

# Certified Higher-Order Recursive Path Ordering

... a short story of a never ending formalization

Adam Koprowski

Technical University Eindhoven  
Department of Mathematics and Computer Science

13 August 2006  
RTA, Seattle, USA

## Abstract

The paper reports on a formalization of a proof of well-foundedness of the higher-order recursive path ordering (HORPO) in the proof checker Coq. The development is axiom-free and fully constructive. Three substantive parts that could be used also in other developments are the formalizations of the simply-typed lambda calculus, of finite multisets and of the multiset ordering. The Coq code consists of more than 1000 lemmas and 300 definitions.

## Abstract

The paper reports on a formalization of a proof of well-foundedness of the **higher-order recursive path ordering (HORPO)** in the proof checker Coq. The development is axiom-free and fully constructive. Three substantive parts that could be used also in other developments are the formalizations of the simply-typed lambda calculus, of finite multisets and of the multiset ordering. The Coq code consists of more than 1000 lemmas and 300 definitions.

## Abstract

The paper reports on a formalization of a **proof of well-foundedness** of the higher-order recursive path ordering (HORPO) in the proof checker Coq. The development is axiom-free and fully constructive. Three substantive parts that could be used also in other developments are the formalizations of the simply-typed lambda calculus, of finite multisets and of the multiset ordering. The Coq code consists of more than 1000 lemmas and 300 definitions.

## Abstract

The paper reports on a **formalization** of a proof of well-foundedness of the higher-order recursive path ordering (HORPO) **in the proof checker Coq**. The development is axiom-free and fully constructive. Three substantive parts that could be used also in other developments are the formalizations of the simply-typed lambda calculus, of finite multisets and of the multiset ordering. The Coq code consists of more than 1000 lemmas and 300 definitions.

## Abstract

The paper reports on a formalization of a proof of well-foundedness of the higher-order recursive path ordering (HORPO) in the proof checker Coq. **The development is axiom-free and fully constructive.** Three substantive parts that could be used also in other developments are the formalizations of the simply-typed lambda calculus, of finite multisets and of the multiset ordering. The Coq code consists of more than 1000 lemmas and 300 definitions.

## Abstract

The paper reports on a formalization of a proof of well-foundedness of the higher-order recursive path ordering (HORPO) in the proof checker Coq. The development is axiom-free and fully constructive. Three substantive parts that could be used also in other developments are the formalizations of the **simply-typed lambda calculus**, of **finite multisets** and of the **multiset ordering**. The Coq code consists of more than 1000 lemmas and 300 definitions.

# Outline

- 1 **Background**
  - Termination of rewriting
    - Recursive path ordering (RPO)
    - Higher-order rewriting
    - Higher-order recursive path ordering
- 2 **Formalization**
  - History
  - Motivation & Goals
  - Overview
- 3 **Conclusions**

# Outline

- 1 Background
  - Termination of rewriting
  - Recursive path ordering (RPO)
  - Higher-order rewriting
  - Higher-order recursive path ordering
- 2 Formalization
  - History
  - Motivation & Goals
  - Overview
- 3 Conclusions

# Outline

- 1 **Background**
  - Termination of rewriting
  - Recursive path ordering (RPO)
  - Higher-order rewriting
  - Higher-order recursive path ordering
- 2 **Formalization**
  - History
  - Motivation & Goals
  - Overview
- 3 **Conclusions**

# Outline

- 1 Background
  - Termination of rewriting
  - Recursive path ordering (RPO)
  - Higher-order rewriting
  - Higher-order recursive path ordering
- 2 Formalization
  - History
  - Motivation & Goals
  - Overview
- 3 Conclusions

# Outline

- 1 Background
  - Termination of rewriting
  - Recursive path ordering (RPO)
  - Higher-order rewriting
  - Higher-order recursive path ordering
- 2 Formalization
  - History
  - Motivation & Goals
  - Overview
- 3 Conclusions

# Outline

- 1 Background
  - Termination of rewriting
  - Recursive path ordering (RPO)
  - Higher-order rewriting
  - Higher-order recursive path ordering
- 2 Formalization
  - History
  - Motivation & Goals
  - Overview
- 3 Conclusions

# Outline

- 1 Background
  - Termination of rewriting
  - Recursive path ordering (RPO)
  - Higher-order rewriting
  - Higher-order recursive path ordering
- 2 Formalization
  - History
  - Motivation & Goals
  - Overview
- 3 Conclusions

# Outline

- 1 Background
  - Termination of rewriting
    - Recursive path ordering (RPO)
    - Higher-order rewriting
    - Higher-order recursive path ordering
- 2 Formalization
- 3 Conclusions

# Termination of rewriting

- **Termination is an important concept in term rewriting**
- ... but in general it is undecidable
- ... but there are many techniques for proving termination.
- The simplest approach is embedding into a well-founded partial order.

$$t \rightarrow_R u \implies \phi(t) > \phi(u) \quad (> \text{ well - founded})$$

- The simplest class of techniques are direct techniques using reduction orderings.
- Those are basic techniques but important nevertheless as they are heavily used in more complex transformational techniques.

# Termination of rewriting

- Termination is an important concept in term rewriting
- ... but in general it is undecidable
- ... but there are many techniques for proving termination.
- The simplest approach is embedding into a well-founded partial order.

$$t \rightarrow_R u \implies \phi(t) > \phi(u) \quad (> \text{ well - founded})$$

- The simplest class of techniques are direct techniques using reduction orderings.
- Those are basic techniques but important nevertheless as they are heavily used in more complex transformational techniques.

# Termination of rewriting

- Termination is an important concept in term rewriting
- ... but in general it is undecidable
- ... but there are many techniques for proving termination.
- The simplest approach is embedding into a well-founded partial order.

$$t \rightarrow_R u \implies \phi(t) > \phi(u) \quad (> \text{ well - founded})$$

- The simplest class of techniques are direct techniques using reduction orderings.
- Those are basic techniques but important nevertheless as they are heavily used in more complex transformational techniques.

# Termination of rewriting

- Termination is an important concept in term rewriting
- ... but in general it is undecidable
- ... but there are many techniques for proving termination.
- **The simplest approach is embedding into a well-founded partial order.**

$$t \rightarrow_R u \implies \phi(t) > \phi(u) \quad (> \text{ well - founded})$$

- The simplest class of techniques are direct techniques using reduction orderings.
- Those are basic techniques but important nevertheless as they are heavily used in more complex transformational techniques.

