Consistency and Completeness of Rewriting in the Calculus of Constructions

Daria Walukiewicz-Chrząszcz and Jacek Chrząszcz*

Institute of Informatics, Warsaw University
ul. Banacha 2, 02-097 Warsaw, Poland

{daria,chrzaszcz}@mimuw.edu.pl

Abstract. Adding rewriting to a proof assistant based on the Curry-Howard isomorphism, such as Coq, may greatly improve usability of the tool. Unfortunately adding an arbitrary set of rewrite rules may render the underlying formal system undecidable and inconsistent. While ways to ensure termination and confluence, and hence decidability of type-checking, have already been studied to some extent, logical consistency has got little attention so far.

In this paper we show that consistency is a consequence of canonicity, which in turn follows from the assumption that all functions defined by rewrite rules are complete. We provide a sound and terminating, but necessarily incomplete algorithm to verify this property. The algorithm accepts all definitions that follow dependent pattern matching schemes presented by Coquand and studied by McBride in his PhD thesis. Moreover, many definitions by rewriting containing rules which depart from standard pattern matching are also accepted.

1 Introduction

Equality is ubiquitous in mathematics. Yet it turns out that proof assistants based on the Curry-Howard isomorphism, such as Coq [9] are not very good at handling equality. While proving an equality is not a problem in itself, using already established equalities is quite problematic.

Apart from equalities resulting from internal reductions (beta, iota), which can be used via the conversion rule without being recorded in the proof term, any other use of an equality requires giving all details about the context explicitly in the proof. As a result, proof terms may become extremely large, taking up memory and making type-checking time consuming: working with equations in Coq is not very convenient.

A straightforward idea for reducing the size of proof terms is to allow other equalities in the conversion, making their use transparent. This can be done by using user-defined *rewrite rules*. However, adding arbitrary rules may easily lead to logical inconsistency, making the proof environment useless. It is of

* This work was partly supported by KBN grant 3 T11C 002 27.

course possible to put the responsibility on the user, but it is contrary to the current Coq policy to guarantee consistency of developments without axioms. It is desirable to retain this guarantee when rewriting is added to Coq. Since consistency is undecidable in presence of rewriting, one has find some decidable criteria satisfied only by rewriting systems which do not violate consistency.

The syntactical proof of consistency of the calculus of constructions, which is the base of the formalism implemented in Coq, requires every term to have a normal form [2]. The same proof is also valid for the calculus of inductive constructions [22], which is even closer to the formalism implemented in Coq.

There exist several techniques to prove (strong) normalization of the calculus of constructions with rewriting [1, 5, 4, 19, 20], following numerous works about rewriting in the simply-typed lambda calculus. Practical criteria for ensuring other fundamental properties, like confluence, subject reduction and decidability of type-checking are addressed e.g. in [4].

Logical consistency is also studied in [4]. It is shown under an assumption that for every symbol f defined by rewriting, $f(t_1, \ldots, t_n)$ is reducible if $t_1 \ldots t_n$ are terms in normal form in the environment consisting of one type variable. Apart from an informal proof that this is the case for the two rules defining the induction predicate for natural numbers and a remark that this property resembles the completeness of definitions, satisfying the assumption of the consistency lemma is not discussed.

Practical techniques for checking completeness of definitions are known for almost 30 years for the first-order algebraic setting [12, 18, 13]. More recently, their adaptations to type theory appeared in [10, 14] and [16]. In this paper we show how the latter algorithm can be tailored to the calculus of constructions extended with rewriting. We study a system, similar to a proof assistant reality, where the sets of available function symbols and rewrite rules are not known from the beginning but may grow as the proof development advances.

We show that logical consistency is an easy consequence of canonicity (see Def. 2 and Lemma 4), which in turn can be proved from completeness of definitions by rewriting, provided that termination and confluence are proved first. Our completeness checking algorithm closes the list of necessary procedures needed to guarantee logical consistency of developments in a proof assistant based on the calculus of constructions with rewriting. For additional examples and the proofs not included in [21], we refer the reader to the appendix.

2 Rewriting in the Calculus of Constructions

Let us briefly discuss how we imagine introducing rewriting in Coq and what problems we encounter on the way to a usable system.

From the user's perspective definitions by rewriting could be entered just as all other definitions:[1]

```
Inductive  nat : Set  :=  O : nat | S : nat → nat.
Symbol + : nat → nat → nat
Rules
  O + y   ⟶   y                      x + O   ⟶   x
  (S x) + y ⟶ S (x + y)              x + (S y) ⟶ S (x + y)
  x + (y + z) ⟶ (x + y) + z.
Parameter n : nat.
```

The above fragment can be interpreted as an environment E consisting of the inductive definition of natural numbers, symmetric definition by rewriting of addition and the declaration of a variable n of type *nat*. In this environment all rules for + contribute to conversion. For instance both $\forall x.\ x + 0 = x$ and $\forall x.\ 0 + x = x$ can be proved by $\lambda x : \mathit{nat}.\ \mathit{refl}\ \mathit{nat}\ x$, where *refl* is the only constructor of the Leibniz equality inductive predicate. Note that the definition of + is terminating and confluent. The latter can be checked by an (automatic) examination of its critical pairs.

Rewrite rules can also be used to define higher-order and polymorphic functions, like the `map` function on polymorphic lists.

```
Symbol map : forall (A:Set), (A → A) → list A → list A
Rules
  map A f (nil A) ⟶ nil A
  map A f (cons A a l) ⟶ cons A (f a) (map A f l)
  map A λx.x l ⟶ l
```

Even though we consider higher-order rewriting, we choose the simple matching modulo α-conversion. Higher-order matching is useful for example to encode logical languages by higher-order abstract syntax, but it is seldom used in Coq where modeling relies rather on inductive types. Instead of higher-order matching, one needs a possibility not to specify certain arguments in left-hand sides, and hence to work with rewrite rules built from terms that may be not typable. Consider, for example the type `tree` of trees with size and the function `rotr` performing a right rotation in the root of the tree.

```
Inductive tree : nat → Set :=
  Leaf : tree O
| Node : forall n1:nat, tree n1 → bool → forall n2:nat, tree n2
            → tree (S(n1+n2)).
```

[1] The syntax of the definition by rewriting is inspired by the experimental "Rewriting" branch of Coq developed by Blanqui. For the sake of clarity we omit certain details, like environments of rule variables and allow the infix + in the definition.

```
Symbol rotr : forall n:nat, (tree n) → (tree n)
Rules
   rotr 0 t  ⟶  t
   rotr ? (Node O t1 a n2 t2)  ⟶  Node O t1 a n2 t2
   rotr ?1 (Node ?2 (Node ?3 A b ?4 C) d ?5 E)
            ⟶  Node ?3 A b (S (?4 + ?5))(Node ?4 C d ?5 E)
```

The first argument of `rotr` is the size of the tree and the second is the tree itself. The first two rules cover the trees which cannot be rotated and the third one performs the rotation.

The ? marks above should be treated as different variables. The information they hide is redundant: if we take the third rule for example, the values of ?3, ?4 and ?5 must correspond to the sizes of the trees A, C and E respectively, ?2 must be equal S(?3+?4) and ?1=S(?2+?5). Note that by not writing suitable subterms we make the rule left-linear (and therefore easier to match) and avoid critical pairs with +, therefore helping the confluence proof.

This way of writing left-hand sides of rules was already used by Werner in [22] to define elimination rules for inductive types, making them orthogonal (the left-hand sides are of the form $I_{elim}\ P\ \vec{f}\ \vec{w}\ (c\ \vec{x})$, where P, $\vec{f}$, $\vec{w}$, $\vec{x}$ are distinct variables and c is a constructor of I). In [4], Blanqui gives a precise account of these omissions using them to make more rewriting rules left-linear. Later, the authors of [6] show that these redundant subterms can be completely removed from terms (in a calculus without rewriting however). In [3], a new optimized convertibility test algorithm is presented for Coq, which ignores testing equality of these redundant arguments.

In our paper we do not specify which arguments should/could be replaced by ? and do not restrict left-hand sides to be left-linear, but rely on the acceptance condition for definitions by rewriting to guarantee needed metatheoretical properties listed in the next section.

It is also interesting to note that when the first argument of `rotr` is a ? then we may say that it is matched modulo conversion and not syntactically.

