The Taming of the Rew

A Type Theory with Computational Assumptions

JESPER COCKX, TU Delft, Netherlands
NICOLAS TABAREAU, Inria, France
THÉO WINTERHALTER, Inria, France

Dependently typed programming languages and proof assistants such as Agda and Coq rely on *computation* to automatically simplify expressions during type checking. To overcome the lack of certain programming primitives or logical principles in those systems, it is common to appeal to axioms to postulate their existence. However, one can only postulate the bare existence of an axiom, not its computational behaviour. Instead, users are forced to postulate equality proofs and appeal to them explicitly to simplify expressions, making axioms dramatically more complicated to work with than built-in primitives. On the other hand, the *equality reflection rule* from extensional type theory solves these problems by collapsing computation and equality, at the cost of having no practical type checking algorithm.

This paper introduces *Rewriting Type Theory* (RTT), a type theory where it is possible to add computational assumptions in the form of *rewrite rules*. Rewrite rules go beyond the computational capabilities of intensional type theory, but in contrast to extensional type theory, they are applied automatically so type checking does not require input from the user. To ensure type soundness of RTT—as well as effective type checking—we provide a framework where confluence of user-defined rewrite rules can be checked modularly and automatically, and where adding new rewrite rules is guaranteed to preserve subject reduction. The properties of RTT have been formally verified using the MetaCoq framework and an implementation of rewrite rules is already available in the Agda proof assistant.

CCS Concepts: • **Theory of computation → Type theory**.

Additional Key Words and Phrases: type theory, dependent types, rewriting theory, confluence, termination

ACM Reference Format:
Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. 2021. The Taming of the Rew: A Type Theory with Computational Assumptions. *Proc. ACM Program. Lang.* 5, POPL, Article 60 (January 2021), 29 pages. https://doi.org/10.1145/3434341

1 INTRODUCTION

At the heart of every dependently typed programming language or proof assistant, such as Agda [Agda Development Team 2020] and Coq [Coq Development Team 2016], lies a core type theory, such as Martin-Löf type theory (MLTT) [Martin-Löf 1975] or the calculus of inductive constructions (CIC) [Paulin-Mohring 2015]. To prove theorems that are based on additional properties that are not provable in the considered type theory, the logic or programming principles of those type theories can be extended on the fly with axioms, which technically corresponds to introducing new variables in the context. The working mathematician or computer scientist is familiar with

Authors' addresses: Jesper Cockx, TU Delft, Delft, Netherlands; Nicolas Tabareau, Inria, Gallinette Project-Team, Nantes, France; Théo Winterhalter, Inria, Gallinette Project-Team, Nantes, France.

2475-1421/2021/1-ART60
https://doi.org/10.1145/3434341

the fact that such additional axioms may turn the underlying theory into an inconsistent one, but traditionally consider the gain in expressiveness to be worth it.

However, being able to assume only logical axioms has a severe limitation, because axioms are computational black boxes. For instance, to add a higher inductive type (HIT) [Univalent Foundations Program 2013] to a proof assistant it is not sufficient to postulate its existence and elimination principle; one also needs its elimination principle to compute on constructors. The standard workaround is to replace computation by propositional equality, but this dramatically complicates working with the eliminator as one now has to appeal to the computation law explicitly every time it is used. In addition, in homotopy type theory these postulated equalities require additional equalities to ensure coherence, leading to the situation known as 'coherence hell'.

Thus one may wonder why it is not possible to extend the computational content of MLTT or CIC by postulating new computational rules, just as it is possible to postulate new axioms. The reason is that adding arbitrary computational rules in the theory has a much stronger impact than adding axioms. It may break not only consistency or canonicity, but also decidability of type checking and even subject reduction, thus leading to an unpredictable theory that can no longer be considered as a typed programming language. As we discuss in Sect. 3, computation rules may even break subject reduction without breaking consistency.

This paper proposes RTT, the first dependent type theory that can be extended with rewrite rules, with a *modular* and *decidable* syntactic check that guarantees subject reduction and partial decidability of type checking—and this without relying on weak or strong normalization of the resulting rewrite system. This is achieved by having a clear syntactical guideline of which rewrite rules can be added to the system, together with a confluence check based on the *triangle property* [Terese 2003] which is stronger than mere confluence and does not require termination.

An implementation of RTT with the check of the triangle property is already available in the Agda proof assistant. Furthermore, the correctness of our approach has been formalized[1] in the Coq proof assistant, using the MetaCoq project [Sozeau et al. 2020], which is a necessary step to get a trusted extension of a proof assistant. In all the lemmas that have been formalized, we refer to the Coq formalization using *[file.v/lemma]* to indicate the file in which the lemma has been proven and the exact name of the lemma. Actually, the setting of the formalization is more general than the theory exposed in the paper as it extends the whole theory of Coq, but we prefer to keep the presentation of the theory as simple as possible to focus on the essentials of the framework.

Let us be explicit upfront that we do not claim any novelty on the generality of the criterion that we use, as it corresponds to an extension of the standard Tait-Martin Löf criterion on parallel reduction [Barendregt 1984; Takahashi 1995] to rewrite rules. Neither do we pretend to be more general than results on higher-order rewriting as found for example in the work by van Oostrom [1994a]. The novelty of our work lies rather in the definition of a decidable and modular criterion in a dependently typed setting (where reduction and typing are intricate), thus leading to a type theory that can be implemented inside a proof assistant. Besides, our framework does not rely on termination, which allows us to treat termination just like consistency: a property that may not be valid anymore in the presence of postulated rewrite rules, but is not necessary to get a usable proof assistant, except to guarantee totality of the type checking algorithm.

Outline of the paper. We start by demonstrating in Sect. 2 the usefulness of rewrite rules by means of various examples. In Sect. 3 we analyse various reasons for rewrite rules to break subject reduction, most notably by breaking confluence of reduction. Then we explain in Sect. 4 a modular and checkable criterion for guaranteeing confluence in a type theory with computational assumptions. The formal part of the paper begins in Sect. 5 with the presentation of RTT, an extension of the

[1] https://github.com/TheoWinterhalter/template-coq/tree/rewrite-rules

Predicative Calculus of Cumulative Inductive Constructions (PCUIC) with user-declared rewrite rules. In Sect. 6 we then present the main metatheoretical properties of RTT, assuming that the triangle criterion holds: subject reduction, consistency, and (semi-)decidability of type checking. We further describe an algorithm to check that the triangle criterion indeed holds for a given rewrite system, as well as our implementation of rewrite rules for the Agda system, in Sect. 7. Finally, we discuss related work in Sect. 8, future work in Sect. 9, and conclude the paper in Sect. 10.

2 REWRITE RULES FOR TYPE THEORY IN ACTION

Before presenting the theoretical contribution of this paper—a type theory with computational assumptions—we exemplify how it can be used in practice to emulate parallel reduction, define general fixpoints, postulate new kinds of inductive types, or extend the language with exceptions. These examples demonstrate the two main use cases of rewrite rules: improving the computational behaviour of definitions by turning propositional equalities into computation rules (Sect. 2.1 and Sect. 2.2) and extending the theory with new features that would otherwise be impossible to define (Sect. 2.3, Sect. 2.4, and Sect. 2.5).

2.1 Emulating Parallel Reduction

There are several ways of defining the addition of natural numbers in Agda or Coq, but none of them satisfy together the laws

$$\begin{array}{lcl} \mathsf{plus}\ (\mathsf{suc}\ m)\ n & \rightsquigarrow & \mathsf{suc}\ (\mathsf{plus}\ m\ n) \\ \mathsf{plus}\ m\ (\mathsf{suc}\ n) & \rightsquigarrow & \mathsf{suc}\ (\mathsf{plus}\ m\ n) \end{array}$$

Such a function can be postulated using the following block of rewrite rules:

$$\begin{array}{l} \mathsf{postulate\ rewrite\ block}\ \mathsf{Plus}\mathbb{N}\ \mathsf{where} \\ \quad \mathsf{axiom}\ \left[\mathsf{plus}\ :\ \mathbb{N} \to \mathbb{N} \to \mathbb{N}\right. \\ \quad \mathsf{rewrite} \left[\begin{array}{lcl} \mathsf{plus\ zero}\ ?n & \rightarrow & n \\ \mathsf{plus}\ ?n\ \mathsf{zero} & \rightarrow & n \\ \mathsf{plus}\ (\mathsf{suc}\ ?m)\ ?n & \rightarrow & \mathsf{suc}\ (\mathsf{plus}\ m\ n) \\ \mathsf{plus}\ ?m\ (\mathsf{suc}\ ?n) & \rightarrow & \mathsf{suc}\ (\mathsf{plus}\ m\ n) \end{array}\right. \end{array}$$

The name $\mathsf{Plus}\mathbb{N}$ of the rewrite block is just there for convenience but cannot be referenced directly. The first square bracket in the syntax of a rewrite block gathers all the axioms of the block, while the second one gathers the rewrite rules.

In this situation, it is perhaps not immediately obvious that the rewrite system is confluent, as there is overlap between the left-hand sides of the rewrite rules. However, this rewrite system in fact satisfies the property of *strong confluence*: for any term of the form $\mathsf{plus}\ u\ v$ that can be rewritten to two different terms w_1 and w_2 in a single step, we can always find another term w such that both w_1 and w_2 can be rewritten to w, again in a single step. It is a standard result from rewriting theory that this is enough to conclude (global) confluence of the system.

2.2 Fixpoints: Beyond the Guard Condition

In Coq or Agda, the definition of fixpoints is guarded by a syntactic condition which ensures that the function is terminating. However, this condition, which roughly amounts to checking that recursive calls are done on syntactic subterms, is often too restrictive. For instance, consider the naive definition of a divide predicate:

$$\begin{array}{lcl} \mathsf{divide} : \mathbb{N} \to \mathbb{N} \to \square & & \\ \mathsf{divide}\ m\ \mathsf{zero} & = & \top \\ \mathsf{divide}\ \mathsf{zero}\ (\mathsf{suc}\ n) & = & \bot \\ \mathsf{divide}\ (\mathsf{suc}\ m)\ (\mathsf{suc}\ n) & = & (m \le n)\ \&\ (\mathsf{divide}\ (\mathsf{suc}\ m)\ (n - m)) \end{array}$$

This definition is rejected because the recursive call divide (suc m) (n - m) is not done on a subterm of suc m and suc n. However, it is possible to define the function divide using an accessibility predicate. This justifies the definition propositionally. Hence it is justified to add the definition of the function using rewrite rules, without any risk with respect to termination:

$$\begin{array}{l} \mathsf{postulate\ rewrite\ block\ Divide\ where} \\ \quad \mathsf{axiom} \ \left[\mathsf{divide} : \mathbb{N} \to \mathbb{N} \to \square \right. \\ \quad \mathsf{rewrite} \left[\begin{array}{lcl} \mathsf{divide}\ m\ \mathsf{zero} & \rightarrowtriangle & \top \\ \mathsf{divide}\ \mathsf{zero}\ (\mathsf{suc}\ n) & \rightarrowtriangle & \bot \\ \mathsf{divide}\ (\mathsf{suc}\ m)\ (\mathsf{suc}\ n) & \rightarrowtriangle & (m \le n)\ \&\ (\mathsf{divide}\ (\mathsf{suc}\ m)\ (n - m)) \end{array} \right. \end{array}$$

This is one instance of a general principle: if we can define a certain function and prove an equation about it *propositionally* then it is fine to add that as a rewrite rule. Other examples of this principle can be found in the work of Allais et al. [2013].

2.3 Non-Strictly Positive Inductive Types

As explained in the introduction, using rewrite rules, it becomes possible to postulate the existence of inductive types that do not exist in the original theory. Although postulating the existence of negative inductive types, *e.g.*, the untyped lambda calculus, directly leads to an inconsistent theory, the situation is less clear for positive inductive types not respecting the strict positivity condition. It is known since the work of Coquand and Paulin [1988] that positive inductive types in presence of an impredicativite universe Prop may lead to inconsistency, as it is the case for the inductive type

$$\begin{array}{l} \mathsf{data\ P} : \square\ \mathsf{where} \\ \quad \mathsf{introP}\ :\ ((\mathsf{P} \to \mathsf{Prop}) \to \mathsf{Prop}) \to \mathsf{P} \end{array}$$

But if the inductive definition does not involve an impredicative universe, it is believed that adding the positive inductive type to the type theory is safe, as it is already the case for system F. For instance, consider the following example originally due to Hofmann [1993] that can be used to solve the breadth-first traversal problem, with a constructor that makes use of something that looks like a continuation, or a coroutine:

$$\begin{array}{l} \mathsf{postulate\ rewrite\ block\ Coroutine\ where} \\ \quad \mathsf{axiom} \left[\begin{array}{lcl} \mathsf{Cor} & : & \square \\ \mathsf{Over} & : & \mathsf{Cor} \\ \mathsf{Next} & : & ((\mathsf{Cor} \to \mathsf{list}\ \mathbb{N}) \to \mathsf{list}\ \mathbb{N}) \to \mathsf{Cor} \\ \mathsf{elim}_{\mathsf{Cor}} & : & (P : \mathsf{Cor} \to \square) \to (P_{\mathsf{Over}} : P\ \mathsf{Over}) \to \\ & & (P_{\mathsf{Next}} : \forall f, P\ (\mathsf{Next}\ f)) \to (c : \mathsf{Cor}) \to P\ c \end{array} \right. \\ \quad \mathsf{rewrite} \left[\begin{array}{lcl} \mathsf{elim}_{\mathsf{Cor}}\ ?P\ ?P_{\mathsf{Over}}\ ?P_{\mathsf{Next}}\ (\mathsf{Over}) & \rightarrowtriangle & P_{\mathsf{Over}} \\ \mathsf{elim}_{\mathsf{Cor}}\ ?P\ ?P_{\mathsf{Over}}\ ?P_{\mathsf{Next}}\ (\mathsf{Next}\ f) & \rightarrowtriangle & P_{\mathsf{Next}}\ f \end{array} \right. \end{array}$$

This inductive type is not strictly positive because Cor occurs (positively) on the left of an arrow. Although there is no general argument for termination of the elimination principle, Berger et al. [2019] have shown that the breadth-first traversal algorithm defined by Hofmann using this non-strictly positive inductive type actually terminates.

2.4 Higher Inductive Types

In the same way, it is possible to postulate the existence of higher inductive types (HITs) (or also higher inductive-inductive types and higher inductive-recursive types) in the theory. For instance, it is possible to define the HIT S^1 of the circle with the following record:

$$\begin{array}{ll}
\multicolumn{2}{l}{\textsf{postulate rewrite block Circle where}} \\
\textsf{axiom} & \left[\begin{array}{lll}
S^1 & : & \square \\
\textsf{base} & : & S^1 \\
\textsf{loop} & : & \textsf{base} = \textsf{base} \\
\textsf{elim}S^1 & : & (P : S^1 \to \square)(b : P\ \textsf{base})(l : (\textsf{loop} \mathbin{\#} b) = b)(x : S^1) \to P\ x \\
\textsf{elim}S^1{}_{\textsf{loop}} & : & (P : S^1 \to \square)(b : P\ \textsf{base})(l : (\textsf{loop} \mathbin{\#} b) = b) \\
 & & \to \textsf{apD}\ (\textsf{elim}S^1\ P\ b\ l)\ \textsf{loop} = \textsf{loop}
\end{array}\right. \\
\textsf{rewrite} & \left[\ \textsf{elim}S^1\ ?P\ ?b\ ?l\ \textsf{base} \ \rightharpoonup\ b\right.
\end{array}$$

In the definition above, $e \mathbin{\#} t$ denotes the transport of t by an equality proof e and

$$\textsf{apD} : \{A : \square\}\{B : A \to \square\}(f : \forall x, Bx)\{x\ y : A\}(p : x = y) \to p \mathbin{\#} (f\ x) = f\ y$$

is the dependent version of the functoriality of functions.