# Termination of rewriting

- Termination is an important concept in term rewriting
- ... but in general it is undecidable
- ... but there are many techniques for proving termination.
- The simplest approach is embedding into a well-founded partial order.

$$t \rightarrow_R u \implies \phi(t) > \phi(u) \quad (> \text{ well - founded})$$

- **The simplest class of techniques are direct techniques using reduction orderings.**
- Those are basic techniques but important nevertheless as they are heavily used in more complex transformational techniques.

# Termination of rewriting

- Termination is an important concept in term rewriting
- ... but in general it is undecidable
- ... but there are many techniques for proving termination.
- The simplest approach is embedding into a well-founded partial order.

$$t \rightarrow_R u \implies \phi(t) > \phi(u) \quad (> \text{ well - founded})$$

- The simplest class of techniques are direct techniques using reduction orderings.
- **Those are basic techniques but important nevertheless as they are heavily used in more complex transformational techniques.**

# Reduction orderings

## Definition

A strict order  $>$  on  $\mathcal{T}(\Sigma, \mathcal{V})$  is called a **reduction ordering** iff it is:

- monotonic

$$t > u \implies f(\dots, t, \dots) > f(\dots, u, \dots)$$

- stable

$$t > u \implies t\sigma > u\sigma$$

- well-founded

## Theorem

*A term rewriting system  $R$  is terminating iff there exist a reduction ordering  $>$  that satisfied  $\ell > r$  for all  $\ell \rightarrow r \in R$ .*

# Reduction orderings

## Definition

A strict order  $>$  on  $\mathcal{T}(\Sigma, \mathcal{V})$  is called a reduction ordering iff it is:

- **monotonic**

$$t > u \implies f(\dots, t, \dots) > f(\dots, u, \dots)$$

- stable

$$t > u \implies t\sigma > u\sigma$$

- well-founded

## Theorem

*A term rewriting system  $R$  is terminating iff there exist a reduction ordering  $>$  that satisfied  $\ell > r$  for all  $\ell \rightarrow r \in R$ .*

# Reduction orderings

## Definition

A strict order  $>$  on  $\mathcal{T}(\Sigma, \mathcal{V})$  is called a reduction ordering iff it is:

- monotonic

$$t > u \implies f(\dots, t, \dots) > f(\dots, u, \dots)$$

- stable

$$t > u \implies t\sigma > u\sigma$$

- well-founded

## Theorem

*A term rewriting system  $R$  is terminating iff there exist a reduction ordering  $>$  that satisfied  $\ell > r$  for all  $\ell \rightarrow r \in R$ .*

# Reduction orderings

## Definition

A strict order  $>$  on  $\mathcal{T}(\Sigma, \mathcal{V})$  is called a reduction ordering iff it is:

- monotonic

$$t > u \implies f(\dots, t, \dots) > f(\dots, u, \dots)$$

- stable

$$t > u \implies t\sigma > u\sigma$$

- **well-founded**

## Theorem

*A term rewriting system  $R$  is terminating iff there exist a reduction ordering  $>$  that satisfied  $\ell > r$  for all  $\ell \rightarrow r \in R$ .*

# Reduction orderings

## Definition

A strict order  $>$  on  $\mathcal{T}(\Sigma, \mathcal{V})$  is called a reduction ordering iff it is:

- monotonic

$$t > u \implies f(\dots, t, \dots) > f(\dots, u, \dots)$$

- stable

$$t > u \implies t\sigma > u\sigma$$

- well-founded

## Theorem

*A term rewriting system  $R$  is terminating iff there exist a reduction ordering  $>$  that satisfied  $\ell > r$  for all  $\ell \rightarrow r \in R$ .*

# Outline

- 1 **Background**
  - Termination of rewriting
  - **Recursive path ordering (RPO)**
  - Higher-order rewriting
  - Higher-order recursive path ordering
- 2 Formalization
- 3 Conclusions

# Recursive path ordering (RPO)

## Definition (RPO, Dershowitz 1982)

Given order on function symbols  $\triangleright$  called **precedence** and a **status** we define the **RPO ordering**  $\succ_{rpo}$  as follows:

$$s = f(s_1, \dots, s_n) \succ_{rpo} g(t_1, \dots, t_m) = t \iff$$

- 1  $s_i \succeq_{rpo} t$  for some  $1 \leq i \leq n$ .
- 2  $f \triangleright g$  and  $s \succ_{rpo} t_i$  for all  $1 \leq i \leq m$
- 3  $f = g$  and  $(s_1, \dots, s_n) \succ_{rpo}^{\tau(f)} (t_1, \dots, t_m)$

## Theorem

RPO is a **reduction ordering**.

# Recursive path ordering (RPO)

## Definition (RPO, Dershowtiz 1982)

Given order on function symbols  $\triangleright$  called precedence and a status we define the RPO ordering  $\succ_{rpo}$  as follows:

$$s = f(s_1, \dots, s_n) \succ_{rpo} g(t_1, \dots, t_m) = t \iff$$

- 1  $s_j \succ_{rpo} t$  for some  $1 \leq i \leq n$ .
- 2  $f \triangleright g$  and  $s \succ_{rpo} t_i$  for all  $1 \leq i \leq m$
- 3  $f = g$  and  $(s_1, \dots, s_n) \succ_{rpo}^{\tau(f)} (t_1, \dots, t_m)$

## Theorem

RPO is a *reduction ordering*.

# Recursive path ordering (RPO)

## Definition (RPO, Dershowtiz 1982)

Given order on function symbols  $\triangleright$  called precedence and a status we define the RPO ordering  $\succ_{rpo}$  as follows:

$$s = f(s_1, \dots, s_n) \succ_{rpo} g(t_1, \dots, t_m) = t \iff$$

- 1  $s_i \succeq_{rpo} t$  for some  $1 \leq i \leq n$ .
- 2  $f \triangleright g$  and  $s \succ_{rpo} t_i$  for all  $1 \leq i \leq m$
- 3  $f = g$  and  $(s_1, \dots, s_n) \succ_{rpo}^{\tau(f)} (t_1, \dots, t_m)$

## Theorem

RPO is a *reduction ordering*.

# Recursive path ordering (RPO)

## Definition (RPO, Dershowitz 1982)

Given order on function symbols  $\triangleright$  called precedence and a status we define the RPO ordering  $\succ_{rpo}$  as follows:

$$s = f(s_1, \dots, s_n) \succ_{rpo} g(t_1, \dots, t_m) = t \iff$$

- 1  $s_i \succeq_{rpo} t$  for some  $1 \leq i \leq n$ .
- 2  $f \triangleright g$  and  $s \succ_{rpo} t_i$  for all  $1 \leq i \leq m$
- 3  $f = g$  and  $(s_1, \dots, s_n) \succ_{rpo}^{\tau(f)} (t_1, \dots, t_m)$

## Theorem

RPO is a *reduction ordering*.