3 Pure Type Systems with Generative Definitions

Even though most papers motivated by the development of Coq concentrate on the calculus of constructions, we present here a slightly more general formalization of a pure type system with inductive definitions and definitions by rewriting. The presentation, taken from [7, 8], is quite close to the way these elements are and could possibly be implemented in Coq. The formalism is built upon a set of PTS sorts $\mathcal{S}$, a binary relation $\mathcal{A}$ and a ternary relation $\mathcal{R}$ over $\mathcal{S}$ governing the typing rules **(Term/Ax)** and **(Term/Prod)** respectively (Fig. 1). The syntactic class of pseudoterms is defined as follows:

$$t ::= v \mid \mathrm{s} \mid (t_1\ t_2) \mid (\lambda v\!:\!t_1.t_2) \mid ((v\!:\!t_1)t_2)$$

A pseudoterm can be a variable, a sort from $\mathcal{S}$, an application, an abstraction or a product. We use Greek letters γ, δ to denote substitutions which are finite partial maps from variables to pseudo-terms. The postfix notation is used for the application of substitutions to terms.

Inductive definitions and definitions by rewriting are *generative*. They are stored in the environment and are used in terms only through names they "generate". An environment is a sequence of declarations, each of them is a variable declaration $v : t$, an inductive definition $\mathsf{Ind}(\Gamma^I := \Gamma^C)$, where Γ^I and Γ^C are environments of (possibly mutually defined) inductive types and their constructors, or a definition by rewriting $\mathsf{Rew}(\Gamma, R)$, where Γ is an environment of (possibly mutually defined) function symbols and R is a set of rewrite rules defining them. Given an environment E, inductive types, constructors and function symbols declared in E are called constants (even though syntactically they are variables). General environments are denoted by E and the environment containing only variable declarations are denoted by Γ, Δ, G, D. We assume that names of all declarations in environments are pairwise disjoint. A pair consisting of an environment E and a term e is called a sequent and denoted by $E \vdash e$. A sequent is well-typed if $E \vdash e : t$ for some t.

Definition 1. *A* pure type system with generative definitions *is defined by the typing rules in Fig. 1, where:*

- *The relation $\approx$ used in the rule* **(Term/Conv)** *is the smallest congruence on well typed terms, generated by $\longrightarrow$ which is the sum of beta $\longrightarrow_\beta$ and rewrite $\longrightarrow_R$ reductions.*
- *The notation $E \vdash \mathsf{Ind}(\Gamma^I := \Gamma^C) : \mathsf{correct}$ means that:*
  - *$\Gamma^I = I_1 : t_1^I \ldots I_k : t_k^I$ is the environment of inductive types*
  - *$\Gamma^C = c_1 : t_1^C \ldots c_n : t_n^C$ is the environment of inductive constructors*
  - *the positivity condition $\mathsf{POS}_E(\Gamma^I := \Gamma^C)$, as defined e.g. in [15], is satisfied.*
- *The notation $E \vdash \mathsf{Rew}(\Gamma, R) : \mathsf{correct}$ means that:*
  - *$\Gamma = f_1 : t_1 \ldots f_n : t_n$ is the environment of function symbols with $E \vdash t_i : \mathrm{s}_i$*
  - *$R = \{\Gamma_i \vdash l_i \longrightarrow r_i\}_{i=1\ldots n}$ is the set of rewrite rules, where Γ_i is the environment of variables (including ? variables), l_i is the left-hand side and r_i is the right-hand side*
  - *for each rule, $E; \Gamma_i \vdash \mathtt{ok}$*
  - *R satisfies the acceptance condition $\mathsf{ACC}_E(\Gamma; R)$ (see below).*
- *The notation $\delta : \Gamma \to E$ means that δ is a* well-typed substitution, *i.e. $E \vdash v\delta : t\delta$ for all $v : t \in \Gamma$.*

$$\frac{}{\epsilon \vdash \mathtt{ok}} \qquad \frac{E \vdash \mathtt{ok} \quad E \vdash t : \mathrm{s}}{E; v : t \vdash \mathtt{ok}}$$

$$\frac{E \vdash \mathtt{ok} \quad E \vdash \mathsf{Ind}(\Gamma^I := \Gamma^C) : \mathsf{correct}}{E; \mathsf{Ind}(\Gamma^I := \Gamma^C) \vdash \mathtt{ok}} \qquad \frac{E \vdash \mathtt{ok} \quad E \vdash \mathsf{Rew}(\Gamma, R) : \mathsf{correct}}{E; \mathsf{Rew}(\Gamma, R) \vdash \mathtt{ok}}$$

$$\frac{E_1; v : t; E_2 \vdash \mathtt{ok}}{E_1; v : t; E_2 \vdash v : t}$$

$$\frac{E \vdash \mathtt{ok}}{E \vdash I_i : t_i^I} \qquad \frac{E \vdash \mathtt{ok}}{E \vdash c_i : t_i^C} \qquad \text{where} \begin{cases} E = E_1; \mathsf{Ind}(\Gamma^I := \Gamma^C); E_2 \\ \Gamma^I = I_1 : t_1^I \ldots I_n : t_n^I \\ \Gamma^C = c_1 : t_1^C \ldots c_m : t_m^C \end{cases}$$

$$\frac{E \vdash \mathtt{ok}}{E \vdash f_i : t_i} \qquad \frac{E \vdash \mathtt{ok} \quad \delta : \Gamma_i \to E}{E \vdash l_i\delta \longrightarrow_R r_i\delta} \qquad \text{where} \begin{cases} E = E_1; \mathsf{Rew}(\Gamma, R); E_2 \\ \Gamma = f_1 : t_1 \ldots f_n : t_n \\ R = \{\Gamma_i : l_i \longrightarrow r_i\}_{i=1\ldots m} \end{cases}$$

(Term/Prod)
$$\frac{E \vdash t_1 : \mathrm{s}_1 \quad E; v : t_1 \vdash t_2 : \mathrm{s}_2}{E \vdash (v\!:\!t_1)t_2 : \mathrm{s}_3}$$
where $\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3 \in \mathcal{S}$

(Term/Abs)
$$\frac{E; v : t_1 \vdash e : t_2 \quad E \vdash (v\!:\!t_1)t_2 : \mathrm{s}}{E \vdash \lambda v\!:\!t_1.e : (v\!:\!t_1)t_2}$$

(Term/Ax)
$$\frac{E \vdash \mathtt{ok}}{E \vdash \mathrm{s}_1 : \mathrm{s}_2}$$
where $(\mathrm{s}_1, \mathrm{s}_2) \in \mathcal{A}$

(Term/App)
$$\frac{E \vdash e : (v\!:\!t_1)t_2 \quad E \vdash e' : t_1}{E \vdash e\ e' : t_2\{v \mapsto e'\}}$$

(Term/Conv)
$$\frac{E \vdash e : t \quad E \vdash t' : \mathrm{s} \quad E \vdash t \approx t'}{E \vdash e : t'}$$

Fig. 1. Environment correctness, environment lookup and PTS rules

As in [20, 4], recursors and their reduction rules have no special status and they are supposed to be expressed by rewriting.

Assumptions. We assume that we are given a positivity condition POS and an acceptance condition ACC for definitions by rewriting. Together with the right choice of the PTS they must imply the following properties:

P1) subject reduction, i.e. $E \vdash e : t$, $E \vdash e \longrightarrow e'$ implies $E \vdash e' : t$

P2) uniqueness of types, i.e. $E \vdash e : t$, $E \vdash e : t'$ implies $E \vdash t \approx t'$.

P3) strong normalization, i.e. $E \vdash \mathtt{ok}$ implies that reductions of all well-typed terms in E are finite

P4) confluence, i.e. $E \vdash e : t$, $E \vdash e \longrightarrow^* e'$, $E \vdash e \longrightarrow^* e''$ implies $E \vdash e' \longrightarrow^* \hat{e}$ and $E \vdash e'' \longrightarrow^* \hat{e}$ for some $\hat{e}$.

These properties are usually true in all well-behaved type theories. They are for example all proved for the calculus of algebraic constructions [4], an extension of the calculus of constructions with rewriting where the General Schema is the ACC condition. We use the notation $t{\downarrow}$ for the unique normal form of t.