It is not possible to define a version of S^1 where the propositional equality $\textsf{elim}S^1{}_{\textsf{loop}}$ is definitional because apD is a function and not a rigid pattern.[2]

2.5 Exceptional Type Theory

Exceptional Type Theory [Pédrot and Tabareau 2018] is an extension of MLTT with call-by-name exceptions. This extension is justified through a syntactic translation into MLTT itself. However, to be useful in practice, it is better to have a direct presentation of the source theory, as described by Pédrot et al. [2019] with a raise operator. In RTT, raise can be postulated together with its behaviour on inductive types: matching on an exception raises an exception in the return predicate.

$$\begin{array}{ll}
\multicolumn{2}{l}{\textsf{postulate rewrite block Exception where}} \\
\textsf{axiom} & \left[\begin{array}{lll}
\mathbb{E} & : & \square \\
\textsf{raise} & : & (A : \square) \to \mathbb{E} \to A
\end{array}\right. \\
\textsf{rewrite} & \left[\begin{array}{lll}
\textsf{elim}_{\mathbb{N}}\ ?P\ ?P_0\ ?P_S\ (\textsf{raise}\ \mathbb{N}\ ?e) & \rightharpoonup & \textsf{raise}\ (P\ (\textsf{raise}\ \mathbb{N}\ e))\ e \\
\textsf{elim}_{\mathbb{B}}\ ?P\ ?P_{\texttt{true}}\ ?P_{\texttt{false}}\ (\textsf{raise}\ \mathbb{B}\ ?e) & \rightharpoonup & \textsf{raise}\ (P\ (\textsf{raise}\ \mathbb{B}\ e))\ e
\end{array}\right.
\end{array}$$

Here we define a particular behaviour of the already existing elimination principle when the matched symbol is raise, which is to our knowledge a novel possibility of our system.

However, to recover the weak canonicity result proven by Pédrot and Tabareau [2018], we would need to add a rewrite rule to avoid a stuck exception in a dependent product.

$$\textsf{raise}\ ((x : ?A) \to ?B)\ ?e \rightharpoonup \lambda\,(x : A).\ \textsf{raise}\ B\ e.$$

Unfortunately, such a rule is not yet covered by our framework as it involves a matching on the binder $(x : ?A) \to ?B$ which requires some limited form of higher-order matching. For the moment, we need to postulate this as a propositional equality instead.[3]

[2]This is possible in the current Agda implementation, by adding a rewrite rule to apD [Cockx 2020]. However, in our framework this would require to define apD in the same block as $\textsf{elim}S^1$, which is usually not possible.

[3]In the Agda implementation, this rule can be declared and is accepted by the confluence checker. However, it falls beyond the type theory covered in this paper.

3 THE NEED FOR SYNTACTIC RESTRICTIONS AND CONFLUENCE

To motivate the need for syntactic restrictions on the kind of rewrite rules that can be written, as well as the need for confluence of the resulting rewriting system, let us now review some seemingly harmless well-typed rewrite rules which break subject reduction and thus the very meaning of dependently typed programs.

Note that the examples in this section are not necessarily meant to be realistic, but rather to showcase the different corner cases that need to be excluded in order to build a system that is usable in practice.

Breaking subject reduction by rewriting type constructors. Equality between endofunctions on natural numbers and functions from natural numbers to booleans

$$(\mathbb{N} \to \mathbb{N}) = (\mathbb{N} \to \mathbb{B})$$

is admissible in type theory. Suppose now that one wants to add it as a definitional rewrite rule

$$(\mathbb{N} \to \mathbb{N}) \twoheadrightarrow (\mathbb{N} \to \mathbb{B})$$

in the theory. Then, the term $(\lambda\,(x : \mathbb{N}).\ x)\ 0$ has type $\mathbb{B}$ because $\lambda\,(x : \mathbb{N}).\ x$ has type $\mathbb{N} \to \mathbb{B}$. Indeed $\lambda\,(x : \mathbb{N}).\ x$ has type $\mathbb{N} \to \mathbb{N}$ which is convertible to $\mathbb{N} \to \mathbb{B}$. However, $(\lambda\,(x : \mathbb{N}).\ x)\ 0$ reduces to 0 after β-reduction, which does not have type $\mathbb{B}$ because $\mathbb{N}$ is not convertible to $\mathbb{B}$. Subject reduction is not valid anymore. The problem is not specific to the considered equality, nor to the fact that it is just admissible but not provable. Indeed, consider the following derivable equality on vectors exploiting the commutativity of addition

$$(\mathsf{Vec}\ A\ (n+m) \to \mathsf{Vec}\ A\ (n+m)) = (\mathsf{Vec}\ A\ (n+m) \to \mathsf{Vec}\ A\ (m+n))$$

where $A : \square$ and $n, m : \mathbb{N}$. Turning this equality into a rewrite rule would lead to the exact same issue: if $v : \mathsf{Vec}\ A\ (n+m)$ then $(\lambda\,(x : \mathsf{Vec}\ A\ (n+m)).\ x)\ v$ has type $\mathsf{Vec}\ A\ (m+n)$, and reduces to $v : \mathsf{Vec}\ A\ (n+m)$, but these two types are not convertible.

These two examples show that rewrite rules may break injectivity of Π types (also known as *product compatibility*), which is a crucial lemma in most proofs of subject reduction.

"*Turning a consistent (even derivable) equality into a rewrite rule may break subject reduction.*"

Breaking subject reduction by rewriting already defined symbols. Let $\mathsf{Box}\ A$ be a record type with a single field:[4]

```
record Box (A : □) : □ where
  constructor box
  field unbox  :  A
```

Thus, $\mathsf{unbox} : \mathsf{Box}\ A \to A$ satisfies the reduction $\mathsf{unbox}\ (\mathsf{box}\ x) \rightsquigarrow x$. Now, if we add a rewrite rule $\mathsf{Box}\ \mathbb{N} \twoheadrightarrow \mathsf{Box}\ \mathbb{B}$, then we have $\mathsf{unbox}\ (\mathsf{box}\ 0) : \mathbb{B}$ but this term again reduces to 0, which does not have type $\mathbb{B}$.

This example exploits the fact that $\mathsf{unbox}_A\ (\mathsf{box}_B\ x) \rightsquigarrow x$ even when $A \neq B$. Here Agda implicitly assumes that Box is injective w.r.t. definitional equality, enforcing the necessary conversion by typing. However, this assumption no longer holds when one adds the rewrite rule $\mathsf{Box}\ \mathbb{N} \twoheadrightarrow \mathsf{Box}\ \mathbb{B}$. If Agda were to check that the types A and B match in the reduction rule for unbox (as we would do if it is declared as a rewrite rule), evaluation would be stuck but subject reduction would be preserved.

"*Rewrite rules on existing symbols may break basic assumptions of type theory.*"

[4]In the following, we use Agda syntax to present our examples although what we say is not specific to Agda and is valid in any proof assistant based on MLTT or CIC.

Breaking subject reduction by breaking confluence. One direct solution to the problems described above is to forbid rewriting of type constructors such as $\longrightarrow$ and Box. However, it is easy to simulate the same conversion by postulating an auxiliary type $\mathsf{A} : \square$ with rewrite rules

$$\mathsf{A} \rightsquigarrow (\mathbb{N} \longrightarrow \mathbb{N}) \quad \text{and} \quad \mathsf{A} \rightsquigarrow (\mathbb{N} \longrightarrow \mathbb{B}).$$

Then, $\mathbb{N} \longrightarrow \mathbb{N}$ becomes convertible to $\mathbb{N} \longrightarrow \mathbb{B}$ and the term $(\lambda\ (x : \mathbb{N}).\ x)\ 0$ still has type $\mathbb{B}$. Here, the symbol that is rewritten is fresh and there seems to be no natural syntactic restriction that would forbid it. However, the resulting rewriting system is obviously not confluent, and this is the reason subject reduction is broken. Somehow, by defining those two rewrite rules, we have simulated a new conversion rule between $\mathbb{N} \longrightarrow \mathbb{N}$ and $\mathbb{N} \longrightarrow \mathbb{B}$. We conclude that a purely syntactic restriction on rewrite rules can never be enough to ensure subject reduction if one does not also have confluence.

"*Confluence is needed for subject reduction.*"

From these examples, we deduce the following requirements:

- Rewrite rules should only rewrite applications of 'fresh' (*i.e.,* postulated) symbols, not pre-existing symbols like $\longrightarrow$ or Box.
- The rewriting system defined by the rewrite rules together with the standard reduction rules of type theory should be confluent.

From these natural restrictions, we derive injectivity of type constructors and hence re-establish subject reduction.

By-block rewrite rules. We want to accept a rewrite rule only if it is defined on a fresh symbol, but at the same time we also want to define several rewrite rules on the same set of symbols. To resolve this tension, we allow rewrite rules to be defined by blocks. This allows us to check confluence of all rewrite rules defined in the block. Rewrite rules in the rewrite block can thus only rewrite symbols that were declared in the same block. For instance, we can define the inductive type of natural numbers as

$$\begin{array}{l}
\mathsf{postulate\ rewrite\ block\ natural\_numbers\ where} \\
\quad \mathsf{axiom} \left[\begin{array}{lcl}
\mathsf{N} & : & \square \\
\mathsf{zero} & : & \mathsf{N} \\
\mathsf{suc} & : & \mathsf{N} \longrightarrow \mathsf{N} \\
\mathsf{elim} & : & (P : \mathsf{N} \longrightarrow \square)(P_0 : P\ \mathsf{zero})(P_S : \forall n, P\ n \longrightarrow P\ (\mathsf{suc}\ n))(n : \mathsf{N}) \longrightarrow P\ n
\end{array}\right. \\
\quad \mathsf{rewrite} \left[\begin{array}{lcl}
\mathsf{elim}\ ?P\ ?P_0\ ?P_S\ \mathsf{zero} & \rightsquigarrow & P_0 \\
\mathsf{elim}\ ?P\ ?P_0\ ?P_S\ (\mathsf{suc}\ ?n) & \rightsquigarrow & P_S\ (\mathsf{elim}\ P\ P_0\ P_S\ n)
\end{array}\right.
\end{array}$$

About termination. In the examples we have considered so far, we have not taken termination into account. This is because we treat termination just like consistency: a property that may no longer hold in the presence of postulates. Indeed, the absence of confluence or subject reduction leads to a notion of conversion that is not (efficiently) implementable. One may still implement a partial conversion checker, but this would miss certain cases where the terms are actually convertible, which would lead the typechecker to behave in unpredictable ways in practice. In contrast, removing the requirement that rewrite rules are terminating only means the algorithm for checking conversion may no longer be total. However, whenever it does return a result, it is always correct, as we show in Sect. 6.5.

"*Termination of reduction is not a critical property of* MLTT *or* CIC."

The main reason why Agda and Coq enforce termination through the guard condition is because defining new functions as fixed points leads not only to non-termination of reduction but also inconsistency of the theory. For instance, let us consider the minimal example of a non-terminating function that allows us to inhabit $\bot$ with a trivial loop:

$$\begin{array}{lcl} \mathsf{loop} & : & \bot \\ \mathsf{loop} & = & \mathsf{loop} \end{array}$$

When checking this, Agda complains about the

problematic calls : loop.

And indeed, accepting loop as a valid fixpoint would create a trivial proof of $\bot$.

In contrast, an inconsistent rewrite rule remains in the context as a witness for inconsistency, in the same way as the axiom absurd : $\bot$ remains in the context. For example, translating the previous non-terminating fixpoint loop to the setting of rewrite rules, we get the following rewrite block:

```
postulate rewrite block Loop where
  axiom   [loop : ⊥
  rewrite [loop ⇀ loop
```

This block is *obviously and explicitly* inconsistent, even when the rewrite rule is omitted. The lesson from this is that the issue is not non-termination per se, but rather that non-terminating functions may introduce new canonical values in the theory, which should instead appear as additional assumptions in the context.

On the other hand, adding a non-terminating but consistent rewrite rule only breaks termination but not consistency. In Sect. 6.4 we show that if every postulated rewrite block has an *instantiation*–consisting of a witness for each axiom and a propositional equality proof for each rewrite rule–then RTT with those rewrite blocks is consistent. This is the case *even for non-terminating rewrite rules*.

In conclusion, from the lens of RTT we see the termination check as a way to simultaneously introduce a fresh axiom with some rewrite rules and at the same time provide an instantiation for it by proving it is total, thus avoiding the need to introduce new assumptions in the context.

4 THE TRIANGLE PROPERTY: CHECKING CONFLUENCE MODULARLY

The previous section shows that confluence of the added postulated rewrite rules is necessary to guarantee both type preservation and practicality of type checking. In this work, we seek a modular and decidable check for confluence, that does not rely on termination. This restricts the kind of examples that can be dealt with in the system, but in return it gives us a theory that is implementable and usable in practice.

4.1 Enforcing Confluence through the Triangle Property

It is well known that in absence of termination, local confluence is not enough to ensure global confluence. In his study of untyped higher-order rewrite rules, van Oostrom [1994a] has introduced the notion of decreasing diagrams, which has recently been used in the setting of the Dedukti proof assistant [Ferey and Jouannaud 2019]. However, it is not clear how to design an automatic tool to detect confluence of a set of rewrite rules using such a technique.

In this paper, we instead rely on a well-known and simpler technique originally introduced by Tait and Martin-Löf [Barendregt 1984], which shows that the notion of *parallel reduction* in untyped λ-calculus satisfies a much stronger property than local confluence, namely the triangle property.

Parallel reduction. Parallel reduction is a variant of one-step reduction that allows to reduce all existing redexes at the same time *in parallel*. It induces a relation that is larger than one-step reduction, but smaller than its reflexive transitive closure.

$$\rightsquigarrow \;\subseteq\; \Rrightarrow \;\subseteq\; \rightsquigarrow^* \tag{1}$$

$$\begin{tikzcd} t \arrow[d, Rrightarrow] \arrow[dr, Rrightarrow] & \\ u \arrow[r, Rrightarrow] & \rho(t) \end{tikzcd}$$

(a) The triangle property.

$$\begin{tikzcd} & t \arrow[dl, Rrightarrow] \arrow[d, Rrightarrow] \arrow[dr, Rrightarrow] & \\ u \arrow[dr, Rrightarrow] & & u' \arrow[dl, Rrightarrow] \\ & \rho(t) & \end{tikzcd}$$

(b) The triangle property implies global confluence.

Fig. 1. A visual representation of Definition 4.1 (left) and Lemma 4.2 (right).