# Recursive path ordering (RPO)

## Definition (RPO, Dershowitz 1982)

Given order on function symbols  $\triangleright$  called precedence and a status we define the RPO ordering  $\succ_{rpo}$  as follows:

$$s = f(s_1, \dots, s_n) \succ_{rpo} g(t_1, \dots, t_m) = t \iff$$

- 1  $s_i \succeq_{rpo} t$  for some  $1 \leq i \leq n$ .
- 2  $f \triangleright g$  and  $s \succ_{rpo} t_i$  for all  $1 \leq i \leq m$
- 3  $f = g$  and  $(s_1, \dots, s_n) \succ_{rpo}^{\tau(f)} (t_1, \dots, t_m)$

## Theorem

RPO is a *reduction ordering*.

# Outline

- 1 **Background**
  - Termination of rewriting
  - Recursive path ordering (RPO)
  - **Higher-order rewriting**
  - Higher-order recursive path ordering
- 2 Formalization
- 3 Conclusions

# Higher-order rewriting

There are three variants of higher-order rewriting:

HRS Higher-order rewriting systems (Nipkow)

→ [Higher-Order Rewriting Systems](#)

→ [Higher-Order Rewriting Systems](#)

AFS Algebraic functional systems (Jouannaud and Okada)

→ [Algebraic Functional Systems](#)

→ [Algebraic Functional Systems](#)

CRS Combinatory reduction systems (Klop)

→ [Combinatory Reduction Systems](#)

In this talk we concentrate on AFSs.

# Higher-order rewriting

There are three variants of higher-order rewriting:

## HRS Higher-order rewriting systems (Nipkow)

- $\lambda \rightarrow$  terms.
- Rules restricted to patterns.
- Rewriting modulo  $\beta\eta$ .

## AFS Algebraic functional systems (Jouannaud and Okada)

- Algebraic terms with arity
- Plain pattern matching

## CRS Combinatory reduction systems (Klop)

- Can be encoded via the other two.

In this talk we concentrate on AFSs.

# Higher-order rewriting

There are three variants of higher-order rewriting:

**HRS** Higher-order rewriting systems (Nipkow)

- $\lambda \rightarrow$  terms.
- Rules restricted to patterns.
- Rewriting modulo  $\beta\eta$ .

**AFS** Algebraic functional systems (Jouannaud and Okada)

- Algebraic terms with arity
- Plain pattern matching

**CRS** Combinatory reduction systems (Klop)

- Can be encoded via the other two.

In this talk we concentrate on AFSs.

# Higher-order rewriting

There are three variants of higher-order rewriting:

**HRS** Higher-order rewriting systems (Nipkow)

- $\lambda \rightarrow$  terms.
- **Rules restricted to patterns.**
- Rewriting modulo  $\beta\eta$ .

**AFS** Algebraic functional systems (Jouannaud and Okada)

- Algebraic terms with arity
- Plain pattern matching

**CRS** Combinatory reduction systems (Klop)

- Can be encoded via the other two.

In this talk we concentrate on AFSs.

# Higher-order rewriting

There are three variants of higher-order rewriting:

**HRS** Higher-order rewriting systems (Nipkow)

- $\lambda \rightarrow$  terms.
- Rules restricted to patterns.
- **Rewriting modulo  $\beta\eta$ .**

**AFS** Algebraic functional systems (Jouannaud and Okada)

- Algebraic terms with arity
- Plain pattern matching

**CRS** Combinatory reduction systems (Klop)

- Can be encoded via the other two.

In this talk we concentrate on AFSs.

# Higher-order rewriting

There are three variants of higher-order rewriting:

**HRS** Higher-order rewriting systems (Nipkow)

- $\lambda \rightarrow$  terms.
- Rules restricted to patterns.
- Rewriting modulo  $\beta\eta$ .

**AFS** Algebraic functional systems (Jouannaud and Okada)

- Algebraic terms with arity
- Plain pattern matching

**CRS** Combinatory reduction systems (Klop)

- Can be encoded via the other two.

In this talk we concentrate on AFSs.

# Higher-order rewriting

There are three variants of higher-order rewriting:

**HRS** Higher-order rewriting systems (Nipkow)

- $\lambda \rightarrow$  terms.
- Rules restricted to patterns.
- Rewriting modulo  $\beta\eta$ .

**AFS** Algebraic functional systems (Jouannaud and Okada)

- Algebraic terms with arity
- Plain pattern matching

**CRS** Combinatory reduction systems (Klop)

- Can be encoded via the other two.

In this talk we concentrate on AFSs.

# Higher-order rewriting

There are three variants of higher-order rewriting:

**HRS** Higher-order rewriting systems (Nipkow)

- $\lambda \rightarrow$  terms.
- Rules restricted to patterns.
- Rewriting modulo  $\beta\eta$ .

**AFS** Algebraic functional systems (Jouannaud and Okada)

- Algebraic terms with arity
- **Plain pattern matching**

**CRS** Combinatory reduction systems (Klop)

- Can be encoded via the other two.

In this talk we concentrate on AFSs.

# Higher-order rewriting

There are three variants of higher-order rewriting:

**HRS** Higher-order rewriting systems (Nipkow)

- $\lambda \rightarrow$  terms.
- Rules restricted to patterns.
- Rewriting modulo  $\beta\eta$ .

**AFS** Algebraic functional systems (Jouannaud and Okada)

- Algebraic terms with arity
- Plain pattern matching

**CRS** Combinatory reduction systems (Klop)

- Can be encoded via the other two.

In this talk we concentrate on AFSs.

# Higher-order rewriting

There are three variants of higher-order rewriting:

**HRS** Higher-order rewriting systems (Nipkow)

- $\lambda \rightarrow$  terms.
- Rules restricted to patterns.
- Rewriting modulo  $\beta\eta$ .

**AFS** Algebraic functional systems (Jouannaud and Okada)

- Algebraic terms with arity
- Plain pattern matching

**CRS** Combinatory reduction systems (Klop)

- **Can be encoded via the other two.**

In this talk we concentrate on AFSs.

# Higher-order rewriting

There are three variants of higher-order rewriting:

**HRS** Higher-order rewriting systems (Nipkow)

- $\lambda \rightarrow$  terms.
- Rules restricted to patterns.
- Rewriting modulo  $\beta\eta$ .

**AFS** Algebraic functional systems (Jouannaud and Okada)

- Algebraic terms with arity
- Plain pattern matching

**CRS** Combinatory reduction systems (Klop)

- Can be encoded via the other two.