4 Towards Consistency and Completeness

Consistency of the calculus of constructions (resp. calculus of inductive constructions) can be shown by rejecting all cases of a hypothetical normalized proof e of $(x : *)x$ in a *closed* environment, i.e. empty environment (resp. an environment containing only inductive definitions and no axioms). Our goal is to extend the definition of closed environments to the calculus of constructions with rewriting, allowing also a certain class of definitions by rewriting.

Let us try to identify that class. If we reanalyze e in the new setting, the only new possible normal form of e is an application of a function symbol f, coming from a rewrite definition $\mathsf{Rew}(\Gamma, R)$, to some arguments in normal form. There is no obvious argument why such terms could not be proofs of $(x : *)x$, but we could complete the consistency proof if we knew that such terms were always reducible. Let us call $\mathsf{COMP}(\Gamma, R)$ the condition on rewrite definitions we are looking for, meaning that function symbols from Γ are completely defined by the given set of rules R.

Note that the completeness of f has to be checked much earlier than it is used: we use it in a given closed environment $E = E_1; \mathsf{Rew}(\Gamma, R); E_2$ but it has to be checked when f is added to the environment, i.e. in the environment E_1. It implies that completeness checking has to account for environment extension and can be performed only with respect to arguments of types whose set of normal forms would not change in the future. This is the case for arguments of inductive types.

The requirement that functions defined by rewriting are completely defined could very well be included in the condition ACC. On the other hand, the separation between ACC and COMP is motivated by the idea of working with abstract function symbols, equipped with some rewrite rules not defining them completely. For example if + from Sect. 2 were declared using only the third rewrite rule, one could develop a theory of an associative function over natural numbers.

In the next section we define COMP and provide an always terminating and sound algorithm checking whether a rewrite definition is complete.

5 Checking Completeness

The intuition behind the definitions given below is the following. A rewrite definition $\mathsf{Rew}(\Gamma, R)$ satisfies COMP (or is *complete*) if for all f in Γ, the goal $f(x_1, \ldots, x_n)$ is *covered* by R. A goal is covered if all its instances are *immediately covered*, i.e reducible by R at the root position. Following the discussion from the previous section we limit ourselves to *normalized canonical instances*, i.e. built from constructors wherever possible.

Definition 2 (Canonical form and canonical substitution). *Given a judgment $E \vdash e : t$ we say that the term e is in* canonical *form if and only if:*

- *if $t\!\downarrow$ is an inductive type then $e = c(e_1, \ldots, e_n)$ for some constructor c and terms $e_1, \ldots, e_n$ in canonical form*
- *otherwise e is arbitrary*

Let Δ be a variable environment and E a correct environment. We call $\delta : \Delta \rightarrow E$ canonical *if for every variable $x \in \Delta$, the term $x\delta$ is canonical.*

From now on, let E be a global environment and let $\mathsf{Rew}(\Gamma, R)$ be a rewrite definition such that $E \vdash \mathsf{Rew}(\Gamma, R) : \mathsf{correct}$. Let $f : (x_1 : t_1) \ldots (x_n : t_n)\, t \in \Gamma$.

Definition 3. *A* goal *is a well-typed sequent $E; \Gamma; \Delta \vdash f(e_1, \ldots, e_n)$.*

A normalized canonical instance *of the goal $E; \Gamma; \Delta \vdash f(e_1, \ldots, e_n)$ is a well-typed sequent $E; \mathsf{Rew}(\Gamma, R); E' \vdash f(e_1\delta\downarrow, \ldots, e_n\delta\downarrow)$ for any canonical substitution $\delta : \Delta \rightarrow E; \mathsf{Rew}(\Gamma, R); E'$.*

A term e is immediately covered *by R if there is a rule $G \vdash l \longrightarrow r$ in R and a substitution γ such that $l\gamma = e$. By obvious extension we can also write that a goal or a normalized canonical instance is immediately covered by R.*

A goal is covered *by R if all its normalized canonical instances are immediately covered by R.*

Note that, formally, a normalized canonical instance is not a goal. The difference is that the conversion corresponding to the environment of an instance contains reductions defined by R, while the one of a goal does not.

Definition 4 (Complete definition). *A rewrite definition $\mathsf{Rew}(\Gamma; R)$ is* complete *in the environment E, which is denoted by $\mathsf{COMP}_E(\Gamma; R)$, if and only if for all function symbols $f : (x_1 : t_1) \ldots (x_n : t_n)\, t \in \Gamma$ the goal $E; \Gamma; x_1 : t_1; \ldots; x_n : t_n \vdash f(x_1, \ldots, x_n)$ is covered by R.*

Example 1. The terms `(S O)`, λ`x:nat.x` and `(Node O Leaf true O Leaf)` are canonical, while `(O + O)` and `(Node nA A b O Leaf)` are not. Given the definition of `rotr` from Sect. 2 consider the following terms:

$t_1 =$ `rotr (S (nA + nC)) (Node nA A b nC C)`
$t_2 =$ `rotr (S O) (Node O Leaf true O Leaf)`

Both (with their respective environments) are goals for `rotr`, and t_2 (with a slightly different environment) is also a normalized canonical instance of t_1. The goal t_1 is not immediately covered, but its instance t_2 is, as it is head-reducible by the second rule defining `rotr`. Since other instances of t_1 are also immediately covered, the goal is covered (see Example 3).

5.1 Splitting

The algorithm that we present in Sect. 5.3 checks that a goal is covered using successive splitting, i.e. replacement of variables of inductive types by constructor patterns. In presence of dependent types not all constructors can be put in every place. Consider the first rule of the definition of `rotr`. It is clear that only `Leaf` can replace `t` and in order to decide so one must use unification modulo conversion.

Even though in general this problem is undecidable, we assume the existence of a partial unification algorithm Alg, which given a unification problem U, returns a substitution γ, $\bot$ or $?$. If it returns a substitution γ then γ is the most general unifier of U and if it returns $\bot$, U has no unifier; $?$ means that the problem is too difficult for the algorithm (for the exact definitions we refer the reader to the appendix). An example of such a partial unification algorithm is constructor unification, that is first-order unification with constructors and type constructors as rigid symbols, answering $?$ whenever one compares a non-trivial pair of terms involving non-rigid symbols.

In the definitions below, we use $Ran(\gamma)$ for the variable part of the codomain of a substitution γ.

Definition 5 (Splitting). *Let $E; \Gamma; \Delta \vdash f(\vec{e})$ be a goal. A variable x is a* splitting variable *if $x : t \in \Delta$ and $t\!\downarrow = I\vec{u}$ for some inductive type $I \in E$.*

A splitting operation *considers all constructors c of the inductive type I and for each of them constructs the following unification problem U_c:*

$$E; \Gamma,\ \Delta; \Delta_c \vdash x = c(z_1, \ldots z_k) \qquad\qquad E; \Gamma,\ \Delta; \Delta_c \vdash I\vec{u} = I\vec{w}$$

where $c : (z_1 : Z_1) \ldots (z_k : Z_k).I\vec{w}$ and $\Delta_c = z_1 : Z_1, \ldots, z_k : Z_k$.

If $Alg(U_c) \neq ?$ for all c, the splitting is successful. In that case, let $Sp(x) = \{\sigma_c \mid \sigma_c = Alg(U_c) \wedge Alg(U_c) \neq \bot\}$. The result of splitting is the set of goals $\{E; \Gamma; Ran(\sigma_c) \vdash f(\vec{e})\sigma_c\}_{\sigma_c \in Sp(x)}$.

If $Alg(U_c) = ?$ for some c, the splitting fails.

Example 2. If one splits the goal `rotr n t` along the variable `n`, one gets two goals: `rotr O t` and `rotr (S m) t`. The first one is immediately covered by the first rule for `rotr` and if we split the second one along `t`, the `Leaf` case is impossible, because `tree O` does not unify with `tree (S m)` and the `Node` case gives `rotr (S (nA + nC)) (Node nA A b nC C)`.

The following lemma states the correctness of splitting, i.e. that splitting does not decrease the set of normalized canonical instances. Note that the lemma would also hold if we had a unification algorithm returning an arbitrary set of most general solutions, but in order for the coverage checking algorithm to terminate the set of goals resulting from splitting must be finite.