Therefore, confluence of parallel reduction is equivalent to confluence of one-step reduction.

The triangle property in MLTT *and* CIC. Crucially for the proof of confluence, parallel reduction for untyped lambda calculus satisfies the property that there is an *optimal* reduced term $\rho(t)$ for every term t (Smolka [2015] provides a detailed exposition of this). This term is defined as a fixpoint on the syntax, performing as many reduction as possible during its traversal. The fact that it is optimal is expressed by the following triangle property (see also Fig. 1a).

Definition 4.1 (Triangle property). Parallel reduction of a rewriting system satisfies the *triangle property* when for every term t, we have $t \Rrightarrow \rho(t)$, and for every u such that $t \Rrightarrow u$, we also have $u \Rrightarrow \rho(t)$.

The main interest of parallel reduction lies in the fact that its confluence implies confluence of the rewriting system. For a terminating rewriting system, it also automatically provides a strategy to compute normal forms by iterating the function ρ.

Lemma 4.2 (Confluence of Parallel Reduction). [PCUICParallelReductionConfluence.v/pred1_diamond] *When parallel reduction satisfies the triangle property, then the rewrite system is globally confluent.*

Proof. Parallel reduction is confluent in exactly one step, by using the triangle property twice (Fig. 1b). Confluence of the rewrite system for one-step reduction then follows from Equation 1. □

The triangle property holds for $\lambda\Pi$, the restriction of Martin-Löf type theory to its functional part. But it also works for its extension to inductive types, pattern-matching, fixpoints and cofixpoints. This has for instance been formally proven for CIC in the MetaCoq project [Sozeau et al. 2019].

The triangle property in RTT. The approach for confluence we take for RTT is to admit a new set of rewrite rules only when it satisfies the triangle property and is orthogonal to all previously declared rewrite rules. This way, the resulting rewriting system can be shown to be always confluent.

The triangle property of a given set of rewrite rules can be automatically checked by choosing a specific function ρ (for example by maximally applying rewrite rules in a given order) and then analysing all critical pairs w.r.t. parallel reduction. In Sect. 7 we describe a general algorithm; here we restrict ourselves to an example.

Example 4.3. Consider a rewrite system with symbols 0, suc, and +, and rules $0 + n \rightarrowtriangle n$, $m + 0 \rightarrowtriangle m$, $(\mathsf{suc}\ m) + n \rightarrowtriangle \mathsf{suc}\ (m + n)$, and $m + (\mathsf{suc}\ n) \rightarrowtriangle \mathsf{suc}\ (m + n)$. We define $\rho(u)$ as follows:

- If u matches one of the rewrite rules, we apply the first rule that matches and apply ρ recursively to each subterm in a variable position (i.e. the positions where variables occur on the left-hand side of the rewrite rule).
- If u does not match any of the rewrite rules, we apply ρ to each subterm recursively.

However, with this definition of ρ, the triangle property is not satisfied: we have $\rho((\mathsf{suc}\, m) + (\mathsf{suc}\, n)) = \mathsf{suc}\,(m+(\mathsf{suc}\, n))$, but we also have $(\mathsf{suc}\, m)+(\mathsf{suc}\, n) \Rrightarrow \mathsf{suc}\,((\mathsf{suc}\, m)+n) \not\Rrightarrow \mathsf{suc}\,(m+(\mathsf{suc}\, n))$. To satisfy the triangle property, we extend the rewrite system with an additional 'parallel rewrite rule' $(\mathsf{suc}\, m) + (\mathsf{suc}\, n) \rightarrowtriangle \mathsf{suc}\,(\mathsf{suc}\,(m + n))$.

4.2 Modularity of Confluence

To avoid having to re-check previously declared rewrite rules when adding new rewrite rules, confluence needs to be *modular*: confluence of each individual block of rewrite rules should imply confluence of the combined system. Following the work of Van Oostrom [van Oostrom 1994b, Theorem 3.1.54], this leads us to restrict the rewrite rules to be *left-linear* and rewrite blocks to be two-by-two *orthogonal*. Linearity (or more precisely left-linearity) of a rewrite rule means that each pattern variable occurs at most once in the left-hand side of the rewrite rule. Meanwhile, orthogonality of rewrite blocks means that a term cannot be an instance of the left-hand sides of two rewrite rules from two different blocks.

Orthogonality is easily obtained by asking that each block of rewrite rules only rewrites patterns introduced in the same block. To see why non-linearity breaks modularity of confluence, consider the classic example [Appel et al. 2010, Counterexample 2.2]:

$$\begin{array}{l}\textsf{postulate rewrite block NonLinearRule where}\\ \quad \textsf{axiom}\ \left[f \ :\ \mathbb{N} \to \mathbb{N} \to \mathbb{N} \right.\\ \quad \textsf{rewrite}\ \left[\begin{array}{lll} f\ ?X\ ?X & \rightarrowtriangle & \textsf{true} \\ f\ ?X\ (S\ ?X) & \rightarrowtriangle & \textsf{false} \end{array}\right.\end{array}$$

Adding this system to CIC is confluent, because there is no critical pair. However, if one additionally postulates the existence of an infinite integer

$$\begin{array}{l}\textsf{postulate rewrite block Omega where}\\ \quad \textsf{axiom}\ \left[\omega \ :\ \mathbb{N} \right.\\ \quad \textsf{rewrite}\ \left[\omega \ \rightarrowtriangle\ S\ \omega \right.\end{array}$$

then we have $f\ \omega\ (S\ \omega) \rightsquigarrow \textsf{false}$ but also $f\ \omega\ (S\ \omega) \rightsquigarrow f\ (S\ \omega)\ (S\ \omega) \rightsquigarrow \textsf{true}$ and hence the system is non-confluent, even though the two systems are orthogonal.

Although full non-linearity cannot be accepted, we can still allow a weaker form of non-linearity. Indeed, consider the definition of the Martin-Löf identity type, with elimination principle $\textsf{elim}_{\textsf{Id}}$:

$$\begin{array}{l}\textsf{postulate rewrite block IdentityType where}\\ \quad \textsf{axiom}\ \left[\begin{array}{ll} \textsf{Id} & :\ (A:\square) \to A \to A \to \square \\ \textsf{refl} & :\ (A:\square) \to (x:A) \to \textsf{Id}\ A\ x\ x \\ \textsf{elim}_{\textsf{Id}} & :\ (A:\square) \to (x:A) \to (P:(y:A) \to (\textsf{Id}\ A\ x\ y) \to \square) \to \\ & \quad (t:P\ x\ (\textsf{refl}\ A\ x)) \to (y:A) \to (e:\textsf{Id}\ A\ x\ y) \to P\ y\ e \end{array}\right.\\ \quad \textsf{rewrite}\ \left[\textsf{elim}_{\textsf{Id}}\ ?A\ ?x\ ?P\ ?P_{refl}\ ?y\ (\textsf{refl}\ ?A'\ ?x')\ \rightarrowtriangle\ P_{refl} \right.\end{array}$$

Here, the left hand side is well-typed only when $?A$ is convertible to $?A'$ and when $?x$, $?y$ and $?x'$ are all convertible. So the rewrite rule is equivalent to the non-linear rule

$$\textsf{elim}_{\textsf{Id}}\ ?A\ ?x\ ?P\ ?P_{refl}\ ?x\ (\textsf{refl}\ ?A\ ?x) \rightarrowtriangle P_{refl}$$

However, this does not have to be checked when applying the rewrite rule since it is enforced by typing.

$$
\begin{array}{rcll}
A, B, t, u \ldots & ::= & \square_\ell & \text{universe at level } \ell \\
 & \mid & x & \text{variable} \\
 & \mid & f & \text{symbol} \\
 & \mid & \Pi\,(x : A).\, B & \text{dependent product type} \\
 & \mid & \lambda\,(x : A).\, t & \lambda\text{-abstraction} \\
 & \mid & t\ u & \text{application} \\
 & \mid & \mathbb{N} & \text{natural numbers} \\
 & \mid & \mathsf{zero} & \text{zero} \\
 & \mid & \mathsf{suc}\ t & \text{successor} \\
 & \mid & \mathsf{elim}\ P\ P_0\ P_S\ n & \text{natural number induction} \\
 & \mid & \Sigma\,(x : A).\, B & \text{dependent sum type} \\
 & \mid & (t, u) & \text{pair construction} \\
 & \mid & \pi_1\ t & \text{first projection} \\
 & \mid & \pi_2\ t & \text{second projection}
\end{array}
$$

$$
\begin{array}{lcll}
p, q \ldots & ::= & ?x \mid c \mid \mathbb{N} \mid \mathsf{zero} \mid \mathsf{suc}\ p \mid (p, q) & \text{pattern} \\
\mathbb{F} \ldots & ::= & \diamond\ p \mid \pi_1\ \diamond \mid \pi_2\ \diamond \mid \mathsf{elim}\ p\ p_0\ p_S\ \diamond & \text{pattern frame} \\
\mathbb{S} & ::= & \bullet \mid \mathbb{F}, \mathbb{S} & \text{(pattern) frame stack}
\end{array}
$$

Fig. 2. Term syntax and frames of RTT

5 A TYPE THEORY WITH COMPUTATIONAL ASSUMPTIONS

In this section, we give a formal description of Rewriting Type Theory (RTT) as an extension of the Predicative Calculus of Cumulative Inductive Constructions (PCUIC) as formalized in MetaCoq. For reasons of presentation, we omit some parts of PCUIC such as inductive datatypes other than $\mathbb{N}$, (co)fixpoints, and cumulativity. Our formalization adds rewrite rules to the full version of PCUIC. We use boxes to mark pieces of $\boxed{\text{new syntax}}$ where they are first introduced.

5.1 Syntax, Scope, and Substitution

The syntax for RTT is described in Figure 2. The usual dependent λ-calculus is present with type annotation for the domain of a λ-abstraction. The natural numbers $\mathbb{N}$ come with their unary representation and induction principle. There is a notion of symbols which represent new symbols introduced by a rewrite block (see below).

We define variable scope and substitutions as usual. Given a substitution σ, we write $\boxed{\sigma(x)}$ for the value assigned to the variable x by σ, and $\boxed{u\sigma}$ for the application of σ to the term u. In particular we have $x\sigma = \sigma(x)$. We implicitly assume uniqueness of identifiers: names are merely there to increase legibility but variables can be thought of as de Bruijn indices. We write $\boxed{[\overline{x := u}]}$ for the substitution replacing x_i by u_i for each index i. As a shorthand, we write $t[x := u]$ for single-variable substitution, and we abuse the notation by writing $[\Gamma := \overline{u}]$ for $[\overline{x := u}]$ if $\Gamma := \overline{x : A}$.

We write $\boxed{\mathrm{FV}(t)}$ for the set of free variables of term t. We define scopes of substitutions $\boxed{\sigma \in \Gamma \hookrightarrow \Delta}$ by saying that $\sigma \in \Gamma \hookrightarrow \Delta$ whenever $\mathrm{dom}(\sigma) = \Gamma$ and $\forall x \in \Gamma,\ \mathrm{FV}(\sigma(x)) \subseteq \Delta$. Given $\sigma \in \Xi \hookrightarrow \Delta$ and $\xi \in \Delta \hookrightarrow \Gamma$ we define the composition $\boxed{\sigma \cdot \xi}$ that maps $x \in \Xi$ to $(x\sigma)\xi$.

$\boxed{\mathcal{R} \vdash u \rightsquigarrow v}$ *Reduction of terms.*

$$\frac{}{\mathcal{R} \vdash (\lambda\,(x : A).\, t)\; u \rightsquigarrow t[x := u]} \qquad \frac{}{\mathcal{R} \vdash \mathsf{elim}\; P\; P_0\; P_S\; \mathsf{zero} \rightsquigarrow P_0}$$

$$\frac{}{\mathcal{R} \vdash \mathsf{elim}\; P\; P_0\; P_S\; (\mathsf{suc}\; n) \rightsquigarrow P_S\; n\; (\mathsf{elim}\; P\; P_0\; P_S\; n)} \qquad \frac{}{\mathcal{R} \vdash \pi_1\; (t, u) \rightsquigarrow t} \qquad \frac{}{\mathcal{R} \vdash \pi_2\; (t, u) \rightsquigarrow u}$$

$$\frac{[\overline{f : A} \mid \overline{R}] \in \mathcal{R} \qquad \Delta \ltimes \langle f_i \mid \mathbb{S} \rangle \twoheadrightarrow t \in \overline{R} \qquad \sigma : \Delta \hookrightarrow \Gamma}{\mathcal{R} \vdash \mathrm{eval}\,(f_i \mid \mathbb{S})\sigma \rightsquigarrow t\sigma}$$

Fig. 3. Definition of the reduction (congruence rules omitted)

5.2 Patterns, Rewrite Rules, and Signatures

Patterns. Patterns (also defined in Fig. 2) represent a subset of term syntax that mainly corresponds to constructor forms (except for λ-abstraction), *i.e.*, variables, symbols, inductive types (here $\mathbb{N}$) and constructors of inductive types (here zero and suc) and of pairs are patterns. The syntax for patterns does not include λ-abstractions or Π-types. Allowing these as patterns would lead to a possible extension of our theory to higher-order rewriting. Higher-order rewriting is already implemented in Agda, but its formalization is still work in progress so we leave it as the subject for future work.

Frame Stacks. A (pattern) frame stack is a list of pattern frames, which corresponds to the destructors of the syntax (application, projections and elimination) with a hole on the scrutinee (the matched term of the destructor) and when other arguments are patterns (see Fig. 2).

Rewrite Rules. A rewrite rule is of the form $\boxed{\Delta \ltimes \langle f_i \mid \mathbb{S} \rangle \twoheadrightarrow t}$ where Δ is the context of pattern variables, f_i is the *head symbol* which should be one of the symbols declared in the same rewrite block, $\mathbb{S}$ is a frame stack, and t is the right-hand side of the rewrite rule.

Signatures. Reduction and typing of RTT are parametrized over a global *signature* $\mathcal{R}$ that declares new symbols and rewrite rules on them. A signature consists of a sequence of *rewrite blocks* B of the form $\boxed{[\overline{f : A} \mid \overline{R}]}$, where $f_i : A_i$ are freshly declared symbols and $\overline{R}$ are rewrite rules on these symbols.

5.3 Reduction and Conversion

We define a judgement $\boxed{\mathcal{R} \vdash u \rightsquigarrow v}$ corresponding to full reduction in Fig. 3. Since reduction makes use of the rewrite rules, it is defined relative to a signature $\mathcal{R}$. The only rule that differs from standard type theory is the one for applying a rewrite rule. It is used when a symbol f_i declared in the signature is applied to a stack that is an instance of the pattern stack of one of the rewrite rules declared in the same block as f_i. Rewrite rules only apply when the stack matches the patterns *on the nose*, but this is not an issue since we allow full reduction in all subterms.

