Certified Higher-Order Recursive Path Ordering

... that is a short story of a never-ending formalization

Adam Koprowski

Technical University Eindhoven
Department of Mathematics and Computer Science

16 February 2006
OAS Group Meeting
Outline

1. Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2. Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. Zooming-in: equivalence on terms extending \( \alpha \)-convertibility
   - Introduction to problem
   - \( \alpha \)-convertibility
   - Equivalence on terms
Outline

1. Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
     - What is higher-order rewriting?
     - What is HORPO?

2. Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
   - Introduction to problem
   - $\alpha$-convertibility
   - Equivalence on terms
Outline

1. Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
     - What is HORPO?

2. Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
   - Introduction to problem
   - $\alpha$-convertibility
   - Equivalence on terms
Outline

1 Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2 Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3 Zooming-in: equivalence on terms extending α-convertibility
   - Introduction to problem
   - α-convertibility
   - Equivalence on terms
**Outline**

1. **Introduction**
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2. **Overview of the formalization**
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. **Zooming-in: equivalence on terms extending \( \alpha \)-convertibility**
   - Introduction to problem
   - \( \alpha \)-convertibility
   - Equivalence on terms
Outline

1. Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2. Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
   - Introduction to problem
   - $\alpha$-convertibility
   - Equivalence on terms
Outline

1. Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2. Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
   - Introduction to problem
   - $\alpha$-convertibility
   - Equivalence on terms
Outline

1 Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2 Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3 Zooming-in: equivalence on terms extending $\alpha$-convertibility
   - Introduction to problem
   - $\alpha$-convertibility
   - Equivalence on terms
Outline

1. Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2. Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. Zooming-in: equivalence on terms extending \(\alpha\)-convertibility
   - Introduction to problem
   - \(\alpha\)-convertibility
   - Equivalence on terms
Outline

1. Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2. Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
   - Introduction to problem
   - $\alpha$-convertibility
   - Equivalence on terms

Adam Koprowski
Certified Higher-Order Recursive Path Ordering
Outline

1. Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2. Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
   - Introduction to problem
   - $\alpha$-convertibility
   - Equivalence on terms
Outline

1. Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2. Overview of the formalization

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
Simply typed lambda calculus (\(\lambda \rightarrow\)) is a formalism to describe computable functions introduced by Church in the 1930s.

**Definition (Simple types)**

Given set of sorts \(S\) we define simple types as:

\[
T := S \mid T \rightarrow T
\]

**Definition (Preterms)**

We define preterms as:

\[
P t := x \mid f \mid @(P t, P t) \mid \lambda x : T . P t
\]

**Definition (Environments)**

We define environment as a set of variable declarations:

\[
\Gamma = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}
\]
Simply typed lambda calculus (\(\lambda \rightarrow\)) is a formalism to describe computable functions introduced by Church in the 1930s.

**Definition (Simple types)**

Given set of sorts \(S\) we define **simple types** as:

\[ T ::= S \mid T \rightarrow T \]

**Definition (Preterms)**

We define preterms as:

\[ Pt ::= x \mid f \mid \theta(Pt, Pt) \mid \lambda x : T . Pt \]

**Definition (Environments)**

We define environment as a set of variable declarations:

\[ \Gamma = \{ x_1 : \alpha_1, \ldots, x_n : \alpha_n \} \]
Simply typed lambda calculus

Simply typed lambda calculus ($\lambda \to$) is a formalism to describe computable functions introduced by Church in the 1930s.

### Definition (Simple types)

Given set of sorts $S$ we define simple types as:

$$T := S \mid T \to T$$

### Definition (Preterms)

We define **preterms** as:

$$\mathcal{P}t := x \mid f \mid \varnothing(\mathcal{P}t, \mathcal{P}t) \mid \lambda x : T . \mathcal{P}t$$
Simply typed lambda calculus

Simply typed lambda calculus ($\lambda \rightarrow$) is a formalism to describe computable functions introduced by Church in the 1930s.

Definition (Simple types)

Given set of sorts $S$ we define simple types as:

$$T ::= S \mid T \rightarrow T$$

Definition (Preterms)

We define preterms as:

$$Pt ::= x \mid f \mid @(Pt, Pt) \mid \lambda x : T. Pt$$

Definition (Environments)

We define environment as a set of variable declarations:

$$\Gamma = \{ x_1 : \alpha_1, \ldots, x_n : \alpha_n \}$$
Definition (Typing judgements)

We will write typing judgements of the form $\Gamma \vdash t : \alpha$ to denote that in environment $\Gamma$ preterm $t$ has type $\alpha$. They respect the following inference system rules:

\[
\begin{align*}
\Gamma \vdash x : \alpha & \quad & f : \alpha \in \Sigma & \quad & \Gamma \vdash f : \alpha \\
\Gamma \vdash t : \alpha \rightarrow \beta & & \Gamma \vdash u : \alpha \\
\Gamma \vdash \text{@}(t, u) : \beta & \quad & \Gamma \cup \{x : \alpha\} \vdash t : \beta \\
\Gamma \vdash \lambda x : \alpha. t : \alpha \rightarrow \beta &
\end{align*}
\]
\(\alpha\)-conversion and \(\beta\)-reduction

**Definition (\(\alpha\)-conversion)**

\(\alpha\)-conversion is defined as:

\[
\lambda x : \alpha. t = \lambda y : \alpha. t[x := y]
\]

if \(y\) does not appear freely in \(t\) and \(y\) is not bound in \(t\)

\(\alpha\)-conversions expresses the irrelevance of bound variable names.

**Definition (\(\beta\)-reduction)**

\(\beta\)-reduction is defined as:

\[
\@ (\lambda x : \alpha. t, u) \rightarrow_\beta t[x := u]
\]

\(\beta\)-reduction models computation in \(\lambda \rightarrow\).
\(\alpha\)-conversion and \(\beta\)-reduction

Definition (\(\alpha\)-conversion)

\(\alpha\)-conversion is defined as:

\[ \lambda x : \alpha. t = \lambda y : \alpha. t[x := y] \]

if \(y\) does not appear freely in \(t\) and \(y\) is not bound in \(t\)

\(\alpha\)-conversions express the irrelevance of bound variable names.

Definition (\(\beta\)-reduction)

\(\beta\)-reduction is defined as:

\[ \ominus(\lambda x : \alpha. t, u) \rightarrow_\beta t[x := u] \]

\(\beta\)-reduction models computation in \(\lambda \rightarrow\).
Outline

1. Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2. Overview of the formalization

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
Recursive path order

- Termination is an important concept in term rewriting.
- RPO is an ordering for proving termination.
- It goes back to Dershowitz 1982.

**Definition (RPO)**

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

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

1. $s_i \triangleright_{rpo} t$ for some $1 \leq i \leq n$.
2. $f \triangleright g$ and $s \triangleright_{rpo} t_i$ for all $1 \leq i \leq m$.
3. $f = g$ and $(s_1, \ldots, s_n) \triangleright_{rpo} \tau_f(t_1, \ldots, t_m)$.
Recursive path order

- Termination is an important concept in term rewriting.
- RPO is an ordering for proving termination.
- It goes back to Dershowitz 1982.

Definition (RPO)

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

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

1. $s_i \succ_{\text{rpo}} t$ for some $1 \leq i \leq n$.
2. $f \succ g$ and $s \succ_{\text{rpo}} t_i$ for all $1 \leq i \leq m$.
3. $f = g$ and $(s_1, \ldots, s_n) \succ_{\text{rpo}}^{\tau(f)} (t_1, \ldots, t_m)$
Recursive path order

- Termination is an important concept in term rewriting.
- RPO is an ordering for proving termination.
- It goes back to Dershowitz 1982.