Lemma 1. *Let $E; \Gamma; \Delta \vdash f(\vec{e})$ be a coverage goal and let $\{E; \Gamma; Ran(\sigma_c) \vdash f(\vec{e})\sigma_c\}_{\sigma_c \in Sp(x)}$ be the result of successful splitting along $x : I\vec{u} \in \Delta$. Then every normalized canonical instance of $E; \Gamma; \Delta \vdash f(\vec{e})$ is a normalized canonical instance of $E; \Gamma; Ran(\sigma_c) \vdash f(\vec{e})\sigma_c$ for some $\sigma_c \in Sp(x)$.*

5.2 Preservation of Reducibility

Although one would expect that an immediately covered goal is also covered, it is not always true, even for confluent systems. In fact we need a property of critical pairs that is stronger than just joinability. Let us suppose that `or : bool → bool → bool` is defined by four rules by cases over `true` and `false` and that `if : bool → bool → bool → bool` is defined by two rules by cases on the first argument.

```
Inductive I : bool → Set := C : forall b:bool, I (or b b).
Symbol f:  forall b:bool, I b → bool
Rules
  f (or b b) (C b) ⟶ if b (f true (C true)) (f false (C false))
```

In the example presented above all expressions used in types and rules are in normal form, all critical pairs are joinable, the system is terminating, and splitting of `f b i` along `i` results in the only reducible goal `f (or b b) (C b)`. In spite of that `f` is not completely defined, as `f true (C true)` is a normalized canonical instance of `f (or b b) (C b)` and it is not reducible. In order to know that an immediately covered goal is always covered we need one more condition on rewrite rules, called preservation of reducibility.

Definition 6. *Definition by rewriting* $\mathsf{Rew}(\Gamma, R)$ preserves reducibility *in an environment E if for every critical pair $\langle f(\vec{u}), r\delta\rangle$ of a rule $G_1 \vdash f(\vec{e}) \longrightarrow r$ in R with a rule $G_2 \vdash g \longrightarrow d$ coming from R or from some other rewrite definition in E, the term $f(\vec{u}\downarrow)$ is immediately covered by R.*

Note that by using ? variables in rewrite rules one can get rid of (some) critical pairs and hence make a definition by rewriting satisfy this property. In the example above one could write `f ? (C b)` as the left-hand side (which would make the system non-terminating, but that is another story). Of course all orthogonal rewrite systems, in particular inductive elimination schemes, as defined in [22], preserve reducibility.

Lemma 2. *Let $\mathsf{Rew}(\Gamma, R)$ preserve reducibility in an environment E and let $E; \Gamma; \Delta \vdash f(\vec{e})$ be a goal. If it is immediately covered then it is covered.*

5.3 Coverage Checking Algorithm

In this section we present an algorithm checking whether a set of goals is covered by the given set of rewrite rules. The algorithm is correct only for definitions that preserve reducibility. The algorithm, in a loop, picks a goal, checks whether it is immediately covered, and if not, splits the goal replacing it by the subgoals resulting from splitting. In order to ensure termination splitting is limited to *safe splitting variables*. Intuitively, a splitting variable is safe if it lies within the contour of the left-hand side of some rule when we superpose the tree representation of the left-hand side with the tree representation of the goal. The number of nodes that have to be added to the goal in order to fill the tree of the left-hand side is called a distance, and a sum of distances over all rules is called a measure. Since the measures of goals resulting from splitting are smaller than the measure of the original goal, the coverage checking algorithm terminates.

Safe splitting variables used in the algorithm are computed by the splitting matching procedure: $SV(\phi, R)$ is the set of safe splitting variables for the goal ϕ along left-hand sides of rules from R.

We refer the reader to the appendix for the formal definitions of the notions mentioned above and the corresponding lemmas.

Definition 7 (Coverage checking algorithm). *Let W be a set of pairs consisting of a goal and a set of safe variables of that goal along left-hand sides of rules from R and let CE be a set of goals. The coverage checking algorithm works as follows:*

Initialize
$W = \{(E; \Gamma; x_1 : t_1; \ldots; x_n : t_n \vdash f(x_1, \ldots, x_n),\;\; SV(f(x_1, \ldots, x_n), R))\}$
$CE = \emptyset$

Repeat
1. *choose a pair (ϕ, X) from W,*
2. *if ϕ is immediately covered by one of the rules from R then* $W := W \setminus \{(\phi, X)\}$
3. *otherwise*
   (a) *if $X = \emptyset$ then $W := W \setminus \{(\phi, X)\}$, $CE := CE \cup \{\phi\}$*
   (b) *otherwise choose $x \in X$; split ϕ along x*
      i. *if splitting is successful and returns $\{\phi_1 ... \phi_n\}$ then* $W := W \setminus \{(\phi, X)\} \cup \{(\phi_i, SV(\phi_i, R))\}_{i=1...n}$,
      ii. *otherwise $W := W \setminus \{(\phi, X)\} \cup \{(\phi, X \setminus \{x\})\}$*

until $W = \emptyset$

Lemma 3. *If $\mathsf{Rew}(\Gamma, R)$ preserves reducibility and the algorithm stops with $CE = \emptyset$ then the initial goal is covered.*

Example 3. The beginning of a possible run of the algorithm for the function `rotr` is presented already in Example 2. Both splitting operations are performed on safe variables, as required. We are left with the remaining goal `rotr (S (nA + nC)) (Node nA A b nC C)`. Splitting along `A` results in:

```
rotr (S (O + nC)) (Node O Leaf b nC C)
rotr (S((S(nX+nZ))+nC)) (Node (S(nX+xZ)) (Node nX X y nZ Z) b nC C)
```

immediately covered by the second and the third rule respectively.

Since we started with the initial goal `rotr n t` and since the definition of `rotr` preserves reducibility, it is complete.

When the coverage checking algorithm stops with $CE \neq \emptyset$, we cannot deduce that R is complete. The set CE contains potential counterexamples. They can be true counterexamples, false counterexamples, or goals for which splitting failed along all safe variables, due to incompleteness of the unification algorithm. In some cases further splitting of a false counterexample may result in reducible goals or in the elimination of the goal as uninhabited, but it may also loop. Some solutions preventing looping (finitary splitting) can be found in [16].

Unfortunately splitting failure due to incompleteness of the unification may happen while checking coverage of a definition by case analysis over complex dependent inductive types (for example trees of size 2), even if rules for all constructors are given. Therefore, it is advisable to add a second phase to our algorithm, which would treat undefined output of unification as success. Using this second phase of the algorithm, one can accept all definitions by case analysis that can be written in Coq.

6 Completeness Implies Canonicity Implies Consistency

It follows that completeness of definitions by rewriting guarantees canonicity and logical consistency.

Definition 8. *An environment E is* closed *if and only if it contains only inductive definitions and complete definitions by rewriting, i.e. for each partition of E into $E_1; \mathsf{Rew}(\Gamma, R); E_2$ the condition $\mathsf{COMP}_{E_1}(\Gamma, R)$ is satisfied.*

Lemma 4 (Canonicity). *Let E be a closed environment. If $E \vdash e : t$ and e is in normal form then e is canonical.*

Theorem 1. *Every closed environment is consistent.*

Proof. Let E be a closed environment. Suppose that $E \vdash e : (x : \star)x$. Since $E \vdash \texttt{ok}$ and $E \vdash \mathsf{Ind}(\texttt{False} : \star :=) : \mathsf{correct}$ we have $E' \vdash \texttt{ok}$ where $E' = E; \mathsf{Ind}(\texttt{False} : \star :=)$. Moreover E' is a closed environment.

Hence, we have $E' \vdash e\ \texttt{False} : \texttt{False}$. By Lemma 4, the normal form of $e\ \texttt{False}$ is canonical. Since `False` has no constructors, this is impossible.

7 Conclusions and Related Work

In this paper we study consistency of the calculus of constructions with rewriting. More precisely, we propose a formal system extending an arbitrary PTS with inductive definitions and definitions by rewriting. Assuming that suitable positivity and acceptance conditions guarantee termination and confluence, we formalize the notion of complete definitions by rewriting. We show that in every environment consisting only of inductive definitions and complete definition by rewriting there is no proof of $(x : *)x$. Moreover, we present a sound and terminating algorithm for checking completeness of definitions. It is necessarily incomplete, since in presence of dependent types emptiness of types trivially reduces to completeness and the former is undecidable.

