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 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 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 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 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 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 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 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 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 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 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 α-convertibility
   - Introduction to problem
   - α-convertibility
   - Equivalence on terms

Outline

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 $\mathcal{S}$ we define simple types as:

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

Definition (Preterms)

We define preterms as:

$$\mathcal{P}t := x \mid f \mid @(\mathcal{P}t, \mathcal{P}t) \mid \lambda x : \mathcal{T} . \mathcal{P}t$$

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

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 $\mathcal{S}$ we define simple types as:

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

Definition (Preterms)

We define preterms as:

$$\mathcal{P}t := x \mid f \mid @(\mathcal{P}t, \mathcal{P}t) \mid \lambda x : \mathcal{T}.\mathcal{P}t$$

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

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 $\mathcal{S}$ we define simple types as:

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

Definition (Preterms)

We define preterms as:

$$\mathcal{P}t := x \mid f \mid @(\mathcal{P}t, \mathcal{P}t) \mid \lambda x : \mathcal{T}.\mathcal{P}t$$

Definition (Environments)

We define environment as a set of variable declarations:

$$\Gamma = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$$

technische universiteit eindhoven

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 $\mathcal{S}$ we define simple types as:

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

Definition (Preterms)

We define preterms as:

$$\mathcal{P}t := x \mid f \mid @(\mathcal{P}t, \mathcal{P}t) \mid \lambda x : \mathcal{T}.\mathcal{P}t$$

Definition (Environments)

We define environment as a set of variable declarations:

$$\Gamma = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$$

technische universiteit eindhoven

$\lambda^{\rightarrow}$ typing discipline

Definition (Typing judgements)

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

$$\frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha} \qquad \frac{f : \alpha \in \Sigma}{\Gamma \vdash f : \alpha}$$

$$\frac{\Gamma \vdash t : \alpha \rightarrow \beta \qquad \Gamma \vdash u : \alpha}{\Gamma \vdash @(t, u) : \beta} \qquad \frac{\Gamma \cup \{x : \alpha\} \vdash t : \beta}{\Gamma \vdash \lambda x : \alpha . t : \alpha \rightarrow \beta}$$

α-conversion and β-reduction

Definition (α-conversion)

α-conversion is defined as:

$$\lambda x:\alpha.t = \lambda y:\alpha.t[x := y] \text{ if } y \text{ does not appear freely in } t \text{ and } y \text{ is not bound in } t$$

α-conversions expresses the irrelevance of bound variable names.

Definition (β-reduction)

β-reduction is defined as:

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

β-reduction models computation in $\lambda^\rightarrow$.

α-conversion and β-reduction

Definition (α-conversion)

α-conversion is defined as:

$$\lambda x : \alpha . t = \lambda y : \alpha . t[x := y] \text{ if } y \text{ does not appear freely in } t \text{ and } y \text{ is not bound in } t$$

α-conversions expresses the irrelevance of bound variable names.

Definition (β-reduction)

β-reduction is defined as:

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

β-reduction models computation in $\lambda^{\rightarrow}$.

Outline

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

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 $(s_1, \ldots, s_n) \succ_{rpo}^{\tau(f)} (t_1, \ldots, t_m)$

technische universiteit eindhoven

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

technische universiteit eindhoven

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

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 $(s_1, \ldots, s_n) \succ_{rpo}^{\tau(f)} (t_1, \ldots, t_m)$

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 $\triangleright$ 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. $s_i \succeq_{rpo} t$ 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 $(s_1, \ldots, s_n) \succ_{rpo}^{\tau(f)} (t_1, \ldots, t_m)$

technische universiteit eindhoven

Recursive path order

Definition (RPO)

Given order on function symbols $\triangleright$ 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. $s_i \succeq_{rpo} t$ 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 $(s_1, \ldots, s_n) \succ_{rpo}^{\tau(f)} (t_1, \ldots, t_m)$

Theorem

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

technische universiteit eindhoven

Outline

Higher-order rewriting

There are three variants of higher-order rewriting:

HRS Higher-order rewriting systems (Nipkow)

AFS Algebraic functional systems (Jouannaud and Okada)

CRS Combinatory reduction systems (Klop)

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.

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.

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.