**Definition (RPO)**

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

\[
s = f(s_1, \ldots, s_n) \succ_{rpo} g(t_1, \ldots, t_m) = t \iff \\
\text{1. } s_i \succ_{rpo} t \text{ for some } 1 \leq i \leq n. \\
\text{2. } f \succ g \text{ and } s \succ_{rpo} t_i \text{ for all } 1 \leq i \leq m. \\
\text{3. } f = g \text{ and } (s_1, \ldots, s_n) \succ_{rpo}^{\tau(f)} (t_1, \ldots, t_m).
\]
Recursive path order

- Termination is an important concept in term rewriting.
- RPO is an ordering for proving termination.
- It goes back to Dershowitz 1982.

Definition (RPO)

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

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

1. $s_i \succ_{rpo} t$ for some $1 \leq i \leq n$.
2. $f \succ g$ and $s \succ_{rpo} t_i$ for all $1 \leq i \leq m$.
3. $f = g$ and $(s_1, \ldots, s_n) \succ^{\tau(f)}_{rpo} (t_1, \ldots, t_m)$
Recursive path order

**Definition (RPO)**

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

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

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

**Theorem**

RPO is a reduction ordering meaning that given TRS $R$ and a well-founded precedence $\triangleright$ if for every rule $\ell \rightarrow r$ of $R$, $\ell \triangleright_{rpo} r$ then $R$ is terminating.
Outline

1 Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2 Overview of the formalization

3 Zooming-in: equivalence on terms extending $\alpha$-convertibility
Higher-order rewriting

There are three variants of higher-order rewriting:

- **HRS** Higher-order rewriting systems (Nipkow)
  - HλT terms
  - Rules restricted to patterns
  - Rewriting modulo βη

- **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 AFS.
There are three variants of higher-order rewriting:

**HRS** Higher-order rewriting systems (Nipkow)
- $\lambda \rightarrow \text{terms.}$
- 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 AFS.
Higher-order rewriting

There are three variants of higher-order rewriting:

**HRS**  Higher-order rewriting systems (Nipkow)
- $\lambda \to$ 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 AFS.
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 AFS.
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 AFS.
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 AFS.
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 AFS.
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 AFS.
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 AFS.
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 AFS.
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 AFS.
Examples of higher-order rewriting

Example (AFS for map)

map(nil, F) → nil
map(cons(x, l), F) → cons(@(F, x), 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)

\[
\begin{align*}
\text{map}(\text{nil}, F) & \rightarrow \text{nil} \\
\text{map}(\text{cons}(x, l), F) & \rightarrow \text{cons}(\mathbb{O}(F, x), \text{map}(l, F))
\end{align*}
\]

Example (AFS for summation)

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

\[
\begin{align*}
\Sigma(0, F) & \rightarrow \mathbb{O}(F, 0) \\
\Sigma(s(n), F) & \rightarrow +\left(\Sigma(n, F), \mathbb{O}(F, s(n))\right)
\end{align*}
\]
1. Introduction
   - Crash course in simply typed lambda calculus
   - What is RPO?
   - What is higher-order rewriting?
   - What is HORPO?

2. Overview of the formalization

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
Higher-order recursive path ordering

Definition (HORPO)

\[ \Gamma \vdash t : \delta \succ \Gamma \vdash u : \delta \text{ iff one of the following holds:} \]

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

where \( \succ \) is defined as:

\[ t = f(t_1, \ldots, t_k) \succ \{u_1, \ldots, u_n\} \text{ iff } \forall i \in \{1, \ldots, n\} \cdot t \succ u_i \lor (\exists j \cdot t_j \succeq u_i). \]
### Higher-order recursive path ordering

**Definition (RPO)**

\[
s = f(s_1, \ldots, s_n) \succ_{rpo} g(t_1, \ldots, t_m) = t ⇔
\]

1. \(s_i \succ_{rpo} t\) 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 \((s_1, \ldots, s_n) \succ_{rpo} \tau(f) (t_1, \ldots, t_m)\)

**Definition (HORPO)**

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

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

where \(\succ\) is defined as:

\[t = f(t_1, \ldots, t_k) \succ \{u_1, \ldots, u_n\}\] iff
\n\[\forall i \in \{1, \ldots, n\}. t > u_i \lor (\exists j. t_j \geq u_i).\]

Adam Koprowski  
Certified Higher-Order Recursive Path Ordering
Outline

1. Introduction

2. Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
Motivation & Goals

**Motivation:** Why making such formalization?

- Verification of the theory (especially for complicated, not very well-known proofs).
- **CoLoR:** Coq library on rewriting and termination, [http://color.loria.fr](http://color.loria.fr).
- Because it is fun.

**Goal:** formalization that is:

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

Motivation: Why making such formalization?

- Verification of the theory (especially for complicated, not very well-known proofs).
- Because it is fun.

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).
- Because it is fun.

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).
- **CoLoR**: Coq library on rewriting and termination, [http://color.loria.fr](http://color.loria.fr).
- **Because it is fun.**

**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).
- Because it is fun.

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).
- **CoLoR**: Coq library on rewriting and termination, [http://color.loria.fr](http://color.loria.fr).
- Because it is fun.

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).
- Because it is fun.

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).
- Because it is fun.

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).
- Because it is fun.

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).
- **CoLoR:** Coq library on rewriting and termination, http://color.loria.fr.
- Because it is fun.

**Goal:** formalization that is:

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

Motivation: Why making such formalization?

- Verification of the theory (especially for complicated, not very well-known proofs).
- Because it is fun.

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).
- Because it is fun.

**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).
- Because it is fun.

**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. Introduction

2. Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
Jean-Pierre Jouannaud and Albert Rubio proved that the higher-order recursive path ordering is a higher-order reduction ordering. This work is a formal verification of this proof in the theorem prover Coq.

The core of that property is the well-foundedness of the union of HORPO relation and the $\beta$-reduction of $\lambda\rightarrow$. Hence as a corollary we get termination of $\lambda\rightarrow$. 
Jean-Pierre Jouannaud and Albert Rubio proved that the higher-order recursive path ordering is a higher-order reduction ordering. This work is a formal verification of this proof in the theorem prover Coq.

The core of that property is the well-foundedness of the union of HORPO relation and the $\beta$-reduction of $\lambda \rightarrow$. Hence as a corollary we get termination of $\lambda \rightarrow$. 
Jean-Pierre Jouannaud and Albert Rubio proved that the higher-order recursive path ordering is a higher-order reduction ordering. This works is a formal verification of this proof in the theorem prover Coq.

The core of that property is the well-foundedness of the union of HORPO relation and the $\beta$-reduction of $\lambda \rightarrow$. Hence as a corollary we get termination of $\lambda \rightarrow$. 
Jean-Pierre Jouannaud and Albert Rubio proved that the higher-order recursive path ordering is a higher-order reduction ordering. This work is a formal verification of this proof in the theorem prover Coq.

The core of that property is the well-foundedness of the union of HORPO relation and the $\beta$-reduction of $\lambda \rightarrow$. Hence as a corollary we get termination of $\lambda \rightarrow$. 
Jean-Pierre Jouannaud and Albert Rubio proved that the higher-order recursive path ordering is a higher-order reduction ordering. This work is a formal verification of this proof in the theorem prover Coq.