In this talk we concentrate on AFSs.

# Higher-order terms

## Definition (Simple types, $\mathcal{T}_S$ )

Given a set of **sorts**  $\mathcal{S}$  we inductively define a set of **simple types** as follows:

$$\mathcal{T}_S ::= \mathcal{S} \mid \mathcal{T}_S \rightarrow \mathcal{T}_S$$

## Definition (Signature, $\mathcal{F}$ )

A signature is a set of function declarations of the shape:

$$f: \alpha_1 \times \dots \times \alpha_n \rightarrow \beta$$

## Definition (Environment, $\mathcal{E}$ )

Environment is a finite set of distinct variable declarations:

$$\mathcal{E} \subset \mathcal{V} \times \mathcal{T}_S$$

# Higher-order terms

## Definition (Simple types, $\mathcal{T}_S$ )

Given a set of sorts  $\mathcal{S}$  we inductively define a set of simple types as follows:

$$\mathcal{T}_S ::= \mathcal{S} \mid \mathcal{T}_S \rightarrow \mathcal{T}_S$$

## Definition (Signature, $\mathcal{F}$ )

A **signature** is a set of function declarations of the shape:

$$f : \alpha_1 \times \dots \times \alpha_n \rightarrow \beta$$

## Definition (Environment, $\mathcal{E}$ )

Environment is a finite set of distinct variable declarations:

$$\mathcal{E} \subset \mathcal{V} \times \mathcal{T}_S$$

# Higher-order terms

## Definition (Simple types, $\mathcal{T}_S$ )

Given a set of sorts  $\mathcal{S}$  we inductively define a set of simple types as follows:

$$\mathcal{T}_S ::= \mathcal{S} \mid \mathcal{T}_S \rightarrow \mathcal{T}_S$$

## Definition (Signature, $\mathcal{F}$ )

A signature is a set of function declarations of the shape:

$$f : \alpha_1 \times \dots \times \alpha_n \rightarrow \beta$$

## Definition (Environment, $\mathcal{E}$ )

**Environment** is a finite set of distinct variable declarations:

$$\mathcal{E} \subset \mathcal{V} \times \mathcal{T}_S$$

# Higher-order terms cont.

## Definition (Algebraic terms)

A set of **algebraic terms** is defined by the following grammar:

$$Pt := \nu \mid @ (Pt, Pt) \mid \lambda \nu : \mathcal{I}_S. Pt \mid \mathcal{F}(Pt, \dots, Pt)$$

## Definition (Typing rules)

$$\frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}$$

$$\frac{\Gamma \vdash t : \alpha \rightarrow \beta \quad \Gamma \vdash u : \alpha}{\Gamma \vdash @(t, u) : \beta}$$

$$\frac{f : \alpha_1 \times \dots \times \alpha_n \rightarrow \beta \in \Sigma \quad \Gamma \vdash t_1 : \alpha_1, \dots, \Gamma \vdash t_n : \alpha_n}{\Gamma \vdash f(t_1, \dots, t_n) : \beta}$$

$$\frac{\Gamma \cup \{x : \alpha\} \vdash t : \beta}{\Gamma \vdash \lambda x : \alpha. t : \alpha \rightarrow \beta}$$

# Higher-order terms cont.

## Definition (Algebraic terms)

A set of algebraic terms is defined by the following grammar:

$$\mathcal{P}t := \nu \mid @(\mathcal{P}t, \mathcal{P}t) \mid \lambda\nu : \mathcal{I}_S. \mathcal{P}t \mid \mathcal{F}(\mathcal{P}t, \dots, \mathcal{P}t)$$

## Definition (Typing rules)

$$\frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}$$

$$\frac{\Gamma \vdash t : \alpha \rightarrow \beta \quad \Gamma \vdash u : \alpha}{\Gamma \vdash @(t, u) : \beta}$$

$$\frac{f : \alpha_1 \times \dots \times \alpha_n \rightarrow \beta \in \Sigma \quad \Gamma \vdash t_1 : \alpha_1, \dots, \Gamma \vdash t_n : \alpha_n}{\Gamma \vdash f(t_1, \dots, t_n) : \beta}$$

$$\frac{\Gamma \cup \{x : \alpha\} \vdash t : \beta}{\Gamma \vdash \lambda x : \alpha. t : \alpha \rightarrow \beta}$$

# Examples of higher-order rewriting

## Example (AFS for map)

$$\mathcal{S} = \{\text{List}, \text{Nat}\}$$

$$\text{nil} : \text{List}$$

$$\text{cons} : \text{Nat} \times \text{List} \rightarrow \text{List}$$

$$\text{map} : \text{List} \times (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{List}$$

$$\text{map}(\text{nil}, F) \rightarrow \text{nil}$$

$$\text{map}(\text{cons}(x, l), F) \rightarrow \text{cons}(@ (F, x), \text{map}(l, F))$$

## Example (AFS for summation)

Function  $\Sigma(n, F)$  computes  $\sum_{0 \leq i \leq n} F(i)$ .

$$\Sigma(0, F) \rightarrow @(F, 0)$$

$$\Sigma(s(n), F) \rightarrow +(\Sigma(n, F), @(F, s(n)))$$

# Examples of higher-order rewriting

## Example (AFS for map)

$$\begin{aligned}\mathcal{S} &= \{\text{List}, \text{Nat}\} \\ \text{nil} &: \text{List} \\ \text{cons} &: \text{Nat} \times \text{List} \rightarrow \text{List} \\ \text{map} &: \text{List} \times (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{List} \\ \\ \text{map}(\text{nil}, F) &\rightarrow \text{nil} \\ \text{map}(\text{cons}(x, l), F) &\rightarrow \text{cons}(@F, x), \text{map}(l, F)\end{aligned}$$

## Example (AFS for summation)

Function  $\Sigma(n, F)$  computes  $\sum_{0 \leq i \leq n} F(i)$ .

$$\begin{aligned}\Sigma(0, F) &\rightarrow @(F, 0) \\ \Sigma(s(n), F) &\rightarrow +(\Sigma(n, F), @(F, s(n)))\end{aligned}$$

# Examples of higher-order rewriting

## Example (AFS for map)

$$\begin{aligned}\mathcal{S} &= \{\text{List}, \text{Nat}\} \\ \text{nil} &: \text{List} \\ \text{cons} &: \text{Nat} \times \text{List} \rightarrow \text{List} \\ \text{map} &: \text{List} \times (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{List} \\ \\ \text{map}(\text{nil}, F) &\rightarrow \text{nil} \\ \text{map}(\text{cons}(x, l), F) &\rightarrow \text{cons}(@F, x), \text{map}(l, F)\end{aligned}$$