The reduction of a rewrite rule makes use of the application of a term t to a stack pattern $\mathbb{S}$ which is written as $\boxed{\mathrm{eval}\,(t \mid \mathbb{S})}$ and is defined as $\mathrm{eval}\,(t \mid \bullet) = t$ on the empty stack and otherwise:

$$\begin{array}{lcl||lcl} \mathrm{eval}\,(t \mid \diamond\; u, \mathbb{S}) & = & \mathrm{eval}\,(t\; u \mid \mathbb{S}) & \mathrm{eval}\,(t \mid \pi_1\; \diamond, \mathbb{S}) & = & \mathrm{eval}\,(\pi_1\; t \mid \mathbb{S}) \\ \mathrm{eval}\,(t \mid \mathsf{elim}\; P\; P_0\; P_S\; \diamond, \mathbb{S}) & = & \mathrm{eval}\,(\mathsf{elim}\; P\; P_0\; P_S\; t \mid \mathbb{S}) & \mathrm{eval}\,(t \mid \pi_2\; \diamond, \mathbb{S}) & = & \mathrm{eval}\,(\pi_2\; t \mid \mathbb{S}) \end{array}$$

$\boxed{\mathcal{R} \vdash \Gamma}$ *Context formation.*

$$\frac{}{\mathcal{R} \vdash \bullet} \qquad \frac{\mathcal{R}; \Gamma \vdash A}{\mathcal{R} \vdash \Gamma, x : A}$$

$\boxed{\mathcal{R}; \Gamma \vdash t : A}$ *Typing.*

$$\frac{(x : A) \in \Gamma \qquad \mathcal{R} \vdash \Gamma}{\mathcal{R}; \Gamma \vdash x : A} \qquad \frac{(f : A) \in \mathcal{R}}{\mathcal{R}; \Gamma \vdash f : A} \qquad \frac{\mathcal{R}; \Gamma \vdash t : A \qquad \mathcal{R} \vdash A = B \qquad \mathcal{R}; \Gamma \vdash B : \square_\ell}{\mathcal{R}; \Gamma \vdash t : B}$$

$$\frac{\mathcal{R} \vdash \Gamma}{\mathcal{R}; \Gamma \vdash \square_\ell : \square_{\ell+1}} \qquad \frac{\mathcal{R}; \Gamma \vdash A : \square_{\ell_1} \qquad \mathcal{R}; \Gamma, x : A \vdash B : \square_{\ell_2}}{\mathcal{R}; \Gamma \vdash \Pi\, (x : A).\, B : \square_{\max(\ell_1, \ell_2)}}$$

$$\frac{\mathcal{R}; \Gamma, x : A \vdash t : B}{\mathcal{R}; \Gamma \vdash \lambda\, (x : A).\, t : \Pi\, (x : A).\, B} \qquad \frac{\mathcal{R}; \Gamma \vdash t : \Pi\, (x : A).\, B \qquad \mathcal{R}; \Gamma \vdash u : A}{\mathcal{R}; \Gamma \vdash t\ u : B[x := u]}$$

$$\frac{\mathcal{R} \vdash \Gamma}{\mathcal{R}; \Gamma \vdash \mathbb{N} : \square_0} \qquad \frac{\mathcal{R} \vdash \Gamma}{\mathcal{R}; \Gamma \vdash \mathsf{zero} : \mathbb{N}} \qquad \frac{\mathcal{R}; \Gamma \vdash n : \mathbb{N}}{\mathcal{R}; \Gamma \vdash \mathsf{suc}\ n : \mathbb{N}}$$

$$\frac{\begin{array}{c} \mathcal{R}; \Gamma \vdash P : \mathbb{N} \to \square_\ell \\ \mathcal{R}; \Gamma \vdash P_0 : P\ \mathsf{zero} \qquad \mathcal{R}; \Gamma \vdash P_S : \Pi\, (n : \mathbb{N}).\, P\ n \to P\ (\mathsf{suc}\ n) \qquad \mathcal{R}; \Gamma \vdash t : \mathbb{N} \end{array}}{\mathcal{R}; \Gamma \vdash \mathsf{elim}\ P\ P_0\ P_S\ t : P\ t}$$

$$\frac{\mathcal{R}; \Gamma \vdash A : \square_{\ell_1} \qquad \mathcal{R}; \Gamma, x : A \vdash B : \square_{\ell_2}}{\mathcal{R}; \Gamma \vdash \Sigma\, (x : A).\, B : \square_{\max(\ell_1, \ell_2)}} \qquad \frac{\mathcal{R}; \Gamma \vdash t : A \qquad \mathcal{R}; \Gamma \vdash u : B[x := t]}{\mathcal{R}; \Gamma \vdash t, u : \Sigma\, (x : A).\, B}$$

$$\frac{\mathcal{R}; \Gamma \vdash t : \Sigma\, (x : A).\, B}{\mathcal{R}; \Gamma \vdash \pi_1\ t : A} \qquad \frac{\mathcal{R}; \Gamma \vdash t : \Sigma\, (x : A).\, B}{\mathcal{R}; \Gamma \vdash \pi_2\ t : B[x := \pi_1\ t]}$$

Fig. 4. Typing judgments for RTT.

Many-step reduction $\boxed{\mathcal{R} \vdash u \rightsquigarrow^\star v}$ is defined as the reflexive transitive closure of the reduction relation $\mathcal{R} \vdash u \rightsquigarrow v$. Since reduction can reduce subterms in any position, we define conversion $\boxed{\mathcal{R} \vdash u = v}$ as the symmetric and transitive closure of many-step reduction. This implies that conversion depends on the signature $\mathcal{R}$.

5.4 Typing and Signature Validity

The typing rules of the system are described in Fig. 4. We introduce the judgements $\boxed{\mathcal{R}; \Gamma \vdash t : A}$ stating term t has type A in signature $\mathcal{R}$ and context Γ and $\boxed{\mathcal{R} \vdash \Gamma}$ stating Γ is a well-formed context. The typing rules are the standard typing rules of type theory. The only difference lies in the definition of the conversion rule, which depends on the rewrite rules in the signature. We refer the reader to [Sozeau et al. 2020] for an explanation of the typing rules.

Signature validity. Although the definition of the typing system does not require anything on the signature $\mathcal{R}$, the metatheory of RTT can only be stated in a setting where the rewrite blocks are all valid, which ensures for instance that rewrite rules preserve typing.

$\boxed{\Gamma \mid \bar{x} \vdash p}$ *Pattern validity.* (Pattern variables are required to occur linearly)

$$\frac{}{\Gamma \mid x \vdash x} \qquad \frac{f \in \Gamma}{\Gamma \mid \bullet \vdash f} \qquad \frac{}{\Gamma \mid \bullet \vdash \mathbb{N}} \qquad \frac{}{\Gamma \mid \bullet \vdash \mathsf{zero}} \qquad \frac{\Gamma \mid \bar{x} \vdash p}{\Gamma \mid \bar{x} \vdash \mathsf{suc}\ p} \qquad \frac{\Gamma \mid \bar{x}_1 \vdash p \qquad \Gamma \mid \bar{x}_2 \vdash q}{\Gamma \mid \bar{x}_1, \bar{x}_2 \vdash (p, q)}$$

$\boxed{\Gamma \mid \bar{x} \vdash \mathbb{S}}$ *Frame stack validity.*

$$\frac{\Gamma \mid \bar{x}_1 \vdash p \qquad \Gamma \mid \bar{x}_2 \vdash \mathbb{S}}{\Gamma \mid \bar{x}_1, \bar{x}_2 \vdash \diamond\, p :: \mathbb{S}} \qquad \frac{\Gamma \mid \bar{x} \vdash \mathbb{S}}{\Gamma \mid \bar{x} \vdash \pi_1 \diamond :: \mathbb{S}} \qquad \frac{\Gamma \mid \bar{x} \vdash \mathbb{S}}{\Gamma \mid \bar{x} \vdash \pi_2 \diamond :: \mathbb{S}}$$

$$\frac{\Gamma \mid \bar{x}_1 \vdash P \qquad \Gamma \mid \bar{x}_2 \vdash P_0 \qquad \Gamma \mid \bar{x}_3 \vdash P_S \qquad \Gamma \mid \bar{x}_4 \vdash \mathbb{S}}{\Gamma \mid \bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4 \vdash \mathsf{elim}\ P\ P_0\ P_S \diamond :: \mathbb{S}}$$

$\boxed{\mathcal{R}; \Gamma \vdash \Xi \ltimes \langle f_i \mid \mathbb{S} \rangle \rightharpoonup v}$ *Rewrite rule validity.*

$$\frac{f_i : A \in \Gamma \qquad \Gamma_{\mathtt{rig}} \mid \Xi \vdash \mathbb{S} \qquad \forall \Delta.\ \forall T.\ \forall (\sigma : \Xi \hookrightarrow \Gamma\Delta).\ \ \mathcal{R}; \Gamma\Delta \vdash \mathrm{eval}\,(f_i \mid \mathbb{S})\sigma : T \Longrightarrow \mathcal{R}; \Gamma\Delta \vdash v\sigma : T}{\mathcal{R}; \Gamma \vdash \Xi \ltimes \langle f_i \mid \mathbb{S} \rangle \rightharpoonup v}$$

$\boxed{\mathcal{R} \vdash B}$ *Rewrite block validity.*

$$\frac{\mathcal{R}; \bullet \vdash \overline{f : A} \qquad \forall R \in \overline{R}.\ \mathcal{R}; \overline{f : A} \vdash R}{\mathcal{R} \vdash [\overline{f : A} \mid \overline{R}]}$$

$\boxed{\vdash \mathcal{R}}$ *Signature validity.*

$$\frac{}{\vdash \bullet} \qquad \frac{\vdash \mathcal{R} \qquad \mathcal{R} \vdash B}{\vdash \mathcal{R}, B}$$

Fig. 5. Rules for validity of patterns, frame stacks, rewrite rules, rewrite blocks, and whole signatures.

The rules for validity of signatures, rewrite rules, and patterns are given in Fig. 5. $\boxed{\vdash \mathcal{R}}$ checks the validity of a complete signature by checking the validity of each individual rewrite block with $\boxed{\mathcal{R} \vdash B}$, where the type of the fresh symbols can depend on symbols from previous rewrite blocks. $\boxed{\mathcal{R}; \Gamma \vdash \Xi \ltimes \langle f_i \mid \mathbb{S} \rangle \rightharpoonup v}$ certifies the validity of a rewrite rule where Γ is the context of symbols of the rewrite block, and $\Gamma_{\mathtt{rig}}$ its restriction to symbols which are not the head symbol of a rewrite rule. It makes sure that the head symbol of the rewrite rule is one of the symbols declared in the same rewrite block, that the patterns are well-formed, and that the rule preserves typing [Blanqui 2005]. To ensure rewrite rules preserve typing even in an open context, it allows the context to be extended with an arbitrary telescope Δ. This restriction is strictly more liberal than requiring that the left- and right-hand side can both be typed at the same type. $\boxed{\Gamma \mid \bar{x} \vdash p}$ and $\boxed{\Gamma \mid \bar{x} \vdash \mathbb{S}}$ finally assert the well-formedness of a pattern (resp. pattern stack) with symbols Γ and pattern variables $\bar{x}$. In particular, they ensure that all pattern variables are used linearly.

Example 5.1. In Fig. 6 we write down the example from Sect. 2.3 as a rewrite block in the formal syntax of RTT. This example assumes the presence of the inductive type list in the theory, which can be introduced in a similar way to $\mathbb{N}$. We also make use of the $\to$ syntax for non-dependent function types.

$$
\begin{array}{ll}
[& \mathsf{Cor} : \square \\
, & \mathsf{Over} : \mathsf{Cor} \\
, & \mathsf{Next} : ((\mathsf{Cor} \to \mathsf{list}\ \mathbb{N}) \to \mathsf{list}\ \mathbb{N}) \to \mathsf{Cor} \\
, & \mathsf{elim}_{\mathsf{Cor}} : \Pi\, (P : \mathsf{Cor} \to \square).\ \Pi\, (P_{\mathsf{Over}} : P\ \mathsf{Over}). \\
& \qquad\qquad \Pi\, (P_{\mathsf{Next}} : \Pi\, (f : (\mathsf{Cor} \to \mathsf{list}\ \mathbb{N}) \to \mathsf{list}\ \mathbb{N}).\ P\ (\mathsf{Next}\ f)).\ \Pi\, (c : \mathsf{Cor}).\ P\ c \\
, & \bullet \\
| & (P : \mathsf{Cor} \to \square, P_{\mathsf{Over}} : P\ \mathsf{Over}, P_{\mathsf{Next}} : \Pi\, (f : (\mathsf{Cor} \to \mathsf{list}\ \mathbb{N}) \to \mathsf{list}\ \mathbb{N}).\ P\ (\mathsf{Next}\ f), \bullet) \\
& \quad \ltimes\ \langle \mathsf{elim}_{\mathsf{Cor}} \mid \diamond\, ?P, \diamond\, ?P_{\mathsf{Over}}, \diamond\, ?P_{\mathsf{Next}}, \diamond\, \mathsf{Over}, \bullet \rangle \rightarrowtriangle P_{\mathsf{Over}} \\
, & (P : \mathsf{Cor} \to \square, P_{\mathsf{Over}} : P\ \mathsf{Over}, P_{\mathsf{Next}} : \Pi\, (f : (\mathsf{Cor} \to \mathsf{list}\ \mathbb{N}) \to \mathsf{list}\ \mathbb{N}).\ P\ (\mathsf{Next}\ f), \\
& \ f : (\mathsf{Cor} \to \mathsf{list}\ \mathbb{N}) \to \mathsf{list}\ \mathbb{N}, \bullet) \\
& \quad \ltimes\ \langle \mathsf{elim}_{\mathsf{Cor}} \mid \diamond\, ?P, \diamond\, ?P_{\mathsf{Over}}, \diamond\, ?P_{\mathsf{Next}}, \diamond\, (\mathsf{Next}\ f), \bullet \rangle \rightarrowtriangle P_{\mathsf{Next}}\ f \\
, & \bullet\quad]
\end{array}
$$

Fig. 6. The example of Sect. 2.3 in the formal syntax of RTT.

6 METATHEORY OF RTT

In this section, we present our approach to modular confluence of RTT. The correctness of our results has been formalized in the Coq proof assistant, using the MetaCoq project [Sozeau et al. 2020]. The MetaCoq project provides a solid basis for our work as confluence and subject reduction have already been formalized for PCUIC [Sozeau et al. 2019]. Therefore, our work can be seen as the search for a modular and decidable criterion to guarantee that the proof of confluence and subject reduction for PCUIC can be lifted to RTT. Compared to what has been presented in Section 5, our formalization of RTT features general inductive types and co-inductive types, a cumulative hierarchy of predicative universes, an impredicative universe and a `let-in` construct.

6.1 Parallel Reduction and the Triangle Criterion

Parallel reduction of RTT is defined in Fig. 7. Note that $\Rrightarrow$ is closed under substitution. In fact, we even have the stronger property that it is closed in one step in the following sense:

Lemma 6.1. [PCUICParallelReductionConfluence.v/substitution_pred1] *Let u be a term and σ a substitution. If $\mathcal{R} \vdash u \Rrightarrow v$ and $\mathcal{R} \vdash x\sigma \Rrightarrow x\sigma'$ for each $x \in \mathrm{dom}(\sigma)$, then $\mathcal{R} \vdash u\sigma \Rrightarrow v\sigma'$.*

Proof. By induction on the derivation of $\mathcal{R} \vdash u \Rrightarrow v$. In the base case where u is a variable x, the only possible reduction is $\mathcal{R} \vdash x \Rrightarrow x$ hence $\mathcal{R} \vdash x\sigma \Rrightarrow x\sigma'$ follows by assumption. □

Both property 1 ($\rightsquigarrow\ \subseteq\ \Rrightarrow\ \subseteq\ \rightsquigarrow^\star$) and Lemma 4.2 are valid in this setting.