The core of that property is the well-foundedness of the union of HORPO relation and the $\beta$-reduction of $\lambda \rightarrow$. Hence as a corollary we get termination of $\lambda \rightarrow$. 
Jean-Pierre Jouannaud and Albert Rubio proved that the higher-order recursive path ordering is a higher-order reduction ordering. This work is a formal verification of this proof in the theorem prover Coq.

The core of that property is the well-foundedness of the union of HORPO relation and the $\beta$-reduction of $\lambda \rightarrow$. Hence as a corollary we get termination of $\lambda \rightarrow$. 
Jean-Pierre Jouannaud and Albert Rubio proved that the higher-order recursive path ordering is a higher-order reduction ordering. This work is a formal verification of this proof in the theorem prover Coq.

The core of that property is the well-foundedness of the union of HORPO relation and the $\beta$-reduction of $\lambda \rightarrow$. Hence as a corollary we get termination of $\lambda \rightarrow$. 
Jean-Pierre Jouannaud and Albert Rubio proved that the higher-order recursive path ordering is a higher-order reduction ordering. This work is a formal verification of this proof in the theorem prover Coq.

The core of that property is the well-foundedness of the union of HORPO relation and the $\beta$-reduction of $\lambda \rightarrow$. Hence as a corollary we get termination of $\lambda \rightarrow$. 
Jean-Pierre Jouannaud and Albert Rubio proved that the higher-order recursive path ordering is a higher-order reduction ordering. This work is a formal verification of this proof in the theorem prover Coq.

The core of that property is the well-foundedness of the union of HORPO relation and the $\beta$-reduction of $\lambda \rightarrow$. Hence as a corollary we get termination of $\lambda \rightarrow$. 
Jean-Pierre Jouannaud and Albert Rubio proved that the higher-order recursive path ordering is a higher-order reduction ordering. This work is a formal verification of this proof in the theorem prover Coq.

The core of that property is the well-foundedness of the union of HORPO relation and the $\beta$-reduction of $\lambda \rightarrow$. Hence as a corollary we get termination of $\lambda \rightarrow$.

The higher-order recursive path ordering. 

Higher-order recursive path orderings ‘à la carte’
Development contents

- **Auxiliaries.**
  - Multisets & multiset order.
    - Finite multisets as ADT (primitive operations + their specification).
    - Concrete implementation (using lists).
    - A number of abstract properties.
    - Definition of multiset ordering.
    - Proof that multiset ordering preserves well-foundedness.
  - $\lambda\rightarrow$ terms.
    - Decidability of typing.
    - Definition of typed substitution (far from easy).
    - Equivalence relation extending $\alpha$-convertibility to free variables.
    - Termination of $\beta$-reduction.
    - Encoding of algebraic terms via $\lambda\rightarrow$ terms.

- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- **Auxiliaries.**
- **Multisets & multiset order.**
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.
- **$\lambda$ terms.**
  - Decidability of typing.
  - Definition of typed substitution (far from easy).
  - Equivalence relation extending $\alpha$-convertibility to free variables.
  - Termination of $\beta$-reduction.
- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- **Auxiliaries.**
- **Multisets & multiset order.**
  - **Finite multisets as ADT** (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.
- \( \lambda \rightarrow \) terms.
  - Decidability of typing.
  - Definition of typed substitution (far from easy).
  - Equivalence relation extending \( \alpha \)-convertibility to free variables.
  - Termination of \( \beta \)-reduction.
  - Encoding of algebraic terms via \( \lambda \rightarrow \) terms.
- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union \( \beta \)-reduction.
  - Well-foundedness of HORPO union \( \beta \)-reduction.
  - HORPO is a higher-order reduction ordering.
## Development contents

- **Auxiliaries.**
- **Multisets & multiset order.**
  - Finite multisets as ADT (primitive operations + their specification).
  - **Concrete implementation** (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.
- **λ→ terms.**
  - Decidability of typing.
  - Definition of typed substitution (far from easy).
  - Equivalence relation extending α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via λ→ terms.
- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union β-reduction.
  - Well-foundedness of HORPO union β-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- Auxiliaries.
- Multisets & multiset order.
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.
- $\lambda\to$ terms.
  - Decidability of typing.
  - Definition of typed substitution (far from easy).
  - Equivalence relation extending $\alpha$-convertibility to free variables.
  - Termination of $\beta$-reduction.
  - Encoding of algebraic terms via $\lambda\to$ terms.
- HORPO.
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- **Auxiliaries.**
- **Multisets & multiset order.**
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.
- **λ→ terms.**
  - Decidability of typing.
  - Definition of typed substitution (far from easy).
  - Equivalence relation extending \(\alpha\)-convertibility to free variables.
  - Termination of \(\beta\)-reduction.
  - Encoding of algebraic terms via \(\lambda\rightarrow\) terms.
- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union \(\beta\)-reduction.
  - Well-foundedness of HORPO union \(\beta\)-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- **Auxiliaries.**
- **Multisets & multiset order.**
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.
- **$\lambda \to$ terms.**
  - Decidability of typing.
  - Definition of typed substitution (far from easy).
  - Equivalence relation extending $\alpha$-convertibility to free variables.
  - Termination of $\beta$-reduction.
  - Encoding of algebraic terms via $\lambda \to$ terms.
- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- **Auxiliaries.**
- **Multisets & multiset order.**
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.

- **λ→ terms.**
  - Decidability of typing.
  - Definition of typed substitution (far from easy)
  - Equivalence relation extending α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via λ→ terms.

- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union β-reduction.
  - Well-foundedness of HORPO union β-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- **Auxiliaries.**
  - Multisets & multiset order.
    - Finite multisets as ADT (primitive operations + their specification).
    - Concrete implementation (using lists).
    - A number of abstract properties.
    - Definition of multiset ordering.
    - Proof that multiset ordering preserves well-foundedness.

- **$\lambda \rightarrow$ terms.**
  - **Decidability of typing.**
    - Definition of typed substitution (far from easy)
    - Equivalence relation extending $\alpha$-convertibility to free variables.
    - Termination of $\beta$-reduction.
    - Encoding of algebraic terms via $\lambda \rightarrow$ terms.

- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- **Auxiliaries.**
- **Multisets & multiset order.**
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.
- **$\lambda \to$ terms.**
  - Decidability of typing.
  - **Definition of typed substitution (far from easy)**
  - Equivalence relation extending $\alpha$-convertibility to free variables.
  - Termination of $\beta$-reduction.
  - Encoding of algebraic terms via $\lambda \to$ terms.
- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- **Auxiliaries.**
- **Multisets & multiset order.**
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.
- **$\lambda \rightarrow$ terms.**
  - Decidability of typing.
  - Definition of typed substitution (far from easy)
  - **Equivalence relation extending $\alpha$-convertibility to free variables.**
    - Termination of $\beta$-reduction.
    - Encoding of algebraic terms via $\lambda \rightarrow$ terms.
- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- **Auxiliaries.**

- **Multisets & multiset order.**
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.

- **$\lambda \rightarrow$ terms.**
  - Decidability of typing.
  - Definition of typed substitution (far from easy)
  - Equivalence relation extending $\alpha$-convertibility to free variables.
  - **Termination of $\beta$-reduction.**
  - Encoding of algebraic terms via $\lambda \rightarrow$ terms.

- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- Auxiliaries.
- Multisets & multiset order.
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.
- $\lambda \rightarrow$ terms.
  - Decidability of typing.
  - Definition of typed substitution (far from easy)
  - Equivalence relation extending $\alpha$-convertibility to free variables.
  - Termination of $\beta$-reduction.
  - Encoding of algebraic terms via $\lambda \rightarrow$ terms.
- HORPO.
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.