## Example (AFS for summation)

Function  $\Sigma(n, F)$  computes  $\sum_{0 \leq i \leq n} F(i)$ .

$$\begin{aligned}\Sigma(0, F) &\rightarrow @F, 0 \\ \Sigma(s(n), F) &\rightarrow +(\Sigma(n, F), @F, s(n))\end{aligned}$$

# Higher-order reduction orderings

## Definition

A strict order  $>$  on  $\mathcal{T}(\Sigma, \mathcal{V})$  is called a **higher-order reduction ordering** iff it is:

- monotonic  $t > u \implies s[t]_\rho > s[u]_\rho$
- stable  $t > u \implies t\sigma > u\sigma$
- coherent

$$\left. \begin{array}{l} \Gamma \vdash s : \delta > \Gamma \vdash t : \delta \\ \Delta \rightsquigarrow \Gamma \\ \Delta \vdash s : \delta \\ \Delta \vdash t : \delta \end{array} \right\} \implies \Delta \vdash s : \delta > \Delta \vdash t : \delta$$

- functional  $t \rightarrow_\beta u \implies t > u$
- well-founded

# Higher-order reduction orderings

## Definition

A strict order  $>$  on  $\mathcal{T}(\Sigma, \mathcal{V})$  is called a higher-order reduction ordering iff it is:

- **monotonic**  $t > u \implies s[t]_p > s[u]_p$

- **stable**  $t > u \implies t\sigma > u\sigma$

- **coherent**

$$\left. \begin{array}{l} \Gamma \vdash s : \delta > \Gamma \vdash t : \delta \\ \Delta \rightsquigarrow \Gamma \\ \Delta \vdash s : \delta \\ \Delta \vdash t : \delta \end{array} \right\} \implies \Delta \vdash s : \delta > \Delta \vdash t : \delta$$

- **functional**  $t \rightarrow_{\beta} u \implies t > u$

- **well-founded**

# Higher-order reduction orderings

## Definition

A strict order  $>$  on  $\mathcal{T}(\Sigma, \mathcal{V})$  is called a higher-order reduction ordering iff it is:

- monotonic  $t > u \implies s[t]_p > s[u]_p$
- **stable**  $t > u \implies t\sigma > u\sigma$

- coherent

$$\left. \begin{array}{l} \Gamma \vdash s : \delta > \Gamma \vdash t : \delta \\ \Delta \rightsquigarrow \Gamma \\ \Delta \vdash s : \delta \\ \Delta \vdash t : \delta \end{array} \right\} \implies \Delta \vdash s : \delta > \Delta \vdash t : \delta$$

- functional  $t \rightarrow_{\beta} u \implies t > u$
- well-founded

# Higher-order reduction orderings

## Definition

A strict order  $>$  on  $\mathcal{T}(\Sigma, \mathcal{V})$  is called a higher-order reduction ordering iff it is:

- monotonic  $t > u \implies s[t]_p > s[u]_p$
- stable  $t > u \implies t\sigma > u\sigma$

- **coherent**

$$\left. \begin{array}{l} \Gamma \vdash s : \delta > \Gamma \vdash t : \delta \\ \Delta \leftrightarrow \Gamma \\ \Delta \vdash s : \delta \\ \Delta \vdash t : \delta \end{array} \right\} \implies \Delta \vdash s : \delta > \Delta \vdash t : \delta$$

- functional  $t \rightarrow_{\beta} u \implies t > u$
- well-founded

# Higher-order reduction orderings

## Definition

A strict order  $>$  on  $\mathcal{T}(\Sigma, \mathcal{V})$  is called a higher-order reduction ordering iff it is:

- monotonic  $t > u \implies s[t]_p > s[u]_p$
- stable  $t > u \implies t\sigma > u\sigma$
- coherent

$$\left. \begin{array}{l} \Gamma \vdash s : \delta > \Gamma \vdash t : \delta \\ \Delta \leftrightarrow \Gamma \\ \Delta \vdash s : \delta \\ \Delta \vdash t : \delta \end{array} \right\} \implies \Delta \vdash s : \delta > \Delta \vdash t : \delta$$

- **functional**  $t \rightarrow_{\beta} u \implies t > u$
- well-founded

# Higher-order reduction orderings

## Definition

A strict order  $>$  on  $\mathcal{T}(\Sigma, \mathcal{V})$  is called a higher-order reduction ordering iff it is:

- monotonic  $t > u \implies s[t]_p > s[u]_p$
- stable  $t > u \implies t\sigma > u\sigma$
- coherent

$$\left. \begin{array}{l} \Gamma \vdash s : \delta > \Gamma \vdash t : \delta \\ \Delta \leftrightarrow \Gamma \\ \Delta \vdash s : \delta \\ \Delta \vdash t : \delta \end{array} \right\} \implies \Delta \vdash s : \delta > \Delta \vdash t : \delta$$

- functional  $t \rightarrow_{\beta} u \implies t > u$
- **well-founded**

# Reduction orderings

## First-order

- monotonic

$$t > u \implies f(\dots, t, \dots) > f(\dots, u, \dots)$$

- stable

$$t > u \implies t\sigma > u\sigma$$

- well-founded

## Higher-order

- monotonic

$$t > u \implies s[t]_p > s[u]_p$$

- stable

$$t > u \implies t\sigma > u\sigma$$

- coherent

$$\begin{aligned} \Gamma \vdash s : \delta > \Gamma \vdash t : \delta \wedge \dots \\ \implies \Delta \vdash s : \delta > \Delta \vdash t : \delta \end{aligned}$$

- functional

$$t \rightarrow_{\beta} u \implies t > u$$

- well-founded

# Reduction orderings

## Higher-order

- monotonic

$$t > u \implies s[t]_p > s[u]_p$$

- stable

$$t > u \implies t\sigma > u\sigma$$

- coherent

$$\begin{aligned} & \Gamma \vdash s : \delta > \Gamma \vdash t : \delta \wedge \dots \\ & \implies \Delta \vdash s : \delta > \Delta \vdash t : \delta \end{aligned}$$

- functional

$$t \rightarrow_\beta u \implies t > u$$

- well-founded

## Formalization

- monotonic

$$t > u \implies s[t]_p > s[u]_p$$

- stable

$$t > u \implies t\sigma > u\sigma$$

- coherent

$$\begin{aligned} & \Gamma \vdash s : \delta > \Gamma \vdash t : \delta \wedge \dots \\ & \implies \Delta \vdash s : \delta > \Delta \vdash t : \delta \end{aligned}$$