Our coverage checking algorithm resembles the one proposed by Coquand in [10] for Martin-Löf type theory and used by McBride for his OLEG calculus [14]. In these works the procedure consisting in successive case-splittings is used to interactively built pattern matching equations, or to check that a given set of equations can be built this way. Unlike in our paper, Coquand and McBride do not have to worry whether all instances of a reducible subgoal are reducible. Indeed, in [10] pattern matching equations are meant to be applied to terms modulo conversion, and in [14] equations (or rather the order of splittings in the successful run of the coverage checking procedure) serve as a guideline to construct an OLEG term verifying the equations. Equations themselves are never used for reduction and the constructed term reduces according to existing rules.

In our paper rewrite rules are matched against terms modulo α. Rewriting has to be confluent, strongly normalizing and has to preserve reducibility. Under these assumptions we can prove completeness for all examples from [10] and for the class of pattern matching equations considered in [14]. In particular we can deal with elimination rules for inductive types and with Streicher's axiom K stating that all proofs of `(eq A a a)` are equal. Moreover, we can accept definitions which depart from standard pattern matching, like `rotr` and $+$ (for the axiom K and more examples we refer the reader to the appendix).

The formal presentation of our algorithm is directly inspired by the recent work of Pfenning and Schürmann [16]. A motivation for that paper was to verify that a logic program in the Twelf prover covers all possible cases. In LF, the base calculus of Twelf, there is no polymorphism, no rewriting and conversion is modulo $\beta\eta$. The authors use higher-order matching modulo $\beta\eta$, which is decidable for patterns of Miller and strict patterns. Moreover, since all types and function symbols are known in advance, the coverage is checked with respect to all available function symbols. In our paper, conversion contains rewriting and it cannot be used for matching; instead we use matching modulo α. This simplifies the algorithm searching for safe splitting variables, but on the other hand it does not fit well with instantiation and normalization. To overcome this problem we introduce the notions of normalized canonical instance and preservation of reducibility which were not present in previously mentioned papers. Finally, since the sets of function symbols and rewrite rules grow as the environment extends, coverage is checked with respect to constructors only.

Even though the worst-case complexity of the coverage checking is clearly exponential, for practical examples the algorithm should be quite efficient. It is very similar in spirit to the algorithms checking exhaustiveness of definitions by pattern matching in functional programming languages and these are known to work effectively in practice.

An important issue which is not addressed in this paper is to know how much we extend conversion. Of course it depends on the choice of conditions ACC and POS and on the unification algorithm used for coverage checking. In particular, some of the definitions by pattern matching can be encoded by recursors [11], so if ACC is strict, we may have no extension at all. In general there seems to be at least two kinds of extensions. The first are non-standard elimination rules for inductive types, but the work of McBride shows that the axiom K is sufficient to encode all other definitions by pattern matching considered by Coquand. The second are additional rules which extend a definition by pattern matching (like associativity for $+$). It is known that for first-order rewriting, these rules are inductive consequences of the pattern matching ones, i.e. all their canonical instances are satisfied as equations (Thm. 7.6.5 in [17]). Unfortunately, this is no longer true for higher-order rules over inductive types with functional arguments. Nevertheless we believe that such rules are also some sort of inductive consequences of the pattern matching part.

Our completeness condition COMP verifies closure properties defined in [7, 8]. Hence, it is adequate for a smooth integration of rewriting with the module system implemented recently in Coq.

References

1. Franco Barbanera, Maribel Fernández, and Herman Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6):613–660, 1997.

2. Henk Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, vol. 2, chap. 2, pp. 117–309. Oxford University Press, 1992.
3. Bruno Barras and Benjamin Grégoire. On the role of type decorations in the calculus of inductive constructions. In C.-H. Luke Ong, editor, *Proc. of CSL'05*, *LNCS* 3634, pp. 151–166. Springer, 2005
4. Frédéric Blanqui. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.
5. Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. The Calculus of Algebraic Constructions. In P. Narendran and M. Rusinowitch, editors, *Proc. of RTA'99*, *LNCS* 1631, pp. 301–316. Springer, 1999
6. Edwin Brady, Connor McBride, and James McKinna. Inductive families need not store their indices. In S. Berardi, M. Coppo, and F. Damiani, editors, *TYPES 2003, Revised Selected Papers*, *LNCS* 3085, pp. 115–129. Springer, 2004.
7. Jacek Chrząszcz. Modules in Coq are and will be correct. In S. Berardi, M. Coppo, and F. Damiani, editors, *TYPES 2003, Revised Selected Papers*, *LNCS* 3085, pp. 130–146. Springer, 2004.
8. Jacek Chrząszcz. *Modules in Type Theory with Generative Definitions*. PhD thesis, Warsaw Univerity and University of Paris-Sud, Jan 2004.
9. The Coq proof assistant. `http://coq.inria.fr/`.
10. Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pp. 71–83, Båstad, Sweden, 1992.
11. Cristina Cornes. *Conception d'un langage de haut niveau de répresentation de preuves*. PhD thesis, Université Paris VII, 1997.
12. J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.
13. Emmanuel Kounalis. Completeness in data type specifications. In *Proc. of EUROCAL 85*, *LNCS* 204, pp. 348–362. Springer-Verlag, 1985.
14. Conor McBride. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, University of Edinburgh, 1999.
15. Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. F. Groote, editors, *Proc. of TLCA'93*, *LNCS* 664, pp. 328–345. Springer-Verlag, 1993
16. Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proc. of TPHOLs'03*, *LNCS* 2758, pp. 120–135. Springer, 2003.
17. Terese, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
18. J.-J. Thiel. Stop loosing sleep over incomplete specifications. In *Proc. of POPL'84*, pp. 76–82. ACM Press, 1984.
19. Daria Walukiewicz-Chrząszcz. Termination of rewriting in the calculus of constructions. *Journal of Functional Programming*, 13(2):339–414, 2003.
20. Daria Walukiewicz-Chrząszcz. *Termination of Rewriting in the Calculus of Constructions*. PhD thesis, Warsaw University and University Paris XI, 2003.
21. Daria Walukiewicz-Chrząszcz and Jacek Chrząszcz. Consistency and completeness of rewriting in the calculus of constructions. *Proc. of IJCAR 2006*, *LNAI, LNCS* 4130, pp. 619–631. Springer, 2006.
22. Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.

A More examples

A.1 Polymorphic map

Consider the type `list` and the definition of function `map`:

```
Inductive list : (A:Set) : Set  :=  nil : list A
                                  |  cons : A → list A → list A

Symbol map : forall (A:Set), (A → A) → list A → list A
Rules
  map A f (nil A) ⟶ nil A
  map A f (cons A a l) ⟶ cons A (f a) (map A f l)
  map A λx.x l ⟶ l
```

Function `map` is completely defined by first two rules. The third one reflects a property of `map`, but an advantage of writing it as a rewrite rule is to use it in the conversion. Matching modulo α (and not syntactical) is needed for λ`x.x`.

A.2 Streicher's axiom K

Consider the type `eq` and the definition of function `f`:

```
Inductive  eq (A:Set)(a:A): A → Set  :=  refl: eq A a a.

Symbol f : forall (A:Set)(a b:A)(p q: eq A a b), (eq (eq A a b) p q)
Rules
  f A a a (refl A a) (refl A a) ⟶ refl (eq A a a) (refl A a).
```

Function `f` is completely defined since two subsequent splittings of `f A a b p q`, along `p` and along `q`, result in `f A a a (refl A a) (refl A a)` which is exactly the left-hand side of the only rule for `f`.