Adam Koprowski
Certified Higher-Order Recursive Path Ordering
Development contents

- **Auxiliaries.**
- **Multisets & multiset order.**
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.
- **$\lambda \rightarrow$ terms.**
  - Decidability of typing.
  - Definition of typed substitution (far from easy)
  - Equivalence relation extending $\alpha$-convertibility to free variables.
  - Termination of $\beta$-reduction.
  - Encoding of algebraic terms via $\lambda \rightarrow$ terms.
- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- **Auxiliaries.**

- **Multisets & multiset order.**
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.

- **\( \lambda \rightarrow \) terms.**
  - Decidability of typing.
  - Definition of typed substitution (far from easy)
  - Equivalence relation extending \( \alpha \)-convertibility to free variables.
  - Termination of \( \beta \)-reduction.
  - Encoding of algebraic terms via \( \lambda \rightarrow \) terms.

- **HORPO.**
  - **Definition of HORPO.**
  - Proofs of computability properties for HORPO union \( \beta \)-reduction.
  - Well-foundedness of HORPO union \( \beta \)-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- **Auxiliaries.**
- Multisets & multiset order.
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.
- $\lambda \rightarrow$ terms.
  - Decidability of typing.
  - Definition of typed substitution (far from easy)
  - Equivalence relation extending $\alpha$-convertibility to free variables.
  - Termination of $\beta$-reduction.
  - Encoding of algebraic terms via $\lambda \rightarrow$ terms.
- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
    - Well-foundedness of HORPO union $\beta$-reduction.
    - HORPO is a higher-order reduction ordering.
Development contents

- Auxiliaries.
- Multisets & multiset order.
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.
- $\lambda \rightarrow$ terms.
  - Decidability of typing.
  - Definition of typed substitution (far from easy)
  - Equivalence relation extending $\alpha$-convertibility to free variables.
  - Termination of $\beta$-reduction.
  - Encoding of algebraic terms via $\lambda \rightarrow$ terms.
- HORPO.
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- Auxiliaries.

- Multisets & multiset order.
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.

- $\lambda \rightarrow$ terms.
  - Decidability of typing.
  - Definition of typed substitution (far from easy)
  - Equivalence relation extending $\alpha$-convertibility to free variables.
  - Termination of $\beta$-reduction.
  - Encoding of algebraic terms via $\lambda \rightarrow$ terms.

- HORPO.
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.
Development contents

- **Auxiliaries.**
- **Multisets & multiset order.**
  - Finite multisets as ADT (primitive operations + their specification).
  - Concrete implementation (using lists).
  - A number of abstract properties.
  - Definition of multiset ordering.
  - Proof that multiset ordering preserves well-foundedness.

- **$\lambda\rightarrow$ terms.**
  - Decidability of typing.
  - Definition of typed substitution (far from easy)
  - Equivalence relation extending $\alpha$-convertibility to free variables.
  - Termination of $\beta$-reduction.
  - Encoding of algebraic terms via $\lambda\rightarrow$ terms.

- **HORPO.**
  - Definition of HORPO.
  - Proofs of computability properties for HORPO union $\beta$-reduction.
  - Well-foundedness of HORPO union $\beta$-reduction.
  - HORPO is a higher-order reduction ordering.
Outline

1. Introduction

2. Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
Relative sizes of different parts of the development

Image obtained using program SequoiaView developed at TU/e.
The development consists of:

- **29** files.
- >1000 lemmas
- >300 definitions (21 fixpoint def., 24 inductive def., 33 def. by proof)
- >22,000 script lines
- total size: >600 KB.
The development consists of:

- 29 files.
- >1000 lemmas
- >300 definitions (21 fixpoint def., 24 inductive def., 33 def. by proof)
- >22,000 script lines
- total size: >600 KB.
The development consists of:

- 29 files.
- >1000 lemmas
- >300 definitions (21 fixpoint def., 24 inductive def., 33 def. by proof)
- >22,000 script lines
- total size: >600 KB.
The development consists of:

- 29 files.
- >1000 lemmas
- >300 definitions (21 fixpoint def., 24 inductive def., 33 def. by proof)
- >22,000 script lines
- total size: >600 KB.
The development consists of:

- 29 files.
- >1000 lemmas
- >300 definitions (21 fixpoint def., 24 inductive def., 33 def. by proof)
- >22,000 script lines
- total size: >600 KB.
Outline

1. Introduction

2. Overview of the formalization
   - Why: motivation & goals
   - What: content of the formalization
   - How big: size of the development
   - When: history & timeline

3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
Timeline of the project

Two stages of the project:

- **Jan 2004 - Jul 2004**
  Master Thesis at the Vrije Universiteit supervised by Femke van Raamsdonk
  Proof completed but computability properties as axioms.

- **Nov 2004 - Feb 2006**
  Development continued at the Technical University Eindhoven.
  Completed, axiom free proof.
Timeline of the project

Two stages of the project:

- **Jan 2004 - Jul 2004**
  Master Thesis at the Vrije Universiteit supervised by Femke van Raamsdonk
  Proof completed but computability properties as axioms.

- **Nov 2004 - Feb 2006**
  Development continued at the Technical University Eindhoven.
  Completed, axiom free proof.
Introduction
Overview of the formalization
Zooming-in: equivalence on terms extending $\alpha$-convertibility
Summary

Why: motivation & goals
What: content of the formalization
How big: size of the development
When: history & timeline

Timeline of the second stage of the project

![Timeline Graph](image-url)
Outline

1. Introduction
2. Overview of the formalization
3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
   - Introduction to problem
   - $\alpha$-convertibility
   - Equivalence on terms
Problem

We want to consider certain terms as equal (without changing calculus in any way). For instance:

- $\lambda x : \alpha. x \equiv \lambda y : \alpha. y$
- $x : \alpha \vdash x : \alpha = x : \alpha, y : \beta \vdash x : \alpha$
- $x : \alpha \vdash x : \alpha = y : \alpha \vdash y : \alpha$

Solution: define appropriate equivalence relation on terms $\sim$ that enjoys nice properties and covers the above equalities.
Problem

We want to consider certain terms as equal (without changing calculus in any way). For instance:

- $\lambda x: \alpha. x = \lambda y: \alpha. y$
- $x: \alpha \vdash x: \alpha = x: \alpha, y: \beta \vdash x: \alpha$
- $x: \alpha \vdash x: \alpha = y: \alpha \vdash y: \alpha$

Solution: define appropriate equivalence relation on terms $\sim$ that enjoys nice properties and covers the above equalities.
Problem

We want to consider certain terms as equal (without changing calculus in any way). For instance:

- $\lambda x : \alpha. x = \lambda y : \alpha. y$
- $x : \alpha \vdash x : \alpha = x : \alpha, y : \beta \vdash x : \alpha$
- $x : \alpha \vdash x : \alpha = y : \alpha \vdash y : \alpha$

Solution: define appropriate equivalence relation on terms $\sim$ that enjoys nice properties and covers the above equalities.
We want to consider certain terms as equal (without changing calculus in any way). For instance:

- $\lambda x : \alpha. x =_\alpha \lambda y : \alpha. y$
- $x : \alpha \vdash x : \alpha = x : \alpha, y : \beta \vdash x : \alpha$
- $x : \alpha \vdash x : \alpha = y : \alpha \vdash y : \alpha$