Examples of higher-order rewriting

Example (AFS for map)

$$\begin{array}{rcl} \mathsf{map}(\mathsf{nil}, F) & \rightarrow & \mathsf{nil} \\ \mathsf{map}(\mathsf{cons}(x, l), F) & \rightarrow & \mathsf{cons}(@(F, x), \mathsf{map}(l, F)) \end{array}$$

Example (AFS for summation)

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

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

Examples of higher-order rewriting

Example (AFS for map)

$$\begin{array}{rcl} \mathrm{map}(\mathrm{nil}, F) & \rightarrow & \mathrm{nil} \\ \mathrm{map}(\mathrm{cons}(x, l), F) & \rightarrow & \mathrm{cons}(@(F, x), \mathrm{map}(l, F)) \end{array}$$

Example (AFS for summation)

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

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

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 α-convertibility

Higher-order recursive path ordering

Definition (HORPO)

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

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

where $\succ\!\succ$ is defined as:

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

Higher-order recursive path ordering

Definition (RPO)

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

1. $s_i \succeq_{rpo} t$ for some $1 \leq i \leq n$.
2. $f \rhd g$ 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. $t = f(t_1, \ldots, t_n), \exists i \in \{1, \ldots, n\} \,.\, t_i \succeq u$
2. $t = f(t_1, \ldots, t_n), u = g(u_1, \ldots, u_k), f \rhd g, t \succ\!\succ \{u_1, \ldots u_k\}$
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. $@(u_1, \ldots, u_k)$ is a partial flattening of u, $t \succ\!\succ \{u_1, \ldots u_k\}$
5. $t = @(t_l, u_r), u = @(t_l, u_r), \{\{t_l, t_r\}\} \succ_{mul} \{\{u_l, u_r\}\}$
6. $t = \lambda x : \alpha . t', u = \lambda x : \alpha . u', t' \succ u'$

where $\succ\!\succ$ is defined as:
$t = f(t_1, \ldots, t_k) \succ\!\succ \{u_1, \ldots, u_n\}$ iff
$\forall i \in \{1, \ldots, n\} \,.\, t \succ u_i \vee (\exists j \,.\, t_j \succeq u_i)$.

technische universiteit eindhoven

Outline

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 $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$ 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).
- 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 $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$ terms.

Outline

Development overview

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 β-reduction of $\lambda^{\rightarrow}$. Hence as a corollary we get termination of $\lambda^{\rightarrow}$.

Development overview

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 β-reduction of $\lambda^{\rightarrow}$. Hence as a corollary we get termination of $\lambda^{\rightarrow}$.

Development overview

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 β-reduction of $\lambda^{\rightarrow}$. Hence as a corollary we get termination of $\lambda^{\rightarrow}$.

Development overview

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 β-reduction of $\lambda^{\rightarrow}$. Hence as a corollary we get termination of $\lambda^{\rightarrow}$.

Development overview

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 β-reduction of $\lambda^{\rightarrow}$. Hence as a corollary we get termination of $\lambda^{\rightarrow}$.

Development overview

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 β-reduction of $\lambda^{\rightarrow}$. Hence as a corollary we get termination of $\lambda^{\rightarrow}$.

Development overview

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 β-reduction of $\lambda^{\rightarrow}$. Hence as a corollary we get termination of $\lambda^{\rightarrow}$.

Development overview

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 β-reduction of $\lambda^{\rightarrow}$. Hence as a corollary we get termination of $\lambda^{\rightarrow}$.

Development overview

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 β-reduction of $\lambda^{\rightarrow}$. Hence as a corollary we get termination of $\lambda^{\rightarrow}$.

Development overview

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 β-reduction of $\lambda^{\rightarrow}$. Hence as a corollary we get termination of $\lambda^{\rightarrow}$.

J.-P. Jouannaud and A. Rubio.
The higher-order recursive path ordering.
In *Proceedings of the 14th annual IEEE Symposium on Logic in Computer Science (LICS '99)*, pages 402–411, Trento, Italy, July 1999.

J.-P. Jouannaud and A. Rubio.
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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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 α-convertibility to free variables.
  - Termination of β-reduction.
  - Encoding of algebraic terms via $\lambda^{\rightarrow}$ 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.

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 α-convertibility

Relative sizes of different parts of the development

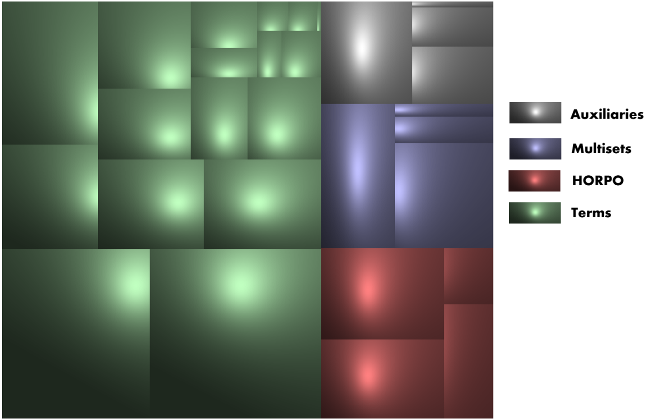


Image obtained using program SequoiaView developed at TU/e.