Note that the rule for f can be written also in a left-linear form:

```
  f A a ?1 (refl ?2 ?3) (refl ?4 ?5) ⟶ refl (eq A a a) (refl A a).
```

A.3 Non pattern matching rules

These are two more examples of complete definitions which are not definitions by pattern matching.

```
Symbol or : bool → bool → bool
  Rules
    or x x ⟶ x
    or true y ⟶ true
    or x true ⟶ true
```

```
Symbol lt, diff : nat → nat → bool
  Rules
    lt O y ⟶ diff O y
    lt x O ⟶ false
    lt (S x) (S y) ⟶ lt x y

    diff x x ⟶ false
    diff O (S y) ⟶ true
    diff (S x) O ⟶ true
    diff (S x) (S y) ⟶ diff x y
```

B Unification

Definition 9 (Unification problem). *A quadruple $E, \Delta \vdash t = s$, where E is an environment, Δ a variable environment and s, t are terms, is a* unification equation in E*. A* unification problem in E *is a finite set of unification equations. Without loss of generality we may assume that variable environments (Δ) in all equations are the same.*

A unifier *or a* solution *of the unification problem U is a substitution $\gamma : \Delta \to E; E'$ such that $E; E' \vdash t\gamma \approx s\gamma$ for every $E; \Delta \vdash t = s$ in U. We say that E' is the co-domain of γ, which is denoted by $Ran(\gamma)$.*

A unifier γ is the most general unifier *if $Ran(\gamma)$ is a variable environment Δ' and for every unifier $\delta : \Delta \to E; E''$ there exist a substitution $\delta' : \Delta' \to E; E''$, such that $\gamma; \delta' = \delta$.*

Definition 10 (Correct unification algorithm). *A* unification algorithm *is a procedure which for every environment E and unification problem $U = \{E; \Delta \vdash t_i = s_i\}$ returns a substitution γ, a bottom $\perp$, or a question mark ?. The algorithm is* correct *iff: if it answers γ it is the most general unifier $\gamma : \Delta \to E; \Delta'$ such that $\Delta' \subseteq \Delta$ and if it answers $\perp$, there is no unifier.*

C Splitting

Proof (of Lemma 1). Let $E; \mathsf{Rew}(\Gamma, R); E' \vdash f(e_1\delta\!\downarrow, \dots, e_n\delta\!\downarrow)$ be a normalized canonical instance according to a substitution $\delta : \Delta \to E; \mathsf{Rew}(\Gamma, R); E'$. Since δ is canonical, $x\delta$ is a constructor term $c(s_1, \dots s_k)$ for some constructor c of I. Let us show that $E; \mathsf{Rew}(\Gamma, R); E' \vdash f(e_1\delta\!\downarrow, \dots, e_n\delta\!\downarrow)$ is a normalized canonical instance of $E; \Gamma; Ran(\sigma_c) \vdash f(e_1\sigma_c, \dots, e_n\sigma_c)$.

Note that $\delta \cup [\vec{s}/\vec{z}] : \Delta; \Delta_c \to E; \mathsf{Rew}(\Gamma, R); E'$ is a solution of the unification problem from the definition of splitting. Indeed, $x\delta = c(s_1, \dots s_k) = c(z_1, \dots z_k)[\vec{s}/\vec{z}]$ and $E; \mathsf{Rew}(\Gamma, R); E' \vdash (I\vec{u})\delta \approx (I\vec{w})[\vec{s}/\vec{z}]$ since they are both types of $c(s_1, \dots s_k)$ in $E; \mathsf{Rew}(\Gamma, R); E'$.

By definition of σ_c we have $\delta \cup [\vec{s}/\vec{z}] = \sigma_c; \delta'$ for some $\delta' : Ran(\sigma_c) \rightarrow E; \mathsf{Rew}(\Gamma, R); E'$. Consequently, $\delta'(z_m) = s_m$ for $z_m \in \Delta_c$ and $\delta'(y) = \delta(y)$ for $y \in \Delta$. Since $\vec{s}$ are canonical terms δ' is a canonical substitution.

Let us look closely at $E; \mathsf{Rew}(\Gamma, R); E' \vdash f((e_1\sigma_c)\delta' \downarrow, \ldots, (e_n\sigma_c)\delta' \downarrow)$ which is a normalized δ'-instance of $E; \Gamma; Ran(\sigma_c) \vdash f(e_1\sigma_c, \ldots, e_n\sigma_c)$. We have $(e_m\sigma_c)\delta' \downarrow = (e_m\sigma_c\delta') \downarrow = (e_m\delta) \downarrow$ for every m and consequently $E; \mathsf{Rew}(\Gamma, R); E' \vdash f(e_1\delta \downarrow, \ldots, e_n\delta \downarrow)$ is a normalized canonical instance of $E; \Gamma; Ran(\sigma_c) \vdash f(e_1\sigma_c, \ldots, e_n\sigma_c)$.

D Preservation of reducibility

Lemma 5. *Let $E \vdash e : t$ and $e = f(e_1, \ldots e_n)$, where f comes from* $\mathsf{Rew}(\Gamma, R)$ *which preserves reducibility. If e is head-reducible by R then $f(e_1 \downarrow, \ldots e_n \downarrow)$ is also head-reducible by R.*

Proof. By induction on $\longrightarrow$. If $e_1, \ldots e_n$ are in normal forms then the conclusion is obvious. Otherwise, let $G_1 \vdash f(\vec{l}) \longrightarrow r$ be a rule from R and γ a substitution such that $f(\vec{e}) = f(\vec{l})\gamma$ and let us make one reduction step $e_i \longrightarrow e'_i$, using $G_2 \vdash g \longrightarrow d$.

There are two possibilities: the reduction in e_i happens either in substitution γ, i.e. in the term $\gamma(x)$, where x is a free variable of $f(\vec{l})$, or it happens on position p that belong to $f(\vec{l})$. In the former case, let us do identical reduction in all other instances of x. Obviously, we get a term $f(e'_1, \ldots e'_n)$ that is smaller than e in $\longrightarrow$ and is still an instance of $f(\vec{l})$. Hence by induction hypothesis we get the desired conclusion.

Otherwise, $f(\vec{l})$ and g superpose and we have $f(\vec{l})\gamma|_p = g\xi$ for some position p and substitution ξ. Let δ be the most general unifier of $f(\vec{l})|_p$ and g and let $\langle f(\vec{u}), r\delta \rangle$ be the corresponding critical pair. Since δ is the most general unifier, there exists σ such that $f(\vec{l})\gamma|_p = f(\vec{l})|_p\delta\sigma = g\delta\sigma = g\xi$. Consequently, $f(e_1, \ldots e'_i \ldots e_n) = f(\vec{u})\sigma$. By preservation of reducibility $f(\vec{u} \downarrow)$ is head-reducible by R, hence $f(\vec{u}\downarrow)\sigma$ is also head-reducible by R. Like above we can apply induction hypothesis and deduce that $f(\vec{e}\downarrow)$ is head-reducible by R.

Proof (of Lemma 2). Let $E; \Gamma; \Delta \vdash f(\vec{e})$ be a goal immediately covered by R and let $\delta : \Delta \rightarrow E; \mathsf{Rew}(\Gamma, R); E'$ be a canonical substitution. Obviously, $E; \mathsf{Rew}(\Gamma, R); E' \vdash f(\vec{e}\delta)$ is immediately covered by R. Hence, by Lemma 5 $E; \mathsf{Rew}(\Gamma, R); E' \vdash f(\vec{e}\delta \downarrow)$ is also immediately covered by R.

E Coverage checking algorithm

Let us start with the splitting matching algorithm which finds variables in t_1 that lie within the contour of t_2.

$$\frac{t_1 <_{p\cdot 1} t_1' \Rightarrow S_1 \qquad t_2 <_{p\cdot 2} t_2' \Rightarrow S_2}{(x\!:\!t_1)t_2 <_p (x\!:\!t_1')t_2' \Rightarrow S_1 \cup S_2} \qquad \frac{t_1 <_{p\cdot 1} t_1' \Rightarrow S_1 \qquad t_2 <_{p\cdot 2} t_2' \Rightarrow S_2}{\lambda x\!:\!t_1.t_2 <_p \lambda x\!:\!t_1'.t_2' \Rightarrow S_1 \cup S_2}$$

$$\frac{t_1 <_{p\cdot 1} t_1' \Rightarrow S_1 \qquad t_2 <_{p\cdot 2} t_2' \Rightarrow S_2}{t_1\, t_2 <_p t_1'\, t_2' \Rightarrow S_1 \cup S_2}$$

$$\frac{t_1 <_{p\cdot 1} t_1' \Rightarrow S_1 \quad \ldots \quad t_n <_{p\cdot n} t_n' \Rightarrow S_n}{h(t_1, \ldots, t_n) <_p h(t_1', \ldots, t_n') \Rightarrow S_1 \cup \cdots \cup S_n}\ h \text{ is a constant}$$

$$\frac{t_1, t_2 \notin \mathit{Var} \quad \mathrm{head}(t_1) \neq \mathrm{head}(t_2)}{t_1 <_p t_2 \Rightarrow \{\bot\}} \qquad \frac{t_1 \in \Delta_1 \quad t_2 \notin (\mathit{Var} \setminus \Delta_2)}{t_1 <_p t_2 \Rightarrow \{p\}}$$

$$\frac{\begin{array}{c}(t_1 \in (\mathit{Var} \setminus \Delta_1) \wedge t_1 = t_2) \vee \\ (t_1 \notin \mathit{Var} \wedge t_2 \in \Delta_2)\end{array}}{t_1 <_p t_2 \Rightarrow \emptyset} \qquad \frac{\begin{array}{c}(t_1 \in (\mathit{Var} \setminus \Delta_1) \wedge t_1 \neq t_2) \vee \\ (t_2 \in (\mathit{Var} \setminus \Delta_2) \wedge t_1 \neq t_2)\end{array}}{t_1 <_p t_2 \Rightarrow \{\bot\}}$$