We now want to state a criterion on $\mathcal{R}$ to ensure that the RTT satisfies the triangle property. The first step is to give a more concrete description of the function ρ. To this end, we assume given a fixed order $>$ on the set of rewrite rules and define from it a specific function $\boxed{\rho_>}$.

Definition 6.2. Let $(\mathcal{R}, >)$ be a set of rewrite rules with a total order $>$. We define the function $\rho_>$ on the set of all terms of RTT as follows:

- If u matches a left-hand side of a rewrite rule in $\mathcal{R}$, let $\Delta \ltimes l \rightarrowtriangle r$ be the first rule for which the left-hand side matches (w.r.t. to $>$) and let σ be the resulting substitution, i.e. $u = l\sigma$. We define a new substitution σ' inductively by $x\sigma' := \rho_>(x\sigma)$ for every variable $x \in \mathrm{dom}(\sigma)$. We then define $\rho_>(u) := r\sigma'$.
- If $u = (\lambda\, (x : A).\ t)\ t'$, then $\rho_>(u) = \rho_>(t)[x := \rho_>(t')]$.
- If $u = \mathsf{elim}\ P\ P_0\ P_S\ \mathsf{zero}$, then $\rho_>(u) = \rho_>(P_0)$.

$\boxed{\mathcal{R} \vdash u \Rrightarrow v}$ *Parallel reduction of terms.* ($\mathcal{R}$ omitted unless relevant)

$$\frac{}{t \Rrightarrow t} \qquad \frac{t \Rrightarrow t' \qquad u \Rrightarrow u'}{(\lambda\,(x : A).\ t)\ u \Rrightarrow t'[x := u']} \qquad \frac{P_0 \Rrightarrow P_0'}{\mathsf{elim}\ P\ P_0\ P_S\ \mathsf{zero} \Rrightarrow P_0'}$$

$$\frac{n \Rrightarrow n' \qquad P \Rrightarrow P' \qquad P_0 \Rrightarrow P_0' \qquad P_S \Rrightarrow P_S'}{\mathsf{elim}\ P\ P_0\ P_S\ (\mathsf{suc}\ n) \Rrightarrow P_S'\ n'\ (\mathsf{elim}\ P'\ P_0'\ P_S'\ n')} \qquad \frac{t \Rrightarrow t'}{\pi_1\ (t, u) \Rrightarrow t'} \qquad \frac{u \Rrightarrow u'}{\pi_1\ (t, u) \Rrightarrow u'}$$

$$\frac{[\overline{f : A} \mid \overline{R}] \in \mathcal{R} \qquad \Delta \ltimes \langle f_i \mid \mathbb{S} \rangle \twoheadrightarrow t \in \overline{R} \qquad \sigma : \Delta \hookrightarrow \Gamma \qquad \forall x \in \Delta.\ x\sigma \Rrightarrow x\sigma'}{\mathcal{R} \vdash \mathrm{eval}\,(f_i \mid \mathbb{S})\sigma \Rrightarrow t\sigma'}$$

$$\frac{A \Rrightarrow A' \qquad B \Rrightarrow B'}{\Pi\,(x : A).\ B \Rrightarrow \Pi\,(x : A').\ B'} \qquad \frac{t \Rrightarrow t'}{\lambda\,(x : A).\ t \Rrightarrow \lambda\,(x : A).\ t'} \qquad \frac{t \Rrightarrow t' \qquad u \Rrightarrow u'}{t\ u \Rrightarrow t'\ u'}$$

$$\frac{t \Rrightarrow t'}{\mathsf{suc}\ t \Rrightarrow \mathsf{suc}\ t'} \qquad \frac{t \Rrightarrow t' \qquad u \Rrightarrow u'}{t, u \Rrightarrow t', u'} \qquad \frac{A \Rrightarrow A' \qquad B \Rrightarrow B'}{\Sigma\,(x : A).\ B \Rrightarrow \Sigma\,(x : A').\ B'}$$

$$\frac{P \Rrightarrow P' \qquad P_0 \Rrightarrow P_0' \qquad P_s \Rrightarrow P_s' \qquad t \Rrightarrow t'}{\mathsf{elim}\ P\ P_0\ P_s\ t \Rrightarrow \mathsf{elim}\ P'\ P_0'\ P_s'\ t'}$$

Fig. 7. Definition of parallel reduction.

- If $u = \mathsf{elim}\ P\ P_0\ P_S\ (\mathsf{suc}\ n)$, then $\rho_{>}(u) = \rho_{>}(P_S)\ \rho_{>}(n)\ (\mathsf{elim}\ \rho_{>}(P)\ \rho_{>}(P_0)\ \rho_{>}(P_S)\ \rho_{>}(n))$.
- Otherwise, we define $\rho_{>}(u)$ by applying ρ to every direct subterm of u. For example, if $u = u_1\ u_2$ then $\rho_{>}(u) := \rho_{>}(u_1)\ \rho_{>}(u_2)$.

Note that the definition of the function $\rho_{>}$ is well-founded even for non-terminating rewrite rules. Intuitively, it maximally unfolds all redexes that are visible immediately, but not any additional redexes created by unfolding some part of the term. Hence $\rho_{>}(u)$ is the maximal *one-step* parallel reduction of u.

We now define the triangle criterion that is at the heart of the modular confluence of RTT.

Definition 6.3 (triangle criterion). $(\mathcal{R}, \succ)$ satisfies the *triangle criterion* when:

(1) For every term u such that there is a rewrite rule $\Delta \ltimes l \twoheadrightarrow r$ in $\mathcal{R}$ and a pattern substitution σ such that $u = l\sigma$, if $\Delta' \ltimes l' \twoheadrightarrow r'$ is the first rule that matches u with substitution τ in $\mathcal{R}$ ($u = l'\tau$), then
$$r\sigma \Rrightarrow r'(\rho_{>}\tau).$$
(2) Also, for every pair of rewrite rules $\Delta \ltimes l \twoheadrightarrow r$ and $\Delta' \ltimes l' \twoheadrightarrow r'$ in $\mathcal{R}$, for every pattern substitution σ and τ and for every stack pattern $\mathbb{S}$, $\mathrm{eval}\,(l \mid \mathbb{S})\sigma = l'\tau$ implies $\mathbb{S} = \bullet$.

The first condition says that the triangle property holds individually for every instance of a left-hand side of each rule of the rewrite block. The second condition is more technical and is here to avoid situations where two rewrite rules have apparently no critical pair because the shapes of the pattern stacks are different. For instance, it prevents the situation where a symbol f has two rewrite rules of different arities $f\ x \twoheadrightarrow \lambda\,(y : A).\ a$ and $f\ x\ y \twoheadrightarrow b$ with $a \neq b$. Here the first condition trivially holds, but only because the critical pair is "hidden". Indeed, changing the first rule with the (up-to η) equivalent rule $f\ x\ y \twoheadrightarrow a$ now breaks the first condition.

6.2 Confluence and Subject Reduction

Theorem 6.4 (Confluence). [PCUICConfluence.v/red_confluence] *If there exists an order $\succ$ on $\mathcal{R}$ such that $(\mathcal{R}, \succ)$ satisfies the triangle criterion, then* RTT *is confluent.*

Proof. Using the triangle criterion, we can show that parallel reduction satisfies the triangle property, and conclude by Theorem 4.2. □

For the rest of the section, we assume that there exists an order $\succ$ on $\mathcal{R}$ such that $(\mathcal{R}, \succ)$ satisfies the triangle criterion.

Lemma 6.5 (Injectivity of Π-types). [PCUICConversion.v/cumul_Prod_inv] *If $\mathcal{R} \vdash \Pi\ (x : A).\ B = \Pi\ (x : A').\ B'$ then $\mathcal{R} \vdash A = A'$ and $\mathcal{R} \vdash B = B'$.*

Proof. We remark that any conversion can be proven without transitivity. Using confluence of the reduction, we conclude that conversion corresponds to first reducing both sides and then applying reflexivity or a congruence rule. We conclude because Π-types have no reduction rule. □

Note that in our more general setting with inductive types, the injectivity of inductive types can be formulated and proven in the same way. We can now turn to subject reduction.

Lemma 6.6 (Subject Reduction). [PCUICSR.v/subject_reduction] *If $\mathcal{R}; \Gamma \vdash t : A$ and $t \rightsquigarrow u$ then $\mathcal{R}; \Gamma \vdash u : A$.*

Proof. By case analysis on the reduction. If it is a rewrite rule, then the property holds by validity of the rewrite rule. If it is a β reduction, then we have $(\lambda\ (x : A).\ t)\ u \rightsquigarrow t[x := u]$ and $\mathcal{R}; \Gamma \vdash (\lambda\ (x : A).\ t)\ u : T$. By inversion of typing, we know that $\mathcal{R}; \Gamma \vdash \lambda\ (x : A).\ t : \Pi\ (x : C).\ D$ and $\mathcal{R}; \Gamma \vdash u : C$ and $\mathcal{R} \vdash D[x := u] = T$. Using inversion again, we get $\mathcal{R}; \Gamma, x : A \vdash t : B$ and $\mathcal{R}; \Gamma \vdash \Pi(x : A).B = \Pi(x : C).D$. We conclude that $\mathcal{R}; \Gamma \vdash t[x := u] : D[x := u]$ using that $A = C$ and $B = D$ by Lemma 6.5. (A similar argument holds for pattern-matching on inductive types.) □

6.3 Modularity of the Triangle Criterion

For RTT to be tractable and usable in practice, we need an additional modularity property of the triangle criterion that guarantees, *e.g.*, that importing a new library that makes use of rewrite rules cannot break confluence because of bad interaction with already imported libraries. Also for efficiency reasons, modularity allows to check the triangle criterion locally without taking all the already introduced rewrite rules into account, which is necessary for the system to scale up.

Theorem 6.7 (Modularity of the triangle criterion). [PCUICParallelReductionConfluence.v/triangle_rules_weakening] *Let $\mathcal{R}$ be a rewriting signature and $\succ$ an order on it. If the triangle criterion holds for each individual rewrite block $[\overline{f : A} \mid \overline{R}] \in \mathcal{R}$, then it also holds for $\mathcal{R}$.*

6.4 Consistency

Until now, we have looked at the computational properties of the system. But as we are dealing with an extension of MLTT, we also expect logical properties. The most important one is consistency. In general, a rewrite block may contain symbols of a type that is otherwise uninhabited, so declaring a rewrite block does not necessarily preserve consistency, in the same way as introducing axioms may break consistency. However, if we can provide an *instantiation* of the rewrite block that satisfies each of the rewrite rules propositionally, then we can ensure consistency. More precisely, an instantiation of a rewrite block $[\overline{f : A} \mid \overline{R}]$ is a pair $(\bar{u}, \bar{e})$ such that:

- For each f_i, we have $\vdash u_i : A_i[\overline{f_j := u_j}]$
- For each $R_i = (\Xi \ltimes \langle f_j \mid \mathbb{S} \rangle \rightarrow v) \in \overline{R}$, we have $\vdash e_i : \mathrm{eval}\,(f_j \mid \mathbb{S})\sigma = v\sigma$ with $\sigma = [\overline{f_k := u_k}]$.

In particular, each u_i and e_i must be typable in an empty signature (and empty context). This restriction is only required to avoid the presence of absurd assumptions in the context; it has nothing to do with the definition of rewrite blocks in the system which can very well depend on previously declared rewrite blocks.

Lemma 6.8 (Consistency). [This lemma has not been formalized] *Let $\mathcal{R}$ be a signature such that each rewrite block in $\mathcal{R}$ can be instantiated. Then there is no closed term t such that $\vdash t : \Pi\,(X : \square_\ell).\,X$.*

Proof. The proof goes by defining a translation from RTT to an extensional type theory (ETT). Each symbol declared in $\mathcal{R}$ is translated to its corresponding term in the instantiation of its block, and each application of a rewrite rule is translated to an application of the reflection rule with the propositional equality provided by the instantiation. The rest of the translation is simply the identity. We thus directly deduce consistency of RTT from consistency of ETT. □

For instance, in the case of the parallel plus function (Section 2.1), the consistency of the theory directly follow from the fact that each rewrite rule already holds propositionally for the standard plus function. This lemma also has an interesting application. It allows us to give one specific implementation of a signature and prove certain equalities about it that are exposed as rewrite rules instead of the computation rules one would normally get. For example, we can give an efficient binary implementation of natural numbers but hide their implementation from the user, and instead expose the simpler computation rules that one would get from a unary implementation.

6.5 Decidability of Type-Checking

As for consistency, the addition of rewrite rules to the system may also break decidability of type checking. We focus here on the algorithm to decide conversion as it is the only place where type checking differs from the regular algorithm for CIC.

To ensure type checking remains decidable, the most obvious requirement is to ask that the rewrite system induced by a rewrite context $\mathcal{R}$ is weakly normalising. This is because confluence implies the uniqueness of normal forms, and thus conversion can be decided by syntactically comparing normal forms. As rewrite rules only modify conversion with respect to CIC, the same argument for the decidability of type-checking for CIC applies to RTT.

What is perhaps less obvious is that, even in the absence of normalisation, the usual algorithm for deciding type checking for CIC by comparing the weak-head normal forms still makes sense. To make this statement precise, we assume for a given signature $\mathcal{R}$ the existence of a (partial) function $\mathtt{whnf}_{\mathcal{R}}$ on terms that satisfies the following properties for all terms u where it is defined:

(1) $\mathcal{R} \vdash u \rightsquigarrow^{\star} \mathtt{whnf}_{\mathcal{R}}(u)$.
(2) If $\mathcal{R} \vdash \mathtt{whnf}_{\mathcal{R}}(u) \rightsquigarrow^{\star} v$ for some term v, then $\mathtt{whnf}_{\mathcal{R}}(u)$ and v have the same top-level term constructor, and $\mathcal{R} \vdash u' \rightsquigarrow^{\star} v'$ for each pair of corresponding direct subterms u' and v'.

The second property gives us for example that if $\mathtt{whnf}_{\mathcal{R}}(u) = u_1\ u_2$, then v is also an application $v_1\ v_2$, and we have $\mathcal{R} \vdash u_1 \rightsquigarrow^{\star} v_1$ and $\mathcal{R} \vdash u_2 \rightsquigarrow^{\star} v_2$. The definition of a concrete function $\mathtt{whnf}_{\mathcal{R}}$ is out of scope of this paper, but the interested reader can find a detailed example in [Cockx 2020].