- $> \cup \rightarrow_\beta$  **well-founded**

# Outline

- 1 **Background**
  - Termination of rewriting
  - Recursive path ordering (RPO)
  - Higher-order rewriting
  - **Higher-order recursive path ordering**
- 2 Formalization
- 3 Conclusions

# Higher-order recursive path ordering

## Definition (HORPO, Jouannaud and Rubio)

$\Gamma \vdash t : \delta \succ \Gamma \vdash u : \delta$  iff one of the following holds:

- 1  $t = f(t_1, \dots, t_n), \exists i \in \{1, \dots, n\} . t_i \succeq u$
- 2  $t = f(t_1, \dots, t_n), u = g(u_1, \dots, u_k), f \triangleright g, t \succcurlyeq \{u_1, \dots, u_k\}$
- 3  $t = f(t_1, \dots, t_n), u = f(u_1, \dots, u_k),$   
 $\{\{t_1, \dots, t_n\}\} \succ_{mul} \{\{u_1, \dots, u_k\}\}$
- 4  $\textcircled{u}_1, \dots, u_k$  is a partial flattening of  $u, t \succcurlyeq \{u_1, \dots, u_k\}$
- 5  $t = \textcircled{t}_l, u_r, u = \textcircled{t}_l, u_r, \{\{t_l, t_r\}\} \succ_{mul} \{\{u_l, u_r\}\}$
- 6  $t = \lambda x : \alpha . t', u = \lambda x : \alpha . u', t' \succ u'$

where  $\succcurlyeq$  is defined as:

$t = f(t_1, \dots, t_k) \succcurlyeq \{u_1, \dots, u_n\}$  iff  
 $\forall i \in \{1, \dots, n\} . t \succ u_i \vee (\exists j . t_j \succeq u_i).$

# HORPO in the formalization

Differences between HORPO as defined in [1] ( $\succ_{JR}$ ) and the formalized variant ( $\succ$ ).

- $\succ$  can only compare terms of equal types whereas in  $\succ_{JR}$  terms of equivalent type can be compared
- $\succ_{JR}$  uses statuses (lexicographic / multiset) whereas in  $\succ$  arguments can be compared only as multisets.
- $\succ$  uses a different variant of a definition of the multiset extension of a relation.
- $\succ$  assumes ground output type for functions.



J.-P. Jouannaud and A. Rubio.

The higher-order recursive path ordering.

In *LICS '99*, pages 402–411, Trento, Italy, July 1999.

# HORPO in the formalization

Differences between HORPO as defined in [1] ( $\succ_{JR}$ ) and the formalized variant ( $\succ$ ).

- $\succ$  can only compare terms of equal types whereas in  $\succ_{JR}$  terms of equivalent type can be compared
- $\succ_{JR}$  uses statuses (lexicographic / multiset) whereas in  $\succ$  arguments can be compared only as multisets.
- $\succ$  uses a different variant of a definition of the multiset extension of a relation.
- $\succ$  assumes ground output type for functions.



J.-P. Jouannaud and A. Rubio.

The higher-order recursive path ordering.

In *LICS '99*, pages 402–411, Trento, Italy, July 1999.

# HORPO in the formalization

Differences between HORPO as defined in [1] ( $\succ_{JR}$ ) and the formalized variant ( $\succ$ ).

- $\succ$  can only compare terms of equal types whereas in  $\succ_{JR}$  terms of equivalent type can be compared
- $\succ_{JR}$  uses statuses (lexicographic / multiset) whereas in  $\succ$  arguments can be compared only as multisets.
- $\succ$  uses a different variant of a definition of the multiset extension of a relation.
- $\succ$  assumes ground output type for functions.



J.-P. Jouannaud and A. Rubio.

The higher-order recursive path ordering.

In *LICS '99*, pages 402–411, Trento, Italy, July 1999.

# HORPO in the formalization

Differences between HORPO as defined in [1] ( $\succ_{JR}$ ) and the formalized variant ( $\succ$ ).

- $\succ$  can only compare terms of equal types whereas in  $\succ_{JR}$  terms of equivalent type can be compared
- $\succ_{JR}$  uses statuses (lexicographic / multiset) whereas in  $\succ$  arguments can be compared only as multisets.
- $\succ$  uses a different variant of a definition of the multiset extension of a relation.
- $\succ$  **assumes ground output type for functions.**



J.-P. Jouannaud and A. Rubio.

The higher-order recursive path ordering.

In *LICS '99*, pages 402–411, Trento, Italy, July 1999.

# Outline

- 1 Background
- 2 **Formalization**
  - **History**
  - Motivation & Goals
  - Overview
- 3 Conclusions

# Timeline of the project

Two stages of the project:

- Jan 2004 - Jul 2004

- Author's Master Thesis at the Vrije Universiteit supervised by dr Femke van Raamsdonk.
- Proof completed but computability properties as axioms.

- Nov 2004 - Apr 2006

- Development continued at the Technical University Eindhoven.
- Complete, axiom free proof.
- Many improvements on the way.
- The size of Coq scripts tripled.

# Timeline of the project

Two stages of the project:

- Jan 2004 - Jul 2004
  - Author's Master Thesis at the Vrije Universiteit supervised by dr Femke van Raamsdonk.
  - Proof completed but computability properties as axioms.
- Nov 2004 - Apr 2006
  - Development continued at the Technical University Eindhoven.
  - Complete, axiom free proof.
  - Many improvements on the way.
  - The size of Coq scripts tripled.

# Timeline of the project

Two stages of the project:

- Jan 2004 - Jul 2004
  - Author's Master Thesis at the **Vrije Universiteit** supervised by dr **Femke van Raamsdonk**.
  - **Proof completed but computability properties as axioms.**
- Nov 2004 - Apr 2006
  - Development continued at the **Technical University Eindhoven**.
  - Complete, **axiom free proof**.
  - **Many improvements on the way.**
  - The size of Coq scripts tripled.

# Timeline of the project

Two stages of the project:

- Jan 2004 - Jul 2004
  - Author's Master Thesis at the **Vrije Universiteit** supervised by dr **Femke van Raamsdonk**.
  - Proof completed but **computability properties as axioms**.
- **Nov 2004 - Apr 2006**
  - Development continued at the Technical University Eindhoven.
  - Complete, axiom free proof.
  - Many improvements on the way.
  - The size of Coq scripts tripled.