Fig. 2. Splitting matching rules

Definition 11 (Splitting matching). *Given two sequents $\Delta_1 \vdash t_1$ and $\Delta_2 \vdash t_2$, the* splitting matching algorithm *returns the unique set S, such that $t_1 <_\Lambda t_2 \Rightarrow S$ can be obtained using the rules given in Fig. 2, parametrized by the environments Δ_1 and Δ_2. The set S is a subset of $\{\bot\} \cup \{p \in Pos(t_1) \mid t_1|_p \in \Delta_1\}$.*

Definition 12 (Safe splitting variable). *Let $\Delta_1 \vdash t_1$ and $\Delta_2 \vdash t_2$ be sequents such that t_2 is a left-hand side of a rule from R and let S be a set such that $t_1 <_\Lambda t_2 \Rightarrow S$ and $\bot \notin S$. A variable $x \in \Delta_1$ is a* safe splitting variable for $\Delta_1 \vdash t_1$ along $\Delta_2 \vdash t_2$ *if it is a splitting variable and there exist $p \in S$ such that $t_1|_p = x$ and either t_2 is a variable declared in Δ_2 or $t_2|_p = c(\vec{e})$ for some constructor c and some terms $\vec{e}$.*

A set of safe splitting variables for $\Delta_1 \vdash t_1$ along $\Delta_2 \vdash t_2$ is denoted by $SV(\Delta_1 \vdash t_1, \Delta_2 \vdash t_2)$ or $SV(t_1, t_2)$ for short.

Example 4. In the goal `rotr' (S (S nC)) (Node O Leaf b (S nC) C)` there are two safe splitting variables: `b` and `C`.

Definition 13 (Distance). *Let $\Delta_1 \vdash t_1$ and $\Delta_2 \vdash t_2$ be sequents and S be a set such that $t_1 <_\Lambda t_2 \Rightarrow S$. If $\bot \notin S$ then the* distance *of $\Delta_1 \vdash t_1$ from $\Delta_2 \vdash t_2$, denoted by $dist(\Delta_1 \vdash t_1, \Delta_2 \vdash t_2)$ or $dist(t_1, t_2)$, equals $\sum_{p \in S} |t_2|_p|$; otherwise it is equal to 0.*

The following two lemmas state that the distance of a term decreases when we apply a substitution, and it decreases strictly if it is a substitution resulting from splitting.

Lemma 6 (Distance of a substituted sequent). *Let $\Delta_1 \vdash t_1$ and $\Delta_2 \vdash t_2$ be sequents and S be a set such that $t_1 <_\Lambda t_2 \Rightarrow S$. Then for every $\gamma : \Delta_1 \to \Delta'$ we have $dist(\Delta' \vdash t_1\gamma, \Delta_2 \vdash t_2) \le dist(\Delta \vdash t_1, \Delta_2 \vdash t_2)$.*

Moreover, if $\perp \in S$ then $dist(\Delta' \vdash t_1\gamma, \Delta_2 \vdash t_2) = dist(\Delta \vdash t_1, \Delta_2 \vdash t_2) = 0$.

Proof. Let S' be a set such that $t_1\gamma <_\Lambda t_2 \Rightarrow S'$ and let us denote $dist(\Delta \vdash t_1, \Delta_2 \vdash t_2)$ by d and $dist(\Delta' \vdash t_1\gamma, \Delta_2 \vdash t_2)$ by d_γ.

If $\perp \in S$ then $d = 0$. Note that $\perp \in S$ if and only if there is a position p such that subterms occurring at p in t_1 and t_2 either have different head symbols, or $t_2|_p$ (resp. $t_1|_p$) is a bound variable in t_2 (resp. t_1) and $t_1|_p \neq t_2|_p$. Of course, if we compare $t_1\gamma|_p$ and $t_2|_p$ we still have either different head-symbols or $t_2|_p$ (resp. $t_1|_p$) a bound variable and $t_1\gamma|_p \neq t_2|_p$. Hence $d_\gamma = 0$.

If $\perp \notin S$ then $d = \sum_{p \in S} |t_2|_p| > 0$. If $\perp \in S'$ then obviously $0 = d_\gamma < d$. Otherwise, let us take $p \in S$ and the set $Q_p = \{q \in S' \mid q \le p\}$. Since all position from Q_p are independent (as $t_1|_q \in Var$ for every $q \in S'$) we have $\sum_{q \in Q_p} |t_2|_q| \le |t_2|_p|$ and the equality holds only if $Q_p = \{p\}$. Let us show that $\forall q \in S' \exists! p \in S\ q \le p$. Indeed, assuming that $\perp \notin S'$, $q \in S'$ either because $q \in S$ and $(t_1|_q)\gamma \in \Delta'$ or because there is a position $p \in S$ such that $q = p \cdot q'$ for some q' and $(t_1|_p\gamma)|_{q'} \in \Delta'$. Of course, in the latter case such p is unique since positions in S are independent.

Hence $S' = \bigcup_{p \in S} Q_p$ and $d_\gamma = \sum_{q \in S'} |t_2|_q| = \sum_{p \in S} \sum_{q \in Q_p} |t_2|_q| \le \sum_{p \in S} |t_2|_p| = d$.

Lemma 7 (Distance after splitting strictly decreases). *Let $E; \Gamma; \Delta \vdash f(e_1, \dots, e_n)$ be a goal, $t = f(e_1, ..., e_n)$, $G \vdash l \longrightarrow r$ one of the rewrite rules for f in R and let S be a set such that $t <_\Lambda l \Rightarrow S$ and $\perp \notin S$. If $x : I\vec{u} \in S$ is a safe splitting variable and $\{\phi_1 ... \phi_m\}$ is a result of successful splitting along x then $dist(\phi_i, G \vdash l) < dist(\Delta \vdash t, G \vdash l)$ for every i.*

Proof. Let $\sigma_c \in Sp(x)$ be one of the substitutions resulting from splitting and S_c be a set such that $t\sigma_c <_\Lambda l \Rightarrow S_c$. By Lemma 6 we have $dist(t\sigma_c, l) \le dist(g, l)$. Let us analyze the proof of that lemma and show that in case of a substitution resulting from splitting there is a strict inequality between $dist(t\sigma_c, l)$ and $dist(t, l)$. In the proof it was noticed that for every $p \in S$, $\sum_{q \in Q_p} |l|_q| \le |l|_p|$, where $Q_p = \{q \in S_c \mid p \le q\}$ and that $\sum_{q \in Q_p} |l|_q| = |l|_p|$ only if $Q_p = \{p\}$. Consequently, if we show that there exist position p such that $p \notin Q_p$ then we get immediately $dist(t\sigma_c, l) < dist(g, l)$.

Since $x : I\vec{u}$ is a safe splitting variable for t along l there exist a position $p \in S$ such that $g|_p = x$ and $l|_p \in G$ or $l|_p = c'(\vec{a})$ for some constructor c'. Since σ_c results from successful splitting, $x\sigma_c = c(\vec{b})$ for some $\vec{b}$. Now, there are three cases. If $l|_p \in G$ then $Q_p = \emptyset$ (and hence $p \notin Q_p$) . Otherwise, if $c' \neq c$ then we fall into an easy case when $\perp \in S_c$ ($dist(t\sigma_c, l) = 0 < |c'(\vec{a})| \leq dist(t, l)$). Finally, if $c' = c$, the computation of S_c passes through the step $c(\vec{b}) <_p c(\vec{a}) \Rightarrow ?$. This means that all positions in Q_p come from $\vec{b}$ and that they are longer than p. Thus $p \notin Q_p$ and $dist(t\sigma_c, l) < dist(t, l)$.