Solution: define appropriate equivalence relation on terms $\sim$ that enjoys nice properties and covers the above equalities.
We want to consider certain terms as equal (without changing calculus in any way). For instance:

- $\lambda x : \alpha. x =_\alpha \lambda y : \alpha. y$
- $x : \alpha \vdash x : \alpha = x : \alpha, y : \beta \vdash x : \alpha$
- $x : \alpha \vdash x : \alpha = y : \alpha \vdash y : \alpha$

**Solution:** define appropriate equivalence relation on terms $\sim$ that enjoys nice properties and covers the above equalities.
Outline

1. Introduction
2. Overview of the formalization
3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
   - Introduction to problem
   - $\alpha$-convertibility
   - Equivalence on terms

Adam Koprowski
Certified Higher-Order Recursive Path Ordering
Tackling $\alpha$-convertibility

Standard solution: de Bruijn indices:
- natural numbers instead of names for variables,
- number of the variable indicates where it is bound,
- lambda binders come with no name,
- variable number indicates how many lambdas in the term tree we have to skip on the way to the root to find the binder for variable,
- in this way we get unique representation for $\alpha$-convertible terms.

Example
- Identity: $\lambda x: \alpha. x = \lambda \alpha.0 = \lambda y: \alpha. y$
- First projection: $\lambda x: \alpha. \lambda y: \alpha. x = \lambda \alpha. \lambda \alpha.1$
- $x: \beta \vdash \lambda y: \alpha \to \beta. \alpha = \beta \vdash \lambda \alpha \to \beta. \alpha(0,1)$
Tackling $\alpha$-convertibility

Standard solution: de Bruijn indices:
- natural numbers instead of names for variables,
- number of the variable indicates where it is bound,
- lambda binders come with no name,
- variable number indicates how many lambdas in the term tree we have to skip on the way to the root to find the binder for variable,
- in this way we get unique representation for $\alpha$-convertible terms.

Example
- Identity: $\lambda x: \alpha. x = \lambda x: \alpha. 0 = \lambda y: \alpha. y$
- First projection: $\lambda x: \alpha. \lambda y: \alpha. x = \lambda x: \alpha. \lambda y: \alpha. 1$
- $\vdash x: \beta \vdash \lambda y: \alpha \to \beta. @(y, x) = \beta \vdash \lambda x: \alpha \to \beta. @(0, 1)$
Tackling $\alpha$-convertibility

Standard solution: de Bruijn indices:
- natural numbers instead of names for variables,
- number of the variable indicates where it is bound,
- lambda binders come with no name,
- variable number indicates how many lambdas in the term tree we have to skip on the way to the root to find the binder for variable,
- in this way we get unique representation for $\alpha$-convertible terms.

Example
- Identity: $\lambda x:\alpha. x = \lambda \alpha. 0 = \lambda y:\alpha. y$
- First projection: $\lambda x:\alpha. \lambda y:\alpha. x = \lambda \alpha. \lambda \alpha. 1$
- $x:\beta \vdash \lambda y:\alpha \rightarrow \beta. \circ(y, x) = \beta \vdash \lambda \alpha \rightarrow \beta. \circ(0, 1)$
Tackling $\alpha$-convertibility

Standard solution: de Bruijn indices:
- natural numbers instead of names for variables,
- number of the variable indicates where it is bound,
- lambda binders come with no name,
- variable number indicates how many lambdas in the term tree we have to skip on the way to the root to find the binder for variable,
- in this way we get unique representation for $\alpha$-convertible terms.

Example
- Identity: $\lambda x : \alpha . x = \lambda x . 0 = \lambda y : \alpha . y$
- First projection: $\lambda x : \alpha . \lambda y : \alpha . x = \lambda x . \lambda y . 1$
- $x : \beta \vdash \lambda y : \alpha \to \beta . \odot (y, x) = \beta \vdash \lambda \alpha : \beta \to \beta . \odot (0, 1)$
Tackling \( \alpha \)-convertibility

Standard solution: de Bruijn indices:

- natural numbers instead of names for variables,
- number of the variable indicates where it is bound,
- lambda binders come with no name,
- variable number indicates how many lambdas in the term tree we have to skip on the way to the root to find the binder for variable,

in this way we get unique representation for \( \alpha \)-convertible terms.

Example

- Identity: \( \lambda x: \alpha. x = \lambda \alpha.0 = \lambda y: \alpha. y \)
- First projection: \( \lambda x: \alpha. \lambda y: \alpha. x = \lambda \alpha. \lambda \alpha.1 \)
- \( \chi: \beta \vdash \lambda y: \alpha \to \beta. @ (y, x) = \beta \vdash \lambda \alpha \to \beta. @ (0, 1) \)
Tackling $\alpha$-convertibility

Standard solution: de Bruijn indices:

- natural numbers instead of names for variables,
- number of the variable indicates where it is bound,
- lambda binders come with no name,
- variable number indicates how many lambdas in the term tree we have to skip on the way to the root to find the binder for variable,
- in this way we get unique representation for $\alpha$-convertible terms.

Example

- Identity: $\lambda x: \alpha. x = \lambda \alpha. 0 = \lambda y: \alpha. y$
- First projection: $\lambda x: \alpha. \lambda y: \alpha. x = \lambda \alpha. \lambda \alpha. 1$
- $x: \beta \vdash \lambda y: \alpha \to \beta. @ (y, x) = \beta \vdash \lambda \alpha \to \beta. @ (0, 1)$
Standard solution: de Bruijn indices:

- natural numbers instead of names for variables,
- number of the variable indicates where it is bound,
- lambda binders come with no name,
- variable number indicates how many lambdas in the term tree we have to skip on the way to the root to find the binder for variable,
- in this way we get unique representation for $\alpha$-convertible terms.

Example

- **Identity**: $\lambda x : \alpha . x = \lambda \alpha . 0 = \lambda y : \alpha . y$
- **First projection**: $\lambda x : \alpha . \lambda y : \alpha . x = \lambda \alpha . \lambda \alpha . 1$
- $x : \beta \vdash \lambda y : \alpha \rightarrow \beta . @ (y, x) = \beta \vdash \lambda \alpha \rightarrow \beta . @ (0, 1)$
Tackling $\alpha$-convertibility

Standard solution: de Bruijn indices:
- natural numbers instead of names for variables,
- number of the variable indicates where it is bound,
- lambda binders come with no name,
- variable number indicates how many lambdas in the term tree we have to skip on the way to the root to find the binder for variable,
- in this way we get unique representation for $\alpha$-convertible terms.

Example
- Identity: $\lambda x : \alpha. x = \lambda \alpha. 0 = \lambda y : \alpha. y$
- First projection: $\lambda x : \alpha. \lambda y : \alpha. x = \lambda \alpha. \lambda \alpha. 1$
- $x : \beta \vdash \lambda y : \alpha \rightarrow \beta. @(y, x) = \beta \vdash \lambda \alpha \rightarrow \beta. @(0, 1)$
Tackling $\alpha$-convertibility

Standard solution: de Bruijn indices:
- natural numbers instead of names for variables,
- number of the variable indicates where it is bound,
- lambda binders come with no name,
- variable number indicates how many lambdas in the term tree we have to skip on the way to the root to find the binder for variable,
- in this way we get unique representation for $\alpha$-convertible terms.