Algorithm 6.9 (Algorithmic Conversion). *Assume we have a (partial) function $\mathtt{whnf}_{\mathcal{R}}$ satisfying properties (1) and (2) defined above. Then algorithmic conversion for a signature $\mathcal{R}$ is given by the following procedure. Given two terms u and v, it first compares syntactically the two terms. If they are the same, it answers true. Otherwise, it computes $\mathtt{whnf}_{\mathcal{R}}(u)$ and $\mathtt{whnf}_{\mathcal{R}}(v)$. If the top-level term constructors of $\mathtt{whnf}_{\mathcal{R}}(u)$ and $\mathtt{whnf}_{\mathcal{R}}(v)$ are different, it answers false. Otherwise, it tests for the conversion of all corresponding pairs of direct subterms of $\mathtt{whnf}_{\mathcal{R}}(u)$ and $\mathtt{whnf}_{\mathcal{R}}(v)$ recursively.*

Theorem 6.10. [This lemma has not been formalized] *Let $\mathcal{R}$ be a valid signature in* RTT *that satisfies the triangle criterion. If algorithmic conversion terminates on inputs u and v, then its answer is* true *if and only if $\mathcal{R} \vdash u = v$.*

Proof. Assume the algorithm terminates when given inputs u and v—i.e. $\mathtt{whnf}_{\mathcal{R}}(u)$ and $\mathtt{whnf}_{\mathcal{R}}(v)$ are defined, as are all the recursive calls to $\mathtt{whnf}_{\mathcal{R}}$ made by the algorithm. The proof proceeds by induction on the number of calls to $\mathtt{whnf}_{\mathcal{R}}$ made by the algorithm. If two terms are syntactically equal, then they are obviously convertible. Otherwise, if the top-level term constructors of their their weak head normal forms are different, then by property (2) of $\mathtt{whnf}_{\mathcal{R}}$, they can never be reduced to the same term, hence they are not convertible by confluence of reduction. On the other hand, if their weak-head normal forms have the same top-level term constructors, then they are convertible if and only if their direct subterms are convertible. So by the inductive hypothesis, the result of the algorithm is true if and only if $\mathcal{R} \vdash \mathtt{whnf}_{\mathcal{R}}(u) = \mathtt{whnf}_{\mathcal{R}}(v)$. By property (1) of $\mathtt{whnf}_{\mathcal{R}}$, we have that $\mathcal{R} \vdash u = v$ if and only if $\mathcal{R} \vdash \mathtt{whnf}_{\mathcal{R}}(u) = \mathtt{whnf}_{\mathcal{R}}(v)$, thus finishing the proof. □

The partial correctness property is interesting because it ensures that RTT is still usable in practice, even when the system is not normalizing: when the system provides an answer, the answer is correct and subjection reduction is preserved. This is in contrast to extensional type theory which has no practical type checking algorithm.

7 CONFLUENCE CHECKING AND Agda IMPLEMENTATION

To guarantee the metatheoretical properties in the previous section, confluence of reduction is crucial. In particular, confluence is important for the practical feasibility of type checking: if a term has more than one normal form, we lose the ability to check conversion by reducing both sides to normal form. Hence we check that any set of rewrite rules given by the user satisfies the triangle criterion (Definition 6.3). In this section, we provide an algorithm that checks the triangle criterion. We also discuss how we integrate rewrite rules in the Agda proof assistant and implement the triangle criterion in this setting.

7.1 Checking the Triangle Criterion

Consider a set of rewrite rules $\mathcal{R}$ for which we want to check that the triangle property holds by using the triangle criterion (Definition 6.3). The problem with this criterion is that it quantifies over all possible terms u, so checking all of them would take a long time indeed. Instead, we only check that it holds for all the left-hand sides of the rewrite rules. For this check to be sufficient, we may have to add some auxiliary rewrite rules that are only important for this check. For example, when there are two rewrite rules $m + 0 \rightarrow m$ and $0 + n \rightarrow n$ we require an additional rule $0 + 0 \rightarrow 0$.

In general, we require that the set of left-hand sides of the rewrite rules is *closed under unification*, i.e. if two patterns in C overlap, then their most general unifier must be in C as well.

Definition 7.1. A *most general unifier* of two patterns p and q is a substitution σ such that $p\sigma = q\sigma$, and moreover every other substitution τ with $p\tau = q\tau$ is of the form $\tau = \sigma \cdot \upsilon$.

Definition 7.2. Let $(C, \succ)$ be an ordered set of patterns. We say that $(C, \succ)$ is *closed under unification* if for every pair of patterns $p, q \in C$ such that p unifies with q with most general unifier σ, we also have $p\sigma \in C$, and $p\sigma \succ p$.

We now state the main theorem used by our algorithm for checking the triangle criterion.

Theorem 7.3. *Let $\mathcal{R}$ be a rewriting signature and let C be the set of left-hand sides of the rules in $\mathcal{R}$. Let further $\succ$ be a total order on C and assume that $(C, \succ)$ is closed under unification. Assume further that for every pair of rewrite rules $\Delta \ltimes \langle f \mid \mathbb{S} \rangle \rightarrow r$ and $\Delta \ltimes \langle f \mid \mathbb{S}' \rangle \rightarrow r'$ with the same* *head symbol f, the stacks $\mathbb{S}$ and $\mathbb{S}'$ have the same length. If for every rewrite rule $\Delta \ltimes p \rightarrow r \in \mathcal{R}$ and every parallel reduction step $\mathcal{R} \vdash p \Rrightarrow u$ we have $\mathcal{R} \vdash u \Rrightarrow r$, then $(\mathcal{R}, \succ)$ satisfies the triangle criterion (Definition 6.3).*

Proof. First, from the condition that two rewrite rules with the same head symbols have pattern stacks of the same length, the second requirement of the triangle criterion follows immediately: if the pattern stacks have the same length, adding anything extra to one of them can never make them equal. Now consider a term u and a rewrite rule $\Delta \ltimes l \rightarrow r$ in $\mathcal{R}$ such that $u = l\theta$ for some substitution θ and let $\Delta' \ltimes p \rightarrow r'$ be the first rule that matches with substitution σ, i.e. $u = p\sigma$. Let $v = r\theta$, then we have to prove that $\mathcal{R} \vdash v \Rrightarrow r'\sigma'$ where $x\sigma' = \rho_{\succ}(x\sigma)$ for each $x \in \mathrm{dom}(\sigma)$. Since C is closed under unification and $l\theta = p\sigma$, there exists some $q \in C$ such that $q \succ l$, $q \succ p$, q is an instance of both l and p, and u matches q. However, p is the first pattern in C matched by u so we must have $q = p$. In particular, p is an instance of l, say $p = l\tau$. Then we have $l\theta = u = p\sigma = l\tau\sigma$, hence $\theta = \tau \cdot \sigma$. Now since $\mathcal{R} \vdash p = l\tau \Rrightarrow r\tau$, by assumption of the theorem we have $\mathcal{R} \vdash r\tau \Rrightarrow r'$. Hence we also have $\mathcal{R} \vdash v = r\theta = r\tau\sigma \Rrightarrow r'\sigma'$, as we wanted to prove. □

For any given term u, the set of terms v such that $u \Rrightarrow v$ is finite and can be computed by exhaustively applying all the rules. Hence we can use the above theorem for our algorithm for checking the triangle criterion.

Algorithm 7.4. *We check the triangle criterion for a given rewrite block $\mathcal{R} = [\overline{f : A} \mid \overline{R}]$ as follows.*

- *First, we set some fixed order $\succ$ on the rewrite rules in $\mathcal{R}$. This order should ensure more specialized rewrite rules are bigger, i.e. if $\Delta \ltimes p \rightarrow u$ and $\Delta \ltimes q \rightarrow v$ are two rewrite rules in $\mathcal{R}$ and $q = p\sigma$ then we should have $q \succ p$. Otherwise the choice of order is arbitrary (e.g. the order in which the rules were declared).*
- *Following that, we check for each symbol f declared in $\mathcal{R}$ that all rewrite rules have the same arity, i.e. the pattern stacks of the rewrite rules with head symbol f all have the same length.*
- *Next, we check that the set of left-hand sides is closed under unification by computing for each pair of patterns $\Delta \ltimes p \rightarrow u$ and $\Delta' \ltimes q \rightarrow v$ in $\mathcal{R}$ the most general unifier*[5] *$\sigma = MGU(p, q)$ and (if it exists) checking that $p\sigma$ is a left-hand side of a rewrite rule in $\mathcal{R}$ (up to α-conversion). If there is no such rule, we extend $\mathcal{R}$ with the auxiliary rewrite rule $\Delta'' \ltimes p\sigma \rightarrow u\sigma$ (where Δ'' is the context of pattern variables of $p\sigma$).*
- *Finally, for each rewrite rule $\Delta \ltimes p \rightarrow r$ in $\mathcal{R}$ we exhaustively enumerate all possible reduction steps $\mathcal{R} \vdash p \Rrightarrow w$ and check that there is a subsequent step $\mathcal{R} \vdash w \Rrightarrow r$.*

The extra rules added by the above algorithm are always instances of existing rewrite rules and thus need not be considered by the reduction algorithm; they are only required while checking the triangle property.

Example 7.5 (Revisiting Example 4.3). Consider the rewrite signature $\mathcal{R}$ with symbols 0, suc, and +, and four rewrite rules (1) $0 + ?n \rightarrow n$, (2) $?m + 0 \rightarrow m$, (3) $(\mathsf{suc}\ ?m) + ?n \rightarrow \mathsf{suc}\ (m + n)$, and (4) $?m + \mathsf{suc}\ ?n \rightarrow \mathsf{suc}\ (m + n)$. Here the set of left-hand sides is not closed under unification, so the algorithm extends it with four more rules (5) $0 + 0 \rightarrow 0$, (6) $0 + (\mathsf{suc}\ ?n) \rightarrow \mathsf{suc}\ n$, (7) $\mathsf{suc}\ ?m + 0 \rightarrow \mathsf{suc}\ m$, and (8) $(\mathsf{suc}\ ?m) + (\mathsf{suc}\ ?n) \rightarrow \mathsf{suc}\ (m + (\mathsf{suc}\ n))$. However, now the third step of the algorithm fails since $\mathcal{R} \vdash (\mathsf{suc}\ m) + (\mathsf{suc}\ n) \Rrightarrow \mathsf{suc}\ ((\mathsf{suc}\ m) + n)$ but $\mathcal{R} \nvdash \mathsf{suc}\ ((\mathsf{suc}\ m) + n) \Rrightarrow \mathsf{suc}\ (m + (\mathsf{suc}\ n))$.

To ensure this rewrite system is accepted, it has to be extended with the additional rule $(\mathsf{suc}\ ?m) + (\mathsf{suc}\ ?n) \rightarrow \mathsf{suc}\ (\mathsf{suc}\ (m+n))$. With this extra rule, we have $\rho_{\succ}((\mathsf{suc}\ m)+(\mathsf{suc}\ n)) = \mathsf{suc}\ (\mathsf{suc}\ (m+n))$ so we have both $\mathcal{R} \vdash \mathsf{suc}\ (m+(\mathsf{suc}\ n) \Rrightarrow \mathsf{suc}\ (\mathsf{suc}\ (m+n))$ and $\mathcal{R} \vdash \mathsf{suc}\ ((\mathsf{suc}\ m)+n) \Rrightarrow \mathsf{suc}\ (\mathsf{suc}\ (m+n))$, thus making the final check succeed.

[5]Since our patterns are first-order, unification is decidable.

It would be possible to improve the algorithm by extending step 3 to compute the normal form $w = NF(p\sigma)$ and extend the rewrite system with the new rule $p\sigma \twoheadrightarrow w$, turning the algorithm from a mere *check* into a full *completion algorithm*. This would make the parallel definition of + be accepted without adding the extra rule for $\mathsf{suc}\, m + \mathsf{suc}\, n$. However, it would also introduce new risks of non-termination. Hence we refrain from adding this step to the algorithm for now.

The requirement that all rewrite rules for a symbol have the same arity is more restrictive than strictly necessary. For example, we could allow rules with different arities if they clearly do not overlap, such as $f\ 0 \twoheadrightarrow \lambda\,(x : A).\ a$ and $f\ (\mathsf{suc}\ n)\ y \twoheadrightarrow b$. However, this does not seem particularly useful in our current setting so we opted to keep the criterion of Theorem 7.3 as simple as possible.

For the final step of the algorithm, one may wonder whether it is possible to check instead that $\rho_C(w) = r$. The following example shows that this is not the case in general.

Example 7.6. Consider the rewrite system $\mathcal{R}$ with four symbols a, b, c, d, and rewrite rules $a \twoheadrightarrow c$, $a \twoheadrightarrow b$, $b \twoheadrightarrow d$, $b \twoheadrightarrow c$, and $c \twoheadrightarrow d$. This system satisfies the triangle criterion with $\rho(a) = c$, $\rho(b) = d$, $\rho(c) = d$, and $\rho(d) = d$. We have $\mathcal{R} \vdash a \Rrightarrow b$ and $\mathcal{R} \vdash b \Rrightarrow c = \rho(a)$, but $\rho(b) = d \neq c$.

From this example, we conclude that it is not a good idea to replace the check that $\mathcal{R} \vdash w \Rrightarrow r$ with a check that $\rho_C(w) = r$, as this would lead to the check to fail where it should not.

7.2 Rewrite Rules Matching on Symbols

The type theory formalized in this paper has a strict separation between symbols that act as the head symbol of a rewrite rule and those acting as rigid symbols in a pattern. However, in the Agda implementation there is no such separation: a symbol can serve both roles in different rules or even in the same one. For example, one may make f into an idempotent operation by declaring a rewrite rule $f\ (f\ x) \twoheadrightarrow f\ x$.

In this more general setting, to check the triangle criterion for a set of rewrite rules $\mathcal{R}$, it is not sufficient anymore to check if two left-hand sides overlap at the root position: we also need to consider when one pattern overlaps with a subpattern of another pattern. To make this notion more formal, we introduce the notion of a *position* in a pattern. Basically, a position is a path in the syntax tree of a pattern. More formally, a position l is a finite sequence of natural numbers. The subpattern (or sub-frame) at a given position in a pattern, written $\boxed{p|_l}$, is defined as follows:

$$\begin{array}{lcl|lcl} p|_\bullet & := & p & \mathbb{F}, \mathbb{S}|_{0::l} & := & \mathbb{F}|_l \\ \langle f \mid \mathbb{S}\rangle|_{0::l} & := & \mathbb{S}|_l & \mathbb{F}, \mathbb{S}|_{1::l} & := & \mathbb{S}|_l \\ (\mathsf{suc}\ p)|_{0::l} & := & p|_l & (\diamond\ p)|_{0::l} & := & p|_l \\ (p, q)|_{0::l} & := & p|_l & (\mathsf{elim}\ P\ P_0\ P_S\ \diamond)|_{0::l} & := & P|_l \\ (p, q)|_{1::l} & := & q|_l & (\mathsf{elim}\ P\ P_0\ P_S\ \diamond)|_{1::l} & := & P_0|_l \\ & & & (\mathsf{elim}\ P\ P_0\ P_S\ \diamond)|_{2::l} & := & P_S|_l \end{array}$$

We define replacement at a given position $\boxed{p[q]_l}$ meaning p where the subpattern at position l is replaced by the pattern q. We then update Definition 7.2 as follows:

Definition 7.7 (Updated version of Definition 7.2). Let $(C, \succ)$ be an ordered set of patterns. We say that $(C, \succ)$ is *closed under unification* if for every pair of patterns $p, q \in C$ and every position l such that $p|_l$ unifies with q with most general unifier σ, we also have $p\sigma \in C$, and $p\sigma \succ p$.