# Timeline of the project

Two stages of the project:

- Jan 2004 - Jul 2004
  - Author's Master Thesis at the **Vrije Universiteit** supervised by dr **Femke van Raamsdonk**.
  - Proof completed but **computability properties as axioms**.
- Nov 2004 - Apr 2006
  - **Development continued at the Technical University Eindhoven.**
  - Complete, **axiom free proof**.
  - **Many improvements** on the way.
  - The size of Coq scripts tripled.

# Timeline of the project

Two stages of the project:

- Jan 2004 - Jul 2004
  - Author's Master Thesis at the **Vrije Universiteit** supervised by dr **Femke van Raamsdonk**.
  - Proof completed but **computability properties as axioms**.
- Nov 2004 - Apr 2006
  - Development continued at the **Technical University Eindhoven**.
  - **Complete, axiom free proof**.
  - **Many improvements** on the way.
  - The size of Coq scripts tripled.

# Timeline of the project

Two stages of the project:

- Jan 2004 - Jul 2004
  - Author's Master Thesis at the **Vrije Universiteit** supervised by dr **Femke van Raamsdonk**.
  - Proof completed but **computability properties as axioms**.
- Nov 2004 - Apr 2006
  - Development continued at the **Technical University Eindhoven**.
  - Complete, **axiom free proof**.
  - **Many improvements on the way**.
  - The size of Coq scripts tripled.

# Timeline of the project

Two stages of the project:

- Jan 2004 - Jul 2004
  - Author's Master Thesis at the **Vrije Universiteit** supervised by dr **Femke van Raamsdonk**.
  - Proof completed but **computability properties as axioms**.
- Nov 2004 - Apr 2006
  - Development continued at the **Technical University Eindhoven**.
  - Complete, **axiom free proof**.
  - **Many improvements** on the way.
  - **The size of Coq scripts tripled**.

# Outline

- 1 Background
- 2 **Formalization**
  - History
  - **Motivation & Goals**
  - Overview
- 3 Conclusions

# Motivation & Goals

## Motivation: Why making such formalization?

- Verification of the theory (especially for complicated, not very well-known proofs).
- Stimulus for improvement of formal methods, in general, and theorem proving, in particular.
- CoLoR: Coq library on rewriting and termination, <http://color.loria.fr>.

Goal: formalization that is:

- complete (axiom-free),
- fully constructive,
- HORPO proof as close as possible to the original one,
- pure  $\lambda^{\rightarrow}$  terms.

# Motivation & Goals

## Motivation: Why making such formalization?

- Verification of the theory (especially for complicated, not very well-known proofs).
- Stimulus for improvement of formal methods, in general, and theorem proving, in particular.
- CoLoR: Coq library on rewriting and termination, <http://color.loria.fr>.

## Goal: formalization that is:

- complete (axiom-free),
- fully constructive,
- HORPO proof as close as possible to the original one,
- pure  $\lambda \rightarrow$  terms.

# Motivation & Goals

**Motivation:** Why making such formalization?

- Verification of the theory (especially for complicated, not very well-known proofs).
- **Stimulus for improvement of formal methods, in general, and theorem proving, in particular.**
- CoLoR: Coq library on rewriting and termination, <http://color.loria.fr>.

**Goal:** formalization that is:

- complete (axiom-free),
- fully constructive,
- HORPO proof as close as possible to the original one,
- pure  $\lambda \rightarrow$  terms.

# Motivation & Goals

**Motivation:** Why making such formalization?

- Verification of the theory (especially for complicated, not very well-known proofs).
- Stimulus for improvement of formal methods, in general, and theorem proving, in particular.
- **CoLoR: Coq library on rewriting and termination**, <http://color.loria.fr>.

**Goal:** formalization that is:

- complete (axiom-free),
- fully constructive,
- HORPO proof as close as possible to the original one,
- pure  $\lambda \rightarrow$  terms.

# Motivation & Goals

**Motivation:** Why making such formalization?

- Verification of the theory (especially for complicated, not very well-known proofs).
- Stimulus for improvement of formal methods, in general, and theorem proving, in particular.
- **CoLoR**: Coq library on rewriting and termination, <http://color.loria.fr>.

**Goal:** formalization that is:

- complete (axiom-free),
- fully constructive,
- HORPO proof as close as possible to the original one,
- pure  $\lambda \rightarrow$  terms.

# Motivation & Goals

**Motivation:** Why making such formalization?

- Verification of the theory (especially for complicated, not very well-known proofs).
- Stimulus for improvement of formal methods, in general, and theorem proving, in particular.
- **CoLoR**: Coq library on rewriting and termination, <http://color.loria.fr>.

**Goal:** formalization that is:

- ✓ **complete (axiom-free)**,
- fully constructive,
- HORPO proof as close as possible to the original one,
- pure  $\lambda \rightarrow$  terms.

# Motivation & Goals

**Motivation:** Why making such formalization?

- Verification of the theory (especially for complicated, not very well-known proofs).
- Stimulus for improvement of formal methods, in general, and theorem proving, in particular.
- **CoLoR**: Coq library on rewriting and termination, <http://color.loria.fr>.

**Goal:** formalization that is:

- ✓ complete (axiom-free),
- ✓ **fully constructive**,
- HORPO proof as close as possible to the original one,
- pure  $\lambda \rightarrow$  terms.

# Motivation & Goals

**Motivation:** Why making such formalization?

- Verification of the theory (especially for complicated, not very well-known proofs).
- Stimulus for improvement of formal methods, in general, and theorem proving, in particular.
- **CoLoR**: Coq library on rewriting and termination, <http://color.loria.fr>.

**Goal:** formalization that is:

- ✓ complete (axiom-free),
- ✓ fully constructive,
- ✓ **HORPO proof as close as possible to the original one,**
- pure  $\lambda \rightarrow$  terms.

# Motivation & Goals

**Motivation:** Why making such formalization?

- Verification of the theory (especially for complicated, not very well-known proofs).
- Stimulus for improvement of formal methods, in general, and theorem proving, in particular.
- **CoLoR**: Coq library on rewriting and termination, <http://color.loria.fr>.