Example

- Identity: $\lambda x : \alpha.x = \lambda \alpha.0 = \lambda y : \alpha.y$
- First projection: $\lambda x : \alpha.\lambda y : \alpha.x = \lambda \alpha.\lambda \alpha.1$
- $x : \beta \vdash \lambda y : \alpha \rightarrow \beta.\otimes(y, x) = \beta \vdash \lambda \alpha \rightarrow \beta.\otimes(0, 1)$
\(\alpha\)-convertibility in Coq

- Environment simply becomes a list of types:
  
  \[
  \text{Env} : \text{list SimpleType}
  \]

- However we need dummy variables so:
  
  \[
  \text{Env} : \text{list (option SimpleType)}
  \]

- But this leads to problems...

- So we need to define custom equality for environments:

  \[
  \text{Definition envSubset E1 E2 := forall x A, E1 \models x := A \rightarrow E2 \models x := A.}
  \]

  \[
  \text{Definition env_eq E1 E2 := envSubset E1 E2 \land\ envSubset E2 E1.}
  \]
Environment simply becomes a list of types:

\[ \text{Env: list SimpleType} \]

However we need dummy variables so:

\[ \text{Env: list (option SimpleType)} \]

But this leads to problems...

So we need to define custom equality for environments:

\[
\text{Definition envSubset E1 E2 := forall x A, E1 \models x := A -> E2 \models x := A.}
\]

\[
\text{Definition env_eq E1 E2 := envSubset E1 E2 /\ envSubset E2 E1.}
\]
Introduction to problem

\( \alpha \)-convertibility

Equivalence on terms

\( \alpha \)-convertibility in Coq

- Environment simply becomes a list of types:
  \[
  \text{Env} : \text{list SimpleType}
  \]

- However we need dummy variables so:
  \[
  \text{Env} : \text{list (option SimpleType)}
  \]

- But this leads to problems...

So we need to define custom equality for environments:

\[
\begin{align*}
\text{Definition envSubset E1 E2} & := \text{forall } x, A, \ E1 \models x := A -> E2 \models x := A. \\
\text{Definition env_eq E1 E2} & := \text{envSubset E1 E2} \land \text{envSubset E2 E1}.
\end{align*}
\]
\(\alpha\)-convertibility in Coq

- Environment simply becomes a list of types:
  \[
  \text{Env} : \text{list} \ \text{SimpleType}
  \]
- However we need dummy variables so:
  \[
  \text{Env} : \text{list} \ (\text{option} \ \text{SimpleType})
  \]
- But this leads to problems...
Environment simply becomes a list of types:

\[ \text{Env: list SimpleType} \]

However we need dummy variables so:

\[ \text{Env: list (option SimpleType)} \]

But this leads to problems...

... as we loose unique representation for environment. For instance empty environment can be represented as \text{nil} or as \text{None::nil} etc.

So we need to define custom equality for environments:

\[ \text{Definition envSubset E1 E2 := forall x A, E1 |= x := A -> E2 |= x := A.} \]
\[ \text{Definition env_eq E1 E2 := envSubset E1 E2} \setminus \text{ envSubset E2 E1.} \]
\(\alpha\)-convertibility in Coq

- Environment simply becomes a list of types:
  \[
  \text{Env} : \text{list} \ \text{SimpleType}
  \]
- However we need dummy variables so:
  \[
  \text{Env} : \text{list} \ (\text{option} \ \text{SimpleType})
  \]
- But this leads to problems...
- ... as we loose unique representation for environment. For instance empty environment can be represented as \text{nil} or as \text{None::nil} etc.
- So we need to define custom equality for environments:

\[
\text{Definition envSubset E1 E2 := forall x A, E1 |- x := A -> E2 |- x := A.}
\]
\[
\text{Definition env_eq E1 E2 := envSubset E1 E2 /\ envSubset E2 E1.}
\]
Outline

1. Introduction
2. Overview of the formalization
3. Zooming-in: equivalence on terms extending $\alpha$-convertibility
   - Introduction to problem
   - $\alpha$-convertibility
   - Equivalence on terms
Introduction to problem
\(\alpha\)-convertibility
Equivalence on terms

1st naive attempt

Definition (Environment compatibility)
We say that environments \(\Gamma\) and \(\Delta\) are compatible (\(\Gamma \leftrightarrow \Delta\)) iff:

\[
\begin{align*}
\{ & x : \alpha \in \Gamma \\
& x : \beta \in \Delta \} \\
\}
\implies \alpha = \beta
\end{align*}
\]

Definition (Equivalence)
Let \(\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta\) iff: \(t = u \land \Gamma \leftrightarrow \Delta\).

- Does not address third equality: \(x : \alpha \vdash x : \alpha = y : \alpha \vdash y : \alpha\),
- Even worse: no transitivity.

\[
\begin{align*}
& x : \beta \vdash c : \alpha \sim \emptyset \vdash c : \alpha \sim x : \gamma \vdash c : \alpha \\
& x : \beta \vdash c : \alpha \sim x : \gamma \vdash c : \alpha
\end{align*}
\]
1st naive attempt

Definition (Environment compatibility)

We say that environments \( \Gamma \) and \( \Delta \) are compatible (\( \Gamma \leftrightarrow \Delta \)) iff:

\[
\begin{align*}
    x : \alpha &\in \Gamma \\
    x : \beta &\in \Delta
\end{align*}
\]

\( \implies \alpha = \beta \)

Definition (Equivalence)

Let \( \Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta \) iff: \( t = u \land \Gamma \leftrightarrow \Delta \).

- Does not address third equality: \( x : \alpha \vdash x : \alpha = y : \alpha \vdash y : \alpha \),
- Even worse: no transitivity.

\[
\begin{align*}
    x : \beta \vdash c : \alpha &\sim \emptyset \vdash c : \alpha \\
    x : \gamma \vdash c : \alpha &\sim x : \gamma \vdash c : \alpha
\end{align*}
\]
1st naive attempt

Definition (Environment compatibility)
We say that environments $\Gamma$ and $\Delta$ are compatible ($\Gamma \leftrightarrow \Delta$) iff:

$$\begin{align*}
x : \alpha & \in \Gamma \\
x : \beta & \in \Delta
\end{align*} \implies \alpha = \beta$$

Definition (Equivalence)
Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff: $t = u \land \Gamma \leftrightarrow \Delta$.

- Does not address third equality: $x : \alpha \vdash x : \alpha = y : \alpha \vdash y : \alpha$.
- Even worse: no transitivity.

$$\begin{align*}
x : \beta & \vdash c : \alpha \sim \emptyset \vdash c : \alpha \sim x : \gamma & \vdash c : \alpha \\
x : \beta & \vdash c : \alpha \sim x : \gamma & \vdash c : \alpha
\end{align*}$$
1st naive attempt

**Definition (Environment compatibility)**
We say that environments $\Gamma$ and $\Delta$ are compatible ($\Gamma \leftrightarrow \Delta$) iff:

$$\{ x : \alpha \in \Gamma, x : \beta \in \Delta \} \implies \alpha = \beta$$

**Definition (Equivalence)**
Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff: $t = u \land \Gamma \leftrightarrow \Delta$.

