p t \leqslant_p u \triangleright \mathcal{E}' \qquad \mathcal{B} \in \{\lambda, \forall\}}{\Sigma;\Theta;\Gamma \vdash_p \mathcal{B}(x:A), t \leqslant_p \mathcal{B}(x:B), u \triangleright \mathcal{E} \cup \mathcal{E}'}\ \textsc{Lambda/Forall}$$

$$\frac{\Sigma;\Theta;\Gamma \vdash_p f \leqslant_p f' \triangleright \mathcal{E} \qquad \Sigma;\Theta;\Gamma \vdash_p a \equiv_p a' \triangleright \mathcal{E}' \qquad \Sigma;\Gamma \vdash f, f' \text{ whne}}{\Sigma;\Theta;\Gamma \vdash_p f\ a \leqslant_p f'\ a' \triangleright \mathcal{E} \cup \mathcal{E}'}\ \textsc{App}$$

$$\frac{\begin{array}{c}\Sigma;\Theta;\Gamma \vdash_p c \equiv_p c' \triangleright \mathcal{E}_c \\ \Sigma;\Theta;\Gamma,\Gamma_i \vdash_p P \equiv_p P' \triangleright \mathcal{E}_r \qquad \left[\Sigma;\Theta;\Gamma,\Gamma_k \vdash_p b_k \equiv_p b'_k \triangleright \mathcal{E}_{b,k}\right] \qquad \Sigma;\Gamma \vdash c, c' \text{ whne}\end{array}}{\Sigma;\Theta;\Gamma \vdash_p \mathrm{case}\, c \,\mathrm{return}\, P \,\mathrm{with}\, \vec{b} \leqslant_p \mathrm{case}\, c' \,\mathrm{return}\, P' \,\mathrm{with}\, \vec{b'} \triangleright\ \mathcal{E}_c \cup \mathcal{E}_r \cup \bigcup_i \mathcal{E}_{b,i}}\ \textsc{Case}$$

$$\frac{}{\Sigma;\Theta;\Gamma \vdash_p ?x \equiv_p t \triangleright [\Delta \vdash ?x := t]}\ \textsc{PatVar}$$

$$\frac{t \text{ or } u \text{ contains a pattern variable or a symbol in head position}}{\Sigma;\Theta;\Gamma \vdash_p t \leqslant_p u \triangleright []}\ \textsc{Failsafe}$$

$$\frac{x \in \Gamma}{\Sigma;\Gamma \vdash x \text{ whne}} \qquad \frac{\Sigma;\Gamma \vdash f \text{ whne}}{\Sigma;\Gamma \vdash f\ a \text{ whne}} \qquad \frac{\Sigma;\Gamma \vdash c \text{ whne}}{\Sigma;\Gamma \vdash \mathrm{case}\, c \,\mathrm{return}\, P \,\mathrm{with}\, \vec{b} \text{ whne}}$$

Figure 6 The rules for conversion judgments $\leqslant_p$ and $\equiv_p$ (replace all $\leqslant_p$ with $\equiv_p$ in the rules)

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

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

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

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

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

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

6 Implementation

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

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

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

```
#[unfold_fix] Symbol error : ℕ.
Rewrite Rule error_case :=
  match error return ℕ with 0 ⇒ _ | S _ ⇒ _ end ⇝ error.
Eval compute in error + 5. (* error *)
```

In practice, metavariables can be used to bind terms on the left-hand side and mention them in the right-hand side, but they are also convenient to avoid describing explicitly sub-patterns that are 'forced' and will be found by the extraction of equalities mechanism described in Section 5. To this end, we have added the `_` to produce metavariables that are 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

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

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.

4 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, pages 50–66, Berlin, Heidelberg, 1990. Springer. doi:10.1007/3-540-52335-9_47.

5 Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-79876-5_37.

6 Thiago Felicissimo. Generic bidirectional typing for dependent type theories, October 2023. arXiv:2307.08523.

7 Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without K. *Proceedings of the ACM on Programming Languages*, 3(POPL):3:1–3:28, January 2019. doi:10.1145/3290316.

8 Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination Des Coupures de l'arithmétique d'ordre Supérieur*. PhD thesis, Université Paris VII, 1972.

9 Antonius J. C. Hurkens. A simplification of Girard's paradox. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 266–278, Berlin, Heidelberg, 1995. Springer. doi:10.1007/BFb0014058.

10 Meven Lennon-Bertrand. Complete Bidirectional Typing for the Calculus of Inductive Constructions. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ITP.2021.24*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.24.

11 Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*, volume 32. Citeseer, 2007.

12 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. doi:10.1007/978-3-319-89884-1_9.

13 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, 2020. doi:10.1007/s10817-019-09540-0.

14 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. doi:10.1145/3371076.

15 The Coq Development Team. The Coq Proof Assistant. Zenodo, June 2023. doi:10.5281/zenodo.8161141.

16 Beta Ziliani and Matthieu Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 179–191, New York, NY, USA, August 2015. Association for Computing Machinery. doi:10.1145/2784731.2784751.