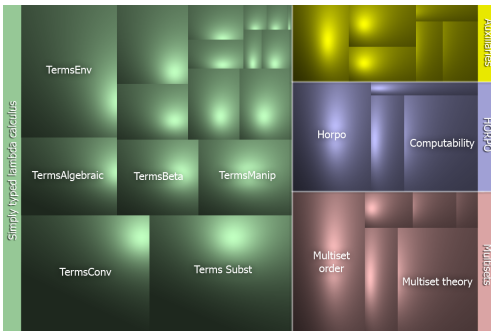
**Goal:** formalization that is:

- ✓ complete (axiom-free),
- ✓ fully constructive,
- ✓ HORPO proof as close as possible to the original one,
- ✓ **pure  $\lambda \rightarrow$  terms.**

# Outline

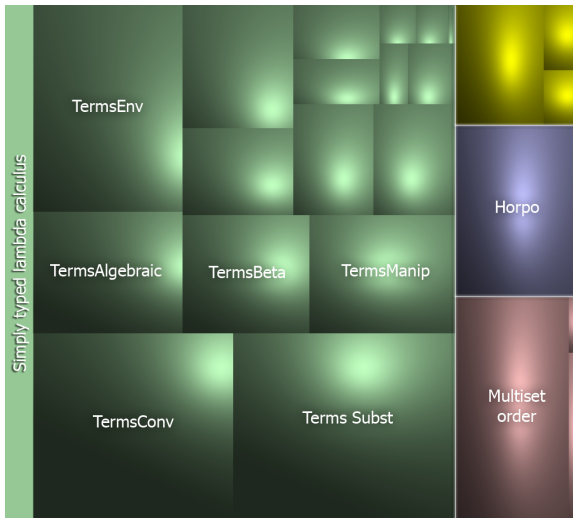
- 1 Background
- 2 **Formalization**
  - History
  - Motivation & Goals
  - **Overview**
- 3 Conclusions

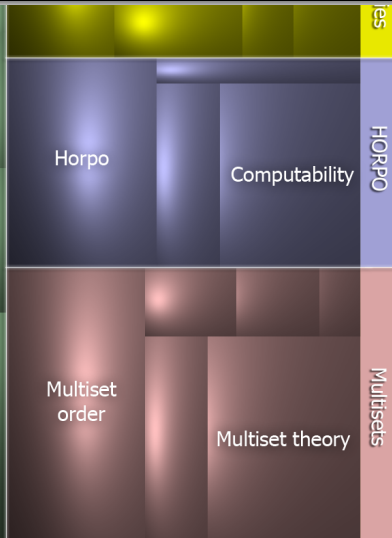
- Terms with typing a'la Church.
- Typing properties: uniqueness of types, decidability of typing.
- Many variable, typed substitution.
- Convertibility relation extending  $\alpha$ -convertibility to free variables.
- $\rightarrow_{\beta}$  and its properties.



- Computability predicate.
- Computability properties.
- Definition of the HORPO.
- Properties of HORPO (inc. decidability).
- Well-foundedness of  $\gamma \sqcup \rightarrow_{\beta}$ .
- Multisets as an abstract data-type.
- Concrete implementation using lists.
- Multiset extensions of a relation.
- Mul. ext. preserves orders.
- Mul. ext. preserves well-foundedness.

- Terms with typing a'la Church.
- Typing properties: uniqueness of types, decidability of typing.
- Many variable, typed substitution.
- Convertibility relation extending  $\alpha$ -convertibility to free variables.
- $\rightarrow_{\beta}$  and its properties.





- Computability predicate.
- Computability properties.
- Definition of the HORPO.
- Properties of HORPO (inc. decidability).
- Well-foundedness of  $\gamma \cup \rightarrow \beta$ .
  
- Multisets as an abstract data-type.
- Concrete implementation using lists.
- Multiset extensions of a relation.
- Mul. ext. preserves orders.
- Mul. ext. preserves well-foundedness.

# Size of the development

- **33 files.**
- > 24,000 lines.
- > 740,000 total characters.
- > 1,100 lemmas.
- > 300 definitions.

# Size of the development

- **33 files.**
- **> 24,000 lines.**
- > 740,000 total characters.
- > 1,100 lemmas.
- > 300 definitions.

# Size of the development

- 33 files.
- > 24,000 lines.
- > 740,000 total characters.
- > 1,100 lemmas.
- > 300 definitions.

# Size of the development

- 33 files.
- > 24,000 lines.
- > 740,000 total characters.
- > 1,100 lemmas.
- > 300 definitions.

# Size of the development

- **33** files.
- **> 24,000** lines.
- **> 740,000** total characters.
- **> 1,100** lemmas.
- **> 300** definitions.

# Evaluation of Coq

- **Big developments in Coq are possible...**
- ... but are still rather time consuming.
- Often simple things turn out not to be that simple (intuition).
- Dependent types are powerful and expressive...
- ... but reasoning about them is difficult.
- Working with equality different than identity is burdensome (although Setoid tactic makes it somehow easier).
- Some form of handling symmetries would be very helpful.

# Evaluation of Coq

- Big developments in Coq are possible...
- ... but are still rather time consuming.
- Often simple things turn out not to be that simple (intuition).
- Dependent types are powerful and expressive...
- ... but reasoning about them is difficult.
- Working with equality different than identity is burdensome (although Setoid tactic makes it somehow easier).
- Some form of handling symmetries would be very helpful.

# Evaluation of Coq

- Big developments in Coq are possible...
- ... but are still rather time consuming.
- **Often simple things turn out not to be that simple (intuition).**
- Dependent types are powerful and expressive...
- ... but reasoning about them is difficult.
- Working with equality different than identity is burdensome (although Setoid tactic makes it somehow easier).
- Some form of handling symmetries would be very helpful.

# Evaluation of Coq

- Big developments in Coq are possible...
- ... but are still rather time consuming.
- Often simple things turn out not to be that simple (intuition).
- **Dependent types are powerful and expressive...**
- ... but reasoning about them is difficult.
- Working with equality different than identity is burdensome (although Setoid tactic makes it somehow easier).
- Some form of handling symmetries would be very helpful.

# Evaluation of Coq

- Big developments in Coq are possible...
- ... but are still rather time consuming.
- Often simple things turn out not to be that simple (intuition).
- Dependent types are powerful and expressive...
- ... but reasoning about them is difficult.
- Working with equality different than identity is burdensome (although Setoid tactic makes it somehow easier).
- Some form of handling symmetries would be very helpful.

# Evaluation of Coq

- Big developments in Coq are possible...
- ... but are still rather time consuming.
- Often simple things turn out not to be that simple (intuition).
- Dependent types are powerful and expressive...
- ... but reasoning about them is difficult.
- Working with equality different than identity is burdensome (although `Setoid` tactic makes it somehow easier).
- Some form of handling symmetries would be very helpful.

# Evaluation of Coq

- Big developments in Coq are possible...
- ... but are still rather time consuming.
- Often simple things turn out not to be that simple (intuition).
- Dependent types are powerful and expressive...
- ... but reasoning about them is difficult.
- Working with equality different than identity is burdensome (although Setoid tactic makes it somehow easier).
- **Some form of handling symmetries would be very helpful.**

So now we can be certain that HORPO is correct.



Thank you for your attention.