- Does not address third equality: $x : \alpha \vdash x : \alpha = y : \alpha \vdash y : \alpha$,
- Even worse: no transitivity.

$$x : \beta \vdash c : \alpha \sim \emptyset \vdash c : \alpha \sim x : \gamma \vdash c : \alpha$$

$$x : \beta \vdash c : \alpha \sim x : \gamma \vdash c : \alpha$$
Definition (Equivalence)

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff there exists a partial injective function $\Phi : \text{Var} \rightarrow \text{Var}$ such that:

$$\forall x : \alpha \in \Gamma, y : \beta \in \Delta . \ x \Phi y \implies \alpha = \beta$$

and $t \approx \Phi u$ where $\approx \Phi$:

- $x \approx \Phi y$ if $x \Phi y$
- $f \approx \Phi f$
- $@(t_l, t_r) \approx \Phi @(u_l, u_r)$ if $t_l \approx \Phi u_l \land t_r \approx \Phi u_r$
- $\lambda \alpha . t \approx \Phi \lambda \alpha . u$ if $t \approx \Phi^{\uparrow 1} u$

Problem: the following property does not hold:

$t \sim \Phi u \land \Phi \subset \Phi' = \implies t \sim \Phi' u$
2nd (somehow) less naive attempt

Definition (Equivalence)
Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff there exists a partial injective function $\Phi : \text{Var} \rightarrow \text{Var}$ such that:

$$\forall x : \alpha \in \Gamma, y : \beta \in \Delta . x \Phi y \implies \alpha = \beta$$

and $t \approx_{\Phi} u$ where $\approx_{\Phi}$:

$$x \approx_{\Phi} x$$
$$f \approx_{\Phi} f$$
$$@(t_l, t_r) \approx_{\Phi} @(u_l, u_r)$$
$$\lambda \alpha . t \approx_{\Phi} \lambda \alpha . u$$

Problem: the following property does not hold:

$$t \sim_{\Phi} u \wedge \Phi \subset \Phi' \implies t \sim_{\Phi'} u$$

Consider:

$$t = x : \alpha \vdash c : \alpha, u = x : \beta \vdash c : \alpha$$

$t \sim_{\emptyset} u$ but $t \sim_{\{(x,x)\}} u$
2nd (somehow) less naive attempt

Definition (Equivalence)

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff there exists a partial injective function $\Phi : \text{Var} \rightarrow \text{Var}$ such that:

$$\forall x : \alpha \in \Gamma, y : \beta \in \Delta. x \Phi y \implies \alpha = \beta$$

and $t \approx_\Phi u$ where $\approx_\Phi$:

- $x \approx_\Phi y$ if $x \Phi y$
- $f \approx_\Phi f$
- $@(t_l, t_r) \approx_\Phi @(u_l, u_r)$ if $t_l \approx_\Phi u_l \land t_r \approx_\Phi u_r$
- $\lambda \alpha. t \approx_\Phi \lambda \alpha. u$ if $t \approx_\Phi 1 u$

Problem: the following property does not hold:

$$t \sim_\Phi u \land \Phi \subset \Phi' \implies t \sim_{\Phi'} u$$

Consider:

$$t = x : \alpha \vdash c : \alpha, u = x : \beta \vdash c : \alpha$$

$t \sim_\emptyset u$ but $t \not\sim \{(x, x)\} u$
Definition (Active environment)

For $\Gamma \vdash t : \alpha$ we define the active environment of $t$ as $\Omega(t)$:

$$
\begin{align*}
\Omega(x : \alpha) &= \{x : \alpha\} \\
\Omega(f) &= \emptyset \\
\Omega(@ (t_l, t_r)) &= \Omega(t_l) \cup \Omega(t_r) \\
\Omega(\lambda \alpha.t) &= \Omega(t) \uparrow^1
\end{align*}
$$

Definition (Equivalence)

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff there exists a partial injective function $\Phi : \text{Var} \rightarrow \text{Var}$ such that:

$$
\forall x : \alpha \in \Omega(\Gamma), \ y : \beta \in \Omega(\Delta). \ x \Phi y \implies \alpha = \beta
$$

Adam Koprowski  Certified Higher-Order Recursive Path Ordering
Introduction

Overview of the formalization

Zooming-in: equivalence on terms extending $\alpha$-convertibility

Summary

Solution

Definition (Active environment)

For $\Gamma \vdash t : \alpha$ we define active environment of $t$ as $\Omega(t)$:

- $\Omega(x : \alpha) = \{x : \alpha\}$
- $\Omega(f) = \emptyset$
- $\Omega(@_t(t_l, t_r)) = \Omega(t_l) \cup \Omega(t_r)$
- $\Omega(\lambda\alpha.t) = \Omega(t) \uparrow^1$

Definition (Equivalence)

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff there exists a partial injective function $\Phi : \text{Var} \to \text{Var}$ such that:

$$\forall x : \alpha \in \Omega(\Gamma), y : \beta \in \Omega(\Delta). x \Phi y \implies \alpha = \beta$$

and $t \approx_\Phi u$ where $\approx_\Phi$ defined as before.

This works fine and enjoys a number of nice properties:
**Solution**

**Definition (Equivalence)**

Let \[ \Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta \] iff there exists a partial injective function \( \Phi : \text{Var} \rightarrow \text{Var} \) such that:

\[
\forall x : \alpha \in \Omega(\Gamma), y : \beta \in \Omega(\Delta). x \Phi y \implies \alpha = \beta
\]

and \( t \approx \Phi u \) where \( \approx \Phi \) defined as before.

This works fine and enjoys a number of nice properties:

- \( t \sim \Phi t' \land \gamma \sim \Phi \gamma' \implies t\gamma \sim \Phi t'\gamma' \)
- \( t \rightarrow_\beta u \land t \sim \Phi t' \land u \sim \Phi u' \implies t' \rightarrow_\beta u' \)
- \( t \succ u \land t \sim \Phi t' \land u \sim \Phi u' \implies t' \succ u' \)

However it is more complicated than the previous variant as it is really a property of typed terms and not of preterms and environments.
Solution

**Definition (Equivalence)**

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff there exists a partial injective function $\Phi : \text{Var} \rightarrow \text{Var}$ such that:

$$\forall x : \alpha \in \Omega(\Gamma), y : \beta \in \Omega(\Delta). x \Phi y \implies \alpha = \beta$$

and $t \approx_{\Phi} u$ where $\approx_{\Phi}$ defined as before.

This works fine and enjoys a number of nice properties:

- $t \sim_{\Phi} t' \land \gamma \sim_{\Phi} \gamma' \implies t\gamma \sim_{\Phi} t'\gamma'$
- $t \rightarrow_{\beta} u \land t \sim_{\Phi} t' \land u \sim_{\Phi} u' \implies t' \rightarrow_{\beta} u'$
- $t \triangleright u \land t \sim_{\Phi} t' \land u \sim_{\Phi} u' \implies t' \triangleright u'$

However it is more complicated than the previous variant as it is really a property of typed terms and not of preterms and environments.
Solution

Definition (Equivalence)

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff there exists a partial injective function $\Phi : \text{Var} \rightarrow \text{Var}$ such that:

$$\forall x : \alpha \in \Omega(\Gamma), y : \beta \in \Omega(\Delta). x \Phi y \implies \alpha = \beta$$

and $t \approx_{\Phi} u$ where $\approx_{\Phi}$ defined as before.

This works fine and enjoys a number of nice properties:

- $t \sim_{\Phi} t' \land \gamma \sim_{\Phi} \gamma' \implies t\gamma \sim_{\Phi} t'\gamma'$
- $t \rightarrow_{\beta} u \land t \sim_{\Phi} t' \land u \sim_{\Phi} u' \implies t' \rightarrow_{\beta} u'$
- $t \succ u \land t \sim_{\Phi} t' \land u \sim_{\Phi} u' \implies t' \succ u'$

However it is more complicated than the previous variant as it is really a property of typed terms and not of preterms and environments.
Solution

Definition (Equivalence)

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff there exists a partial injective function $\Phi : \text{Var} \to \text{Var}$ such that:

$$\forall x : \alpha \in \Omega(\Gamma), y : \beta \in \Omega(\Delta). x \Phi y \implies \alpha = \beta$$

and $t \approx_{\Phi} u$ where $\approx_{\Phi}$ defined as before.