With this updated definition of closure under unification, the second step of the algorithm also has to be updated. Now we compute for every pair of rules $\Delta \ltimes p \twoheadrightarrow u$ and $\Delta' \ltimes q \twoheadrightarrow v$ in $\mathcal{R}$ *and for every position l in p* the most general unifier $\sigma = MGU(p|_l, q)$ and (if it exists) check that $p\sigma$ is the left-hand side of another rule. With this change, our algorithm for checking the triangle criterion can again be used to enforce that the triangle property holds. However, a full formal proof of correctness as we did in the simpler setting is beyond the scope of this paper.

In general, adding auxiliary rules to $\mathcal{R}$ in the second step of the algorithm may now fail to terminate. For example, consider a rewrite system with two symbols s and p and two rewrite rules $s\,(p\,x) \rightarrowtriangle x$ and $p\,(s\,x) \rightarrowtriangle x$. There is no finite set C that includes $s\,(p\,x)$ and $p\,(s\,x)$ that is closed under unification. Even if it does terminate, we then also have to pick right-hand sides for each of these patterns, for which there may be more than one choice. In the implementation for Agda, we avoid this potential for non-termination and non-determinism by refusing to automatically add new rewrite rules. Instead, Agda reports an error and ask the user to manually add a rewrite rule with left-hand side $p\sigma$ to resolve the ambiguity between the two rewrite rules.

One may ask if it is necessary to also consider non-maximal reduction steps $\mathcal{R} \vdash p \Rrightarrow w$ in the final step of the algorithm. The following example shows that it is indeed necessary in general.

Example 7.8. Consider a rewrite system $\mathcal{R}$ with symbols f, a, b, and c, and rewrite rules $f\ a\ a \rightarrowtriangle c$, $f\ b\ b \rightarrowtriangle c$, and $a \rightarrowtriangle b$. This rewrite system does not satisfy the triangle property because $\mathcal{R} \vdash f\ a\ a \Rrightarrow f\ a\ b$, but $\mathcal{R} \nvdash f\ a\ b \Rrightarrow c$. However, if we only considered the possible *maximal* reductions of $f\ a\ a$, we would only consider the case for $f\ b\ b$ and not for $f\ a\ b$ and $f\ b\ a$, which would lead us to wrongly conclude that the triangle property is satisfied.

7.3 Agda **Implementation**

We have implemented user-defined rewrite rules as an extension to the Agda language [Agda Development Team 2020]. With this extension, the user can turn a (global) equality proof or postulate of the form $p : \Gamma \to \mathsf{f}\,\bar{p} \equiv v$ (where $\equiv$ is Agda's built-in equality type) into a rewrite rule by means of a `{-# REWRITE` p `#-}` pragma. With this pragma, Agda automatically reduces every instance of $\mathsf{f}\,\bar{p}$ to the corresponding instance of v during reduction. For example, one can turn the standard definition of + into the parallel version by providing two proofs $(k : \mathbb{N}) \to k + \mathsf{zero} \equiv k$ and $(k\ l : \mathbb{N}) \to k + (\mathsf{suc}\ l) \equiv \mathsf{suc}\ (k + l)$ and using them as rewrite rules.

Since each rewrite rule must be proven (or postulated) to hold as a propositional equality, all the rewrite rules in Agda automatically have an instantiation (see Sect. 6.4). Thus rewrite rules on their own cannot break the consistency of Agda; all unproven assumptions are made explicit as postulate's.

To ensure subject reduction is preserved, the implementation provides an optional confluence check, that can be enabled with the `{-# OPTIONS --confluence-check #-}` pragma. When enabled, Agda checks that the declared rewrite rules satisfy the conditions of the triangle criterion (Definition 6.3).[6] For checking that the set of left-hand sides is closed under unification, we rely on Agda's existing unification algorithm that is used for constraint solving [Abel and Pientka 2011].

Although the type theory presented in this paper requires axioms and rewrite rules to be declared in blocks, the implementation in Agda is more flexible while ensuring that the correct grouping in blocks can be done behind the scenes. In a typical use case of rewrite rules, the user would first define a function using regular means (i.e. pattern matching), then prove certain equalities about the function, and finally register these equalities as rewrite rules. The confluence checker will then take into account both the 'native' rewrite rules implicit in the definition by pattern matching, and the explicitly declared rewrite rules.

The Agda typechecker always performs definitional equality checking by computing the weak-head normal form of both sides of the equation (unless they are syntactically equal), which we have not changed for our prototype implementation. This strategy has bad performance in some

[6]In Agda 2.6.1 the confluence check is based on the standard criterion of joinability of critical pairs that only ensures local confluence. In Agda 2.6.2 this has been replaced with the triangle criterion described in this paper.

cases, but this is a general limitation of Agda and not specific to our implementation. A Coq implementation of rewrite rules can take advantage of Coq's optimisation heuristics, and be more careful to only unfold definitions when required.

The implementation of rewrite rules in Agda goes beyond the type theory described in this paper in several ways [Cockx 2020]:

- As noted before, it allows the same symbol to be used as both the head symbol of a rewrite rule and a rigid symbol in the pattern of a rewrite rule.
- It allows *higher-order* rules that can match on λ-abstractions, Π-types, and bound variables.
- It allows *non-linear* patterns and a (restricted) kind of conditional rewrite rules.

Rewrite rules also interact with several other features of Agda such as η-equality for functions and record types, definitional proof irrelevance [Gilbert et al. 2019], metavariables, universe polymorphism, and parametrized modules. The presence of η-equality in particular means we can omit the requirement that all rewrite rules with the same head symbol have the same arity. Instead, the algorithm for checking that the set of left-hand sides is closed under unification uses η-expansion to ensure stacks have the same shape before attempting to unify them.

These features go beyond the scope of our current work, which focuses instead on a solid and formalized core type theory with rewrite rules. Although we have not yet proven that our results in this paper hold in the more general setting of Agda, we conjecture that the proofs carry over without fundamental issues if we restrict it to linear and non-conditional rewrite rules.

8 RELATED WORK

Compared to existing work on rewriting theory, the main novelty of our framework is the need to consider the interaction between the rewrite rules and the type checking algorithm, and in particular to prevent new rewrite rules from breaking type checking and subject reduction. In this section, we compare our work with other attempts at extending type theory with rewrite rules, and also discuss confluence checking of rewriting systems more generally.

8.1 Rewrite Rules in Type Theory

Combining rewrite systems and type systems stems from the work by Tannen [1988], extending simply typed lambda-calculus with higher-order rewrite rules. This idea was taken to dependent type theory by Barbanera et al. [1997]. They extend the Calculus of Constructions with first- and higher-order rewrite rules, provided the higher-order rules do not introduce any critical pairs. Their proof of subject reduction relies on first proving strong normalization (without relying on confluence). In particular, their proof that β-reduction is type-preserving is greatly complicated by the fact that they can not assume injectivity of Π-types.

Walukiewicz-Chrzaszcz [2003] (section 4.2) proves subject reduction for another variant of the Calculus of Constructions with a more general notion of higher-order rewrite rules. Their proof that β-reduction is type-preserving is essentially the same as the one by Barbanera et al. [1997]. In later work the authors also study completeness and consistency of this system [Walukiewicz-Chrzaszcz and Chrzaszcz 2006], and discuss how to extend Coq with rewrite rules [Chrzaszcz and Walukiewicz-Chrzaszcz 2007].

The Calculus of Algebraic Constructions [Blanqui 2005] is another extension of the Calculus of Constructions with a restricted form of higher-order rewrite rules. It also provides criteria for checking subject reduction and strong normalization. Once again, the proof that β-reduction is type-preserving is based on the same idea as Barbanera et al. [1997].

The Open Calculus of Constructions [Stehr 2005a,b] combines the Calculus of Constructions with conditional rewrite rules and membership equational logic. However, it is based on a set-theoretic semantics and can thus not be used in other settings. Regarding the question of subject reduction, the author notes the following: "Since computation with non-well-typed terms can be semantically justified, there is in particular no need for a syntactic notion of subject reduction in our treatment."

CoqMT and CoqMTU [Barras et al. 2011; Jouannaud and Strub 2017; Strub 2010] continue the work on adding (first-order) rewrite rules to the Calculus of Inductive Constructions and integrate it into the Coq proof assistant. In addition, they also handle non-directed equalities such as commutativity laws. The authors provide proofs of subject reduction, strong normalization, logical consistency, and dedicability of type-checking. The proof of subject reduction relies on the existence of a function **norm** that normalizes the algebraic parts of a term according to the theory under consideration. This means CoqMT only deals with decidable first-order theories. Unfortunately, the implementation (available at https://github.com/strub/coqmt) has not been integrated in Coq itself and is no longer maintained.

Dedukti [Assaf and Burel 2015; Boespflug et al. 2012; Cousineau and Dowek 2007; Saillard 2015] is an implementation of $\lambda\Pi$-modulo, a logical framework with dependent types and user-defined higher-order rewrite rules. In contrast, our work focuses on adding rewrite rules to an existing type theory and the interactions between rewrite rules and other features such as inductive types and a universe hierarchy. Dedukti also provides the possibility to use external tools for checking confluence and termination of rewrite rules. Our focus in this paper is complementary: our formalization gives a precise statement of what properties need to be checked to ensure subject reduction of the whole type theory.

Blanqui [2020] shows how to lift the usual restriction that both sides of a rewrite rule should be well-typed and have the same type, instead checking that well-typedness of the left-hand side of a rewrite rule implies well-typedness of the right-hand side. The algorithm first collects all the constraints that are required for the LHS to be well-typed, and uses Knuth-Bendix completion to turn these rules into a confluent rewrite system that is used for checking the RHS. Our theory already allows rewrite rules that satisfy this more general criterion, and we expect it could also be added to our implementation for Agda.

8.2 Confluence Checking

When dealing with potentially non-terminating rewrite systems, the simplest criterion for confluence is the complete absence of critical pairs, *i.e., non-ambiguity.* For left-linear rewrite systems, this criterion is sufficient to conclude confluence. This criterion is actually enough to prove confluence of several examples in this paper, in particular the ones in Sect. 2.2, Sect. 2.3, Sect. 2.4, and Sect. 2.5.

Another simple but useful confluence criterion is *strong confluence*, *i.e.,* each critical pair $v_1 \leftsquigarrow u \rightsquigarrow v_2$ can be joined as $v_1 \rightsquigarrow w \leftsquigarrow^\star v_2$ with a single step on the left (and symmetrically, also as $v_1 \rightsquigarrow^\star w \leftsquigarrow v_2$). This method is sufficient to prove confluence of the parallel plus in Sect. 2.1.

Apart from strong confluence, there are several more powerful confluence criteria for first-order rewrite systems, most prominently those based on the *decreasing diagrams method* [van Oostrom 1994a]. The main challenge for applying this method is the need to *order* the rewrite rules in some well-founded way (see also the work by Felgenhauer and van Oostrom [2013] and Liu [2016]).

A more permissive version of this criterion is that the rewrite system should be *development closed* [van Oostrom 1995]. This criterion requires that each critical pair (v_1, v_2) can be joined as $v_1 \twoheadrightarrow v_2$, where a *development step* $\boxed{u \twoheadrightarrow v}$ is defined as the congruence closure of the rule

$$\frac{\forall x.\ x\sigma \twoheadrightarrow x\sigma'}{l\sigma \downarrow_\beta \twoheadrightarrow r\sigma' \downarrow_\beta}$$

where $l \rightarrowtriangle r$ is a rewrite rule and $\downarrow_\beta$ denotes β-normalization.

The recent work by Ferey and Jouannaud [2019] on checking confluence of higher-order rewrite systems is based on the decreasing diagrams approach by van Oostrom [1994a] and applies it to the Dedukti system. Like our approach, the authors focus on first proving confluence for the untyped system and using this to prove type preservation and strong normalization. However, the main theorem requires the rewrite rules to be terminating on the set of pure λ-terms, while our criterion does not require any form of termination.

Regarding modularity of confluence, the main result is the one by Van Oostrom [van Oostrom 1994b, Theorem 3.1.54]. He shows that combining two left-linear higher-order rewrite systems is confluent if the systems are individually confluent and weakly orthogonal (*i.e.,* all critical pairs between them are trivial). However, as we mentioned before this result does not apply directly to our setting because of the presence of inductive types.

9 EXTENSIONS AND FUTURE WORK

Checking Termination. To ensure that our type theory enjoys not just subject reduction but also decidable type checking, we would like to check termination of rewrite rules. There are several existing techniques that could be applied to our setting. Barbanera et al. [1997] apply the general schema by Jouannaud and Okada [1991] to a dependently typed language. Later, it is also used by Blanqui [2005] to prove termination of rewrite rules in the Calculus of Algebraic Constructions. The Higher-order recursive path ordering (HORPO) by Jouannaud and Rubio [1999] is used by Walukiewicz-Chrzaszcz [2003] to check termination of rewrite rules in type theory. The computability path ordering (CPO) [Blanqui et al. 2008], an improved version of HORPO, is a general and purely syntactic approach to proving termination. It has been extended to handle dependent types by Jouannaud and Li [2015]. The size-change termination principle [Lee et al. 2001] has been extended to dependently typed rewriting systems by Blanqui [2004, 2018] and has recently been implemented in the SizeChangeTool [Genestier 2019] to check termination of rewrite rules in Dedukti.

As for confluence, termination is not just a property of the rewrite rules themselves but of the whole system, including inductive types and the universe hierarchy. Thus the main challenge in proving termination is to integrate a specific criterion for termination of rewrite rules (e.g. size-change termination) with a general proof of strong normalisation of type theory. One such proof of strong normalization is given by Abel et al. [2018]. A possible direction forward would be to extend this proof to a type theory with rewrite rules.

Allowing more General Patterns. In this paper we extend PCUIC with first-order rewrite rules with linear patterns and a strict separation between function-like symbols and constructor-like symbols. We could extend the pattern language in several ways to allow more general rewrite rules. We can allow higher-order patterns that match against binders such as Π-types and λ-terms. Inside the body of a binder, bound variables can also act as rigid symbols that can be matched against, enabling rewrite rules such as map $(\lambda\,(x : A).\, x)\; l \rightarrowtriangle l$. In addition, we could allow pattern variables inside a binder to be applied to a subset of all bound variables rather than all of them. For example, this would allow the rule subst $(\lambda\,(\_ : \square).\, A)\; p\; x \rightarrowtriangle x$. However, this requires checking freeness of variables during matching. We could allow non-linear patterns by checking conversion during matching. However, this would mean confluence is no longer modular in general, so we would need a new criterion. All of these extensions to the pattern syntax have been implemented in our implementation for Agda, so it would be interesting to also support them in our formalization.

Combining Rewrite Rules with a Type-Directed Equational Theory. Many type theories include equational laws other than β- and ι reduction in their notion of conversion. For example, Agda includes at least the following equational laws: (i) η-equality for functions and record types. (ii)

Proof-irrelevance for types in the Prop universe [Gilbert et al. 2019]. (iii) Equational laws for universe levels such as $m \sqcup n = n \sqcup m$ and lsuc $(m \sqcup n) =$ lsuc $m \sqcup$ lsuc n. These equational laws are taken into account by our implementation in Agda, but they are not yet considered in our formalization. Adding η-equality for functions to MetaCoq is currently work in progress.