Development size

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.

Development size

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.

Development size

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.

Development size

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.

Development size

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 α-convertibility

Timeline of the project

Two stages of the project:

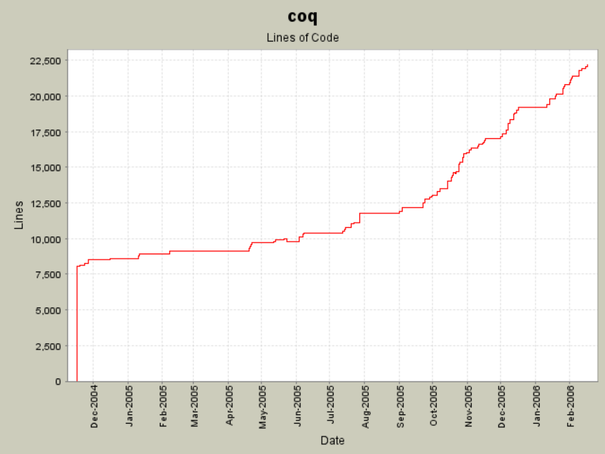
- Jan 2004 - Jul 2004
  Master Thesis at the Vrije Universiteit supervised by Femke van Raamsdonk
  Proof completed but computability properties as axioms.
- Nov 2004 - Feb 2006
  Development continued at the Technical University Eindhoven.
  Completed, axiom free proof.

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.

Timeline of the second stage of the project



Outline

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

Problem

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

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

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.

Problem

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.

Problem

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 α-convertibility
   - Introduction to problem
   - α-convertibility
   - Equivalence on terms

Tackling α-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 α-convertible terms.

Example

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

Tackling α-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 α-convertible terms.

Example

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

Tackling α-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 α-convertible terms.

Example

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

Tackling α-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 α-convertible terms.

Example

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

technische universiteit eindhoven

Tackling α-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 α-convertible terms.

Example

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

Tackling α-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 α-convertible terms.

Example

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

Tackling α-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 α-convertible terms.

Example

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

Tackling α-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 α-convertible terms.

Example

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

Tackling α-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 α-convertible terms.

Example

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

α-convertibility in Coq

- Environment simply becomes a list of types:
  `Env: list SimpleType`
- However we need dummy variables so:
  `Env: list (option SimpleType)`
- But this leads to problems...
- So we need to define custom equality for environments:

```
Definition envSubset E1 E2 := forall x A, E1 |= x := A -> E2 |= x := A.
Definition env_eq E1 E2 := envSubset E1 E2 /\ envSubset E2 E1.
```

α-convertibility in Coq

- Environment simply becomes a list of types:
  `Env: list SimpleType`
- However we need dummy variables so:
  `Env: list (option SimpleType)`
- But this leads to problems...
- So we need to define custom equality for environments:

```
Definition envSubset E1 E2 := forall x A, E1 |= x := A -> E2 |= x := A.
Definition env_eq E1 E2 := envSubset E1 E2 /\ envSubset E2 E1.
```

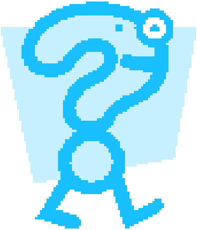
α-convertibility in Coq

- Environment simply becomes a list of types:
  `Env: list SimpleType`
- However we need dummy variables so:
  `Env: list (option SimpleType)`
- But this leads to problems...
- So we need to define custom equality for environments:

```
Definition envSubset E1 E2 := forall x A, E1 |= x := A -> E2 |= x := A.
Definition env_eq E1 E2 := envSubset E1 E2 /\ envSubset E2 E1.
```

α-convertibility in Coq

- Environment simply becomes a list of types:
  `Env: list SimpleType`
- However we need dummy variables so:
  `Env: list (option SimpleType)`
- But this leads to problems...



α-convertibility in Coq

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

```
Definition envSubset E1 E2 := forall x A, E1 |= x := A -> E2 |= x := A.
Definition env_eq E1 E2 := envSubset E1 E2 /\ envSubset E2 E1.
```