This works fine and enjoys a number of nice properties:

- $t \sim_{\Phi} t' \land \gamma \sim_{\Phi} \gamma' \implies t\gamma \sim_{\Phi} t'\gamma'$
- $t \rightarrow_{\beta} u \land t \sim_{\Phi} t' \land u \sim_{\Phi} u' \implies t' \rightarrow_{\beta} u'$
- $t \succ u \land t \sim_{\Phi} t' \land u \sim_{\Phi} u' \implies t' \succ u'$

However it is more complicated than the previous variant as it is really a property of typed terms and not of preterms and environments.
Solution

Definition (Equivalence)

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff there exists a partial injective function $\Phi : \text{Var} \rightarrow \text{Var}$ such that:

$$\forall x : \alpha \in \Omega(\Gamma), y : \beta \in \Omega(\Delta). x \Phi y \implies \alpha = \beta$$

and $t \approx_{\Phi} u$ where $\approx_{\Phi}$ defined as before.

This works fine and enjoys a number of nice properties:

- $t \sim_{\Phi} t' \land \gamma \sim_{\Phi} \gamma' \implies t\gamma \sim_{\Phi} t'\gamma'$
- $t \rightarrow_{\beta} u \land t \sim_{\Phi} t' \land u \sim_{\Phi} u' \implies t' \rightarrow_{\beta} u'$
- $t \triangleright u \land t \sim_{\Phi} t' \land u \sim_{\Phi} u' \implies t' \triangleright u'$

However it is more complicated than the previous variant as it is really a property of typed terms and not of preterms and environments.
We need to encode Φ in Coq, that is:

- a partial, injective function,
- for which we must be able to compute inversion.

Think of proving symmetry: \( t \sim_Φ u \implies u \sim_{Φ^{-1}} t \).
- In general this cannot be done in a constructive way...
- But in our case domain of Φ is finite.

Record EnvSubst : Type := build_envSub { sub: relation nat;
size: nat;
dec: forall i j, \{ sub i j \} + \{ \neg sub i j \};
lok: forall i j j', sub i j -> sub i j' -> j = j';
rok: forall i i' j, sub i j -> sub i' j -> i = i';
sok: forall i j, sub i j -> i < size \land j < size }.
We need to encode $\Phi$ in Coq, that is:

- a partial, injective function,
- for which we must be able to compute inversion.

Think of proving symmetry: $t \sim_\Phi u \implies u \sim_{\Phi^{-1}} t$.

In general this cannot be done in a constructive way...

...but in our case domain of $\Phi$ is finite.

Record EnvSubst : Type := build_envSub {
  sub: relation nat;
  size: nat;
  dec: forall i j, {sub i j} + {~sub i j};
  lok: forall i j j', sub i j -> sub i j' -> j = j';
  rok: forall i i' j, sub i j -> sub i' j -> i = i';
  sok: forall i j, sub i j -> i < size \&\& j < size
}. 

Adam Koprowski
Certified Higher-Order Recursive Path Ordering
Encoding in Coq

We need to encode Φ in Coq, that is:

- a **partial**, injective function,
- for which we must be able to compute inversion.

Think of proving symmetry: \( t \sim_\Phi u \implies u \sim_{\Phi^{-1}} t \).

- In general this cannot be done in a **constructive** way...
- ...but in our case domain of Φ is **finite**.

Record EnvSubst : Type := build_envSub {
    sub: relation nat;
    size: nat;
    dec: forall i j, \{sub i j\} + \{~sub i j\};
    lok: forall i j j’, sub i j -> sub i j’ -> j = j’;
    rok: forall i i’ j, sub i j -> sub i’ j -> i = i’;
    sok: forall i j, sub i j -> i < size /\ j < size
}.

Adam Koprowski
Certified Higher-Order Recursive Path Ordering
Encoding in Coq

We need to encode \( \Phi \) in Coq, that is:

- a **partial**, injective function,
- for which we must be able to compute **inversion**.

Think of proving symmetry: \( t \sim_\Phi u \implies u \sim_{\Phi^{-1}} t \).

- **In general this cannot be done in a constructive way**...

- ...but in our case domain of \( \Phi \) is **finite**.

Record EnvSubst : Type := build_envSub { sub: relation nat; size: nat; dec: forall i j, \{sub i j\} + \{\sim sub i j\}; lok: forall i j j’, sub i j -> sub i j’ -> j = j’; rok: forall i i’ j, sub i j -> sub i’ j -> i = i’; sok: forall i j, sub i j -> i < size /\ j < size }.
**Encoding in Coq**

We need to encode $\Phi$ in Coq, that is:

- a partial, injective function,
- for which we must be able to compute inversion.

Think of proving symmetry: $t \sim \Phi u \implies u \sim \Phi^{-1} t$.

- In general this cannot be done in a constructive way...
- ...but in our case domain of $\Phi$ is finite.

Record EnvSubst : Type := build_envSub {
  sub: relation nat;
  size: nat;
  dec: forall i j, {sub i j} + {~sub i j};
  lok: forall i j j', sub i j -> sub i j' -> j = j';
  rok: forall i i' j, sub i j -> sub i' j -> i = i';
  sok: forall i j, sub i j -> i < size \&\& j < size
}.
We need to encode $\Phi$ in Coq, that is:

- a **partial**, injective function,
- for which we must be able to compute **inversion**.
  
  Think of proving symmetry: $t \sim_{\Phi} u \implies u \sim_{\Phi^{-1}} t$.
- In general this cannot be done in a **constructive** way...
- ...but in our case domain of $\Phi$ is **finite**.

Record EnvSubst : Type := build_envSub { sub : relation nat; size : nat; dec : forall i j, {sub i j} + {~sub i j}; lok : forall i j j', sub i j -> sub i j' -> j = j'; rok : forall i i' j, sub i j -> sub i' j -> i = i'; sok : forall i j, sub i j -> i < size \ / \ j < size }.
Big developments in Coq are possible...

... but are still rather time consuming.

Often simple things turn out not to be that simple (intuition)

Working with dependent types is difficult.

Working with equality different that identity is burdensome (although Setoid tactic makes it somehow easier).

Some form of handling symmetries would be very helpful.
Summary & 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)
- Working with dependent types is difficult.
- Working with equality different that identity is burdensome (although Setoid tactic makes it somehow easier).
- Some form of handling symmetries would be very helpful.
Summary & 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)
- Working with dependent types is difficult.
- Working with equality different that identity is burdensome (although Setoid tactic makes it somehow easier).
- Some form of handling symmetries would be very helpful.
Summary & 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)
- **Working with dependent types is difficult.**
- Working with equality different that identity is burdensome (although Setoid tactic makes it somehow easier).
- Some form of handling symmetries would be very helpful.
Summary & 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)
- Working with dependent types is difficult.
- Working with equality different that identity is burdensome (although Setoid tactic makes it somehow easier).
- Some form of handling symmetries would be very helpful.
Summary & 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)
- Working with dependent types is difficult.
- Working with equality different that identity is burdensome (although Setoid tactic makes it somehow easier).
- Some form of handling symmetries would be very helpful.
Thank you for your attention.