Definition 14 (Measure of a goal). *Let* $E; \Gamma; \Delta \vdash f(e_1, \ldots, e_n)$ *be a goal and* $R_f = \{G_i \vdash l_i \longrightarrow r_i\}_{i=1..m}$ *be the set of rules for* f*. The* measure of $E; \Gamma; \Delta \vdash f(e_1, \ldots, e_n)$ *equals* $\sum_{i=1..m} dist(\Delta \vdash f(e_1, \ldots, e_n), G_i \vdash l_i)$.

It follows directly from Lemmas 6 and 7 that the measure of a goal strictly decreases after applying a substitution resulting from splitting.

Lemma 8 (Measure after splitting strictly decreases). *Let* $E; \Gamma; \Delta \vdash f(e_1, \ldots, e_n)$ *be a goal,* $t = f(e_1, ..., e_n)$*,* $R_f = \{G_i \vdash l_i \longrightarrow r_i\}_{i=1..m}$ *be the set of rules for* f *and* S *be a set such that* $t <_\Lambda l_j \Rightarrow S$ *for some* $j \in 1..m$*. If* $\perp \notin S$*,* $x : I\vec{u} \in S$ *is a safe splitting variable and* $\{\phi_1 ... \phi_k\}$ *a result of successful splitting along* x *then for every* i *the measure of* ϕ_i *is strictly smaller than the measure of* $E; \Gamma; \Delta \vdash t$*.*

Proof. For every $\sigma_c \in Sp(x)$ we have to show that $\sum_{i=1..m} dist(t\sigma_c, l_i) < \sum_{i=1..m} dist(t, l_i)$. It easily follows from $dist(t\sigma_c, l_j) < dist(t, l_j)$, which is true by Lemma 7, and $dist(t\sigma_c, l_i) \leq dist(t, l_i)$ for all $i \in 1..m$, $i \neq j$, which follows from Lemma 6.

Lemma 9. *For all* W *and* R *the cover checking algorithm terminates.*

Proof. Let us consider the following measure $M(W)$: the multiset of lexicographically ordered pairs consisting of the measure of ϕ and the size of X, for all $(\phi, X) \in W$. We will show that every loop of the algorithm strictly decreases $M(W)$. Consider $(\phi, X) \in W$. If ϕ is immediately covered then obviously the measure of $W \setminus \{(\phi, X)\}$ is strictly smaller than the measure of W. Otherwise, we split ϕ along some $x \in X$. If splitting fails then (ϕ, X) is replaced by $(\phi, X \setminus \{x\})$ and the size of the second component strictly decreases. If splitting is successful and returns $\{\phi_1 ... \phi_n\}$ then (ϕ, X) is replaced by $\{(\phi_i, SV(\phi_i, R))\}_{i=1...n}$. By Lemma 8 the measures of goals from $\{\phi_1 ... \phi_n\}$ are strictly smaller than the measure of ϕ and consequently $M(W)$ strictly decreases.

Proof (of Lemma 3). Let us consider a successful run of the algorithm, performing a finite number of times the body of the *Repeat* loop and resulting in

$CE = \emptyset$. By induction on n, the number of `Repeat` steps until the end of the algorithm, we prove that the goals appearing in W are covered.

The base case is trivial: for $n = 0$, W is the final set of goals and it is empty.

Now suppose that n steps before the end of the algorithm all goals in W are covered and let us check that this was true $n + 1$ steps before the end, i.e. one step of the algorithm earlier. Let us call W_n and W_{n+1} the goals from W in the corresponding moments and let us analyse possible relations between W_n and W_{n+1}.

In case 2, W_{n+1} contains all goals from W_n and one goal ϕ which is immediately covered by a rule in R. By preservation of reducibility (Lemma 2) every normalized canonical instance of ϕ is also immediately covered and consequently all goals of W_{n+1} are covered.

Case $3(a)$ is impossible since it makes the set CE non-empty.

In case $3(b)i$, W_{n+1} contains some of the goals from W_n and one goal ϕ whose subgoals resulting from succesful splitting are already in W_n. By Lemma 1) the set of normalized canonical instances of these subgoals contains the set of normalized canonical instances of ϕ. Hence W_{n+1} is covered.

In case $3(b)ii$ we have $W_{n+1} = W_n$.

Hence the initial goal in W is also covered.

F Completeness Implies Canonicity Implies Consistency

Proof (of Lemma 4). By induction on the size of e. If $t\downarrow$ is not an inductive type then any e is canonical. Otherwise, let us analyze the structure of e. It cannot be a product, an abstraction or a sort because $t\downarrow$ is an inductive type. Since E is closed, it is not a variable either. Let us suppose that e is an application whose maximal form is $e' e_1 \dots e_n$. The term e' can be neither a product, a sort (they cannot be applied), nor a variable (E is closed). It is not an application, since $e' e_1 \dots e_n$ is maximal, and it is not an abstraction, since e is in normal form. The only possibility left is an application of a constant to some arguments $h(e_1, \dots, e_n)$.

Since $t\downarrow$ is an inductive type, h cannot be an inductive type. If it is a constructor, by induction hypothesis $e_1, \dots, e_n$ are in canonical form and so is $h(e_1, \dots, e_n)$. If h is a function symbol then $E = E_1; \mathsf{Rew}(\Gamma, R); E_2$ for some E_1, E_2 and $h : (x_1 : t_1) \dots (x_n : t_n)t \in \Gamma$. Since E is closed, $\mathsf{Rew}(\Gamma, R)$ is complete. Let us show that $E \vdash h(e_1, \dots, e_n)$ is a normalized canonical instance of $E_1; \Gamma; \Delta \vdash h(x_1, \dots, x_n)$, where $\Delta = x_1 : t_1 \dots x_n : t_n$. By induction hypothesis, terms $e_1, \dots e_n$ are canonical and consequently $\delta : \Delta \to E$ defined by $\delta(x_i) = e_i$ is canonical. Moreover, $h(e_1 \dots e_n) = h(x_1\delta\downarrow, \dots, x_n\delta\downarrow)$ since $e_1, \dots e_n$ are in normal form. But every normalized canonical instance of a complete definition is reducible, which contradicts the assumption that $e = h(e_1, ..., e_n)$ is in normal form.

G Second phase of the algorithm

Example 5. Let `g` be a function defined by case analysis and let `g'` be its version defined by rewriting (without the impossible `Leaf` case):

```
Definition g (t : tree (S (S O))) : bool := match t with
  Leaf => false
| Node _ Leaf _ _ _ => true
| Node _ (Node _ _ _ _ _) _ _ _ => false
end.

Symbol g' : tree (S (S O)) → bool
Rules
  g' (Node ?1 Leaf b ?2 t') ⟶ true
  g' (Node ?1 (Node ?2 t b ?3 t') b' ?4 t'') ⟶ false
```

Our algorithm starts with the goal `g' x`, splits it along `x`, easily detects that `Leaf` case is not possible but is stuck on `Node n t b m t'`, because this requires deciding that `S (n+m)` is unifiable with `S (S O)`, which may be too hard for a unification algorithm[2]. In that case the initial goal `g' x` becomes a potential counterexample.

Accepting all definitions by case analysis. The second phase of our algorithm would start only for the goals with safe splitting variables, i.e. where regular splitting failed because the unification was too weak. In this phase, the splitting would become lax by treating ? unification result as successful and returning trivial substitutions $\sigma_c = \{x \mapsto c(\vec{z})\}$ for such cases (see Def. 5). As a result the goals would not be well-typed sequents anymore, which has to be taken into account by the unification algorithm. On the other hand typability is not required for splitting matching and the rest of the algorithm which would work just like described in Def. 7. Both arguments of termination and correctness of the algorithm would hold.

Going back to our example. Redoing the lax splitting on `x` in the goal `g' x`, one gets again that `Leaf` is impossible, but `Node` is now accepted and leads to an (untyped) goal `g' (Node n t b m t')`. Splitting on `t` is now successful for both constructors and both resulting goals get reduced.

[2] Note that a better unification algorithm could find the two most general solutions `n=O`, `m=(S O)` and `n=(S O)`, `m=O`. Then splitting would result in two goals immediately covered by rules for `g'`.