Abstracting over Rewrite Rules. An important limitation of our current work is that it is possible to introduce new computational assumptions, but not to discharge them (except on the meta-level as we do in Sect. 6.4). In theory, it should be possible to design a language construct that abstracts over a rewrite rule in the context, similarly to how a lambda abstracts over a value. Such a construct that allows abstraction over computational assumptions and discharging those assumptions when they hold definitionally would have interesting applications when working with abstract structures in dependent type theory. However, doing this naively leads to several problems, in particular it leads to a loss of decidable type checking since "being a linear pattern" is not stable under substitution.

A better approach would be to not abstract over individual rewrite rules but instead abstract simultaneously over a set of values and a set of rewrite rules on these values. This leads us to a notion of 'rewrite record type' consisting of a number of fields plus a number of rewrite rules with these fields as symbols. To construct an element of a rewrite record type, the given fields should satisfy all the rewrite rules definitionally. This solves the problem from before, but there is still the question whether the resulting system satisfies confluence. There is now a critical pair between each rewrite rule and the primitive computation rule for the projection corresponding to the head symbol. Moreover, checking confluence is complicated by the fact that most rewrite rules in this approach are non-linear in the value of the record type itself. How to overcome these problems and provide a practical notion of a 'rewrite record type' is something we would like to work on in the future.

10 CONCLUSION

Since its very beginning, the evolution of dependent type theory has swayed back and forth between the practical decidability of intensional type theory and the expressive power of extensional type theory. Adding rewrite rules to type theory combines some of the strengths of both into a single system. This has lead to a long history of attempts at integrating rewrite rules into type theory, each one with its own subtly different aims, restrictions, and guarantees. Yet despite the great promise of rewriting in type theory and major breakthroughs in the field, the fact remains that the practical use of rewrite rules in type-theoretic proof assistants at the time of writing is still limited. A likely explanation is that both dependent type theory and rewriting theory by themselves are complex theories that are difficult to implement correctly, so their combination pushes the complexity beyond what can be reasonably verified by hand.

This shows the need for a type theory with rewrite rules that is rigorously formalized yet powerful enough to serve as the core for a real proof assistant, which is what we do in this paper. Compared to previous work, we do not have the most general possible notion of rewriting or the most powerful criteria for checking confluence and termination. However, what we do provide is (i) a combination of Coq's core calculus PCUIC with general user-defined rewrite rules, (ii) a formalization of this theory showing important metatheoretical properties, in particular confluence and subject reduction, and (iii) an implementation of rewrite rules in Agda that combines rewrite rules with all of Agda's other complex features and that includes a practical criterion for ensuring metatheoretical properties hold.

We believe our work raises the bar for future research on the areas of both formalization and practically usable implementation, and will ultimately lead to the power of rewrite rules being widely usable in our dependently typed programming languages and proof assistants.

REFERENCES

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of Conversion for Type Theory in Type Theory. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 23 (Jan. 2018), 29 pages. https://doi.org/10.1145/3158111

Andreas Abel and Brigitte Pientka. 2011. Higher-Order Dynamic Pattern Unification for Dependent Types and Records. In *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings (Lecture Notes in Computer Science)*, C.-H. Luke Ong (Ed.), Vol. 6690. Springer, 10–26. https://doi.org/10.1007/978-3-642-21691-6_5

Agda Development Team. 2020. *Agda 2.6.1 documentation*. http://agda.readthedocs.io/en/v2.6.1/

Guillaume Allais, Conor McBride, and Pierre Boutillier. 2013. New Equations for Neutral Terms: A Sound and Complete Decision Procedure, Formalized. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming (DTP '13)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2502409.2502411

Claus Appel, Vincent van Oostrom, and Jakob Grue Simonsen. 2010. Higher-Order (Non-)Modularity. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK (LIPIcs)*, Christopher Lynch (Ed.), Vol. 6. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 17–32. https://doi.org/10.4230/LIPIcs.RTA.2010.17

Ali Assaf and Guillaume Burel. 2015. Translating HOL to Dedukti. *Electronic Proceedings in Theoretical Computer Science* 186 (Jul 2015), 74–88. https://doi.org/10.4204/EPTCS.186.8

Franco Barbanera, Maribel Fernández, and Herman Geuvers. 1997. Modularity of Strong Normalization in the Algebraic-lambda-Cube. *Journal of Functional Programming* 7, 6 (1997), 613–660.

Henk Barendregt. 1984. *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, Vol. 103. Elsevier. https://doi.org/10.1016/B978-0-444-87508-2.50001-0

Bruno Barras, Jean-Pierre Jouannaud, Pierre-Yves Strub, and Qian Wang. 2011. CoQMTU: A Higher-Order Type Theory with a Predicative Hierarchy of Universes Parametrized by a Decidable First-Order Theory. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. IEEE Computer Society, 143–151. https://doi.org/10.1109/LICS.2011.37

Ulrich Berger, Ralph Matthes, and Anton Setzer. 2019. Martin Hofmann's Case for Non-Strictly Positive Data Types. In *24th International Conference on Types for Proofs and Programs (TYPES 2018)*, Peter Dybjer, José Espírito Santo, and Luís Pinto (Eds.). Leibniz International Proceedings in Informatics (LIPIcs), Vol. 130. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 1:1–1:22. https://doi.org/10.4230/LIPIcs.TYPES.2018.1

Frédéric Blanqui. 2004. A Type-Based Termination Criterion for Dependently-Typed Higher-Order Rewrite Systems. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings (Lecture Notes in Computer Science)*, Vincent van Oostrom (Ed.), Vol. 3091. Springer, 24–39. http://springerlink.metapress.com/openurl.asp?genre=article&amp;issn=0302-9743&amp;volume=3091&amp;spage=24

Frédéric Blanqui. 2005. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science* 15, 1 (2005), 37–92. https://doi.org/10.1017/S0960129504004426

Frédéric Blanqui. 2018. Size-based termination of higher-order rewriting. *Journal of Functional Programming* 28 (April 2018), e11. https://doi.org/10.1017/S0956796818000072

Frédéric Blanqui. 2020. Type Safety of Rewrite Rules in Dependent Types. In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference) (LIPIcs)*, Zena M. Ariola (Ed.), Vol. 167. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:14. https://doi.org/10.4230/LIPIcs.FSCD.2020.13

Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. 2008. The Computability Path Ordering: The End of a Quest. In *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings (Lecture Notes in Computer Science)*, Michael Kaminski and Simone Martini (Eds.), Vol. 5213. Springer, 1–14. https://doi.org/10.1007/978-3-540-87531-4_1

Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. 2012. The $\lambda\Pi$-calculus Modulo as a Universal Proof Language. In *the Second International Workshop on Proof Exchange for Theorem Proving (PxTP 2012)*, Tjark Weber David Pichardie (Ed.), Vol. Vol. 878. Manchester, United Kingdom, pp. 28–43. https://hal-mines-paristech.archives-ouvertes.fr/hal-00917845

Jacek Chrzaszcz and Daria Walukiewicz-Chrzaszcz. 2007. Towards Rewriting in Coq. In *Rewriting, Computation and Proof, Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Hubert Comon-Lundh, Claude Kirchner, and Hélène Kirchner (Eds.), Vol. 4600. Springer, 113–131. https://doi.org/10.1007/978-3-540-73147-4_6

Jesper Cockx. 2020. Type theory unchained: Extending Agda with user-defined rewrite rules (system description). (2020). To appear in the post-proceedings of the TYPES conference.

The Coq Development Team. 2016. *The* Coq *proof assistant reference manual*. http://coq.inria.fr Version 8.6.

Thierry Coquand and Christine Paulin. 1988. Inductively Defined Types. In *Proceedings of the International Conference on Computer Logic (COLOG '88)*. Springer-Verlag, Berlin, Heidelberg, 50–66.

Denis Cousineau and Gilles Dowek. 2007. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings (Lecture Notes in Computer Science)*, Simona Ronchi Della Rocca (Ed.), Vol. 4583. Springer, 102–117. https://doi.org/10.1007/978-3-540-73228-0_9

Bertram Felgenhauer and Vincent van Oostrom. 2013. Proof Orders for Decreasing Diagrams. In *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands (LIPIcs)*, Femke van Raamsdonk (Ed.), Vol. 21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 174–189. https://doi.org/10.4230/LIPIcs.RTA.2013.174

Gaspard Ferey and Jean-Pierre Jouannaud. 2019. Confluence in (Un)Typed Higher-Order Theories by means of Critical Pairs. (Dec. 2019). https://hal.archives-ouvertes.fr/hal-02096540 (under submission).

Guillaume Genestier. 2019. SizeChangeTool: A Termination Checker for Rewriting Dependent Types. In *HOR 2019 - 10th International Workshop on Higher-Order Rewriting (Joint Proceedings of HOR 2019 and IWC 2019)*, Mauricio Ayala-Rincón, Silvia Ghilezan, and Jakob Grue Simonsen (Eds.). Dortmund, Germany, 14–19. https://hal.archives-ouvertes.fr/hal-02442465

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages* 3 (Jan. 2019), 1–28. https://doi.org/10.1145/329031610.1145/3290316

Martin Hofmann. 1993. Non Strictly Positive Datatypes in System F. Email on types mailing list. http://www.seas.upenn.edu/ sweirich/types/archive/1993/msg00027.html.

Jean-Pierre Jouannaud and Jianqi Li. 2015. Termination of Dependently Typed Rewrite Rules. In *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland (LIPIcs)*, Thorsten Altenkirch (Ed.), Vol. 38. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 257–272. https://doi.org/10.4230/LIPIcs.TLCA.2015.257

Jean-Pierre Jouannaud and Mitsuhiro Okada. 1991. A Computation Model for Executable Higher-Order Algebraic Specification Languages. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science, 15-18 July, 1991, Amsterdam, The Netherlands*. IEEE Computer Society, 350–361. https://doi.org/10.1109/LICS.1991.151659

Jean-Pierre Jouannaud and Albert Rubio. 1999. The Higher-Order Recursive Path Ordering. In *Proceedings, 14th Annual IEEE Symposium on Logic in Computer Science, 2-5 July, 1999, Trento, Italy*. IEEE Computer Society, 402–411. https://doi.org/10.1109/LICS.1999.782635

Jean-Pierre Jouannaud and Pierre-Yves Strub. 2017. Coq without Type Casts: A Complete Proof of Coq Modulo Theory. In *LPAR-21: 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. EasyChair, Maun, Botswana, 474–489. https://hal.inria.fr/hal-01664457

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, 81–92. https://doi.org/10.1145/360204.360210

Jiaxiang Liu. 2016. *Confluence properties of rewrite rules by decreasing diagrams. (Propriétés de confluence des règles de réécriture par des diagrammes décroissants)*. Ph.D. Dissertation. University of Paris-Saclay, France. https://tel.archives-ouvertes.fr/tel-01515698

Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 73 – 118. https://doi.org/10.1016/S0049-237X(08)71945-1

Christine Paulin-Mohring. 2015. Introduction to the Calculus of Inductive Constructions. In *All about Proofs, Proofs for All*, Bruno Woltzenlogel Paleo and David Delahaye (Eds.). Studies in Logic (Mathematical logic and foundations), Vol. 55. College Publications. https://hal.inria.fr/hal-01094195

Pierre-Marie Pédrot and Nicolas Tabareau. 2018. Failure is Not an Option: An Exceptional Type Theory. In *ESOP 2018 - 27th European Symposium on Programming (Lecture Notes in Computer Science)*, Vol. 10801. Springer, Thessaloniki, Greece, 245–271. https://doi.org/10.1007/978-3-319-89884-1_9

Pierre-Marie Pédrot, Nicolas Tabareau, Hans Jacob Fehrmann, and Éric Tanter. 2019. A Reasonably Exceptional Type Theory. *Proceedings of the ACM on Programming Languages* 3, ICFP, Article 108 (July 2019), 29 pages. https://doi.org/10.1145/3341712

Ronan Saillard. 2015. *Typechecking in the lambda-Pi-Calculus Modulo: Theory and Practice*. Ph.D. Dissertation. MINES ParisTech.

Gert Smolka. 2015. Confluence and Normalization in Reduction Systems. (2015). https://www.ps.uni-saarland.de/courses/sem-ws15/ars.pdf (lecture notes).

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

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 8 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371076

Mark-Oliver Stehr. 2005a. The Open Calculus of Constructions (Part I): An Equational Type Theory with Dependent Types for Programming, Specification, and Interactive Theorem Proving. *Fundamenta Informaticae* 68, 1-2 (2005), 131–174.

Mark-Oliver Stehr. 2005b. The Open Calculus of Constructions (Part II): An Equational Type Theory with Dependent Types for Programming, Specification, and Interactive Theorem Proving. *Fundamenta Informaticae* 68, 3 (2005), 249–288.

Pierre-Yves Strub. 2010. Coq Modulo Theory. In *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings (Lecture Notes in Computer Science)*, Anuj Dawar and Helmut Veith (Eds.), Vol. 6247. Springer, 529–543. https://doi.org/10.1007/978-3-642-15205-4_40

M. Takahashi. 1995. Parallel Reductions in λ-Calculus. *Information and Computation* 118, 1 (1995), 120 – 127. https://doi.org/10.1006/inco.1995.1057

Val Tannen. 1988. Combining Algebra and Higher-Order Types. In *Proceedings, Third Annual Symposium on Logic in Computer Science, 5-8 July 1988, Edinburgh, Scotland, UK*. IEEE Computer Society, 82–90. https://doi.org/10.1109/LICS.1988.5103

Terese. 2003. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science, Vol. 55. Cambridge University Press.

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study.

Vincent van Oostrom. 1994a. Confluence by Decreasing Diagrams. *Theoretical Computer Science* 126, 2 (1994), 259–280. https://doi.org/10.1016/0304-3975(92)00023-K

Vincent van Oostrom. 1994b. *Confluence for abstract and higher-order rewriting*. Ph.D. Dissertation. Vrije Universiteit Amsterdam.

Vincent van Oostrom. 1995. Development Closed Critical Pairs. In *Higher-Order Algebra, Logic, and Term Rewriting, Second International Workshop, HOA '95, Paderborn, Germany, September 21-22, 1995, Selected Papers (Lecture Notes in Computer Science)*, Gilles Dowek, Jan Heering, Karl Meinke, and Bernhard Möller (Eds.), Vol. 1074. Springer, 185–200. https://doi.org/10.1007/3-540-61254-8_26

Daria Walukiewicz-Chrzaszcz. 2003. Termination of rewriting in the Calculus of Constructions. *Journal of Functional Programming* 13, 2 (2003), 339–414. https://doi.org/10.1017/S0956796802004641

Daria Walukiewicz-Chrzaszcz and Jacek Chrzaszcz. 2006. Consistency and Completeness of Rewriting in the Calculus of Constructions. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science)*, Ulrich Furbach and Natarajan Shankar (Eds.), Vol. 4130. Springer, 619–631. https://doi.org/10.1007/11814771_50