α-convertibility in Coq

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

```
Definition envSubset E1 E2 := forall x A, E1 |= x := A -> E2 |= x := A.
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 α-convertibility
   - Introduction to problem
   - α-convertibility
   - Equivalence on terms

1st naive attempt

Definition (Environment compatibility)

We say that environments Γ and Δ are compatible ($\Gamma \leftrightsquigarrow \Delta$) iff:

$$\left.\begin{array}{r} x : \alpha \in \Gamma \\ x : \beta \in \Delta \end{array}\right\} \implies \alpha = \beta$$

Definition (Equivalence)

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff: $t = u \wedge \Gamma \leftrightsquigarrow \Delta$.

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

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

$$x : \beta \vdash c : \alpha \nsim x : \gamma \vdash c : \alpha$$

1st naive attempt

Definition (Environment compatibility)

We say that environments Γ and Δ are compatible ($\Gamma \leftrightsquigarrow \Delta$) iff:

$$\left.\begin{array}{r} x : \alpha \in \Gamma \\ x : \beta \in \Delta \end{array}\right\} \implies \alpha = \beta$$

Definition (Equivalence)

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff: $t = u \wedge \Gamma \leftrightsquigarrow \Delta$.

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

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

$$x : \beta \vdash c : \alpha \not\sim x : \gamma \vdash c : \alpha$$

1st naive attempt

Definition (Environment compatibility)

We say that environments Γ and Δ are compatible ($\Gamma \leftrightsquigarrow \Delta$) iff:

$$\left.\begin{array}{r} x : \alpha \in \Gamma \\ x : \beta \in \Delta \end{array}\right\} \implies \alpha = \beta$$

Definition (Equivalence)

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff: $t = u \wedge \Gamma \leftrightsquigarrow \Delta$.

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

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

1st naive attempt

Definition (Environment compatibility)

We say that environments Γ and Δ are compatible ($\Gamma \leftrightsquigarrow \Delta$) iff:

$$\left.\begin{array}{r} x:\alpha \in \Gamma \\ x:\beta \in \Delta \end{array}\right\} \implies \alpha = \beta$$

Definition (Equivalence)

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff: $t = u \wedge \Gamma \leftrightsquigarrow \Delta$.

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

$$x{:}\beta \vdash c : \alpha \;\sim\; \emptyset \vdash c : \alpha \;\sim\; x{:}\gamma \vdash c : \alpha$$
$$x{:}\beta \vdash c : \alpha \;\nsim\; x{:}\gamma \vdash c : \alpha$$

2nd (somehow) less naive attempt

Definition (Equivalence)

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

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

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

$$\begin{array}{rcll}
x & \approx_\Phi & y & \text{if} \quad x \;\Phi\; y \\
f & \approx_\Phi & f & \\
@(t_l, t_r) & \approx_\Phi & @(u_l, u_r) & \text{if} \quad t_l \approx_\Phi u_l \wedge t_r \approx_\Phi u_r \\
\lambda\alpha.t & \approx_\Phi & \lambda\alpha.u & \text{if} \quad t \approx_{\Phi\uparrow^1} u
\end{array}$$

Problem: the following property does not hold:

$$t \approx_\Phi u \wedge \Phi \subseteq \Phi' \implies t \approx_{\Phi'} u$$

2nd (somehow) less naive attempt

Definition (Equivalence)

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

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

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

$$\begin{array}{rcll} x & \approx_\Phi & y & \text{if} \quad x \,\Phi\, y \\ f & \approx_\Phi & f & \\ @(t_l, t_r) & \approx_\Phi & @(u_l, u_r) & \text{if} \quad t_l \approx_\Phi u_l \wedge t_r \approx_\Phi u_r \\ \lambda\alpha.t & \approx_\Phi & \lambda\alpha.u & \text{if} \quad t \approx_{\Phi\uparrow 1} u \end{array}$$

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 $t \not\sim_{\{(x,x)\}} u$

2nd (somehow) less naive attempt

Definition (Equivalence)

Let $\Gamma \vdash t : \alpha \sim \Delta \vdash u : \beta$ iff there exists a partial injective function Φ : Var $\longrightarrow$ Var such that:

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

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

$$\begin{array}{rclll} x & \approx_\Phi & y & \text{if} & x \, \Phi \, y \\ f & \approx_\Phi & f & & \\ @(t_l, t_r) & \approx_\Phi & @(u_l, u_r) & \text{if} & t_l \approx_\Phi u_l \wedge t_r \approx_\Phi u_r \\ \lambda\alpha.t & \approx_\Phi & \lambda\alpha.u & \text{if} & t \approx_{\Phi\uparrow 1} u \end{array}$$

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 $t \nsim_{\{(x,x)\}} u$

Solution

Definition (Active environment)

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

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

Definition (Equivalence)

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

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

and $t \sim_\Phi u$ where $\sim_\Phi$ defined as before

Solution

Definition (Active environment)

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

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

Definition (Equivalence)

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

- $t \sim_\Phi t' \wedge \gamma \sim_\Phi \gamma' \implies t\gamma \sim_\Phi t'\gamma'$
- $t \rightarrow_\beta u \wedge t \sim_\Phi t' \wedge u \sim_\Phi u' \implies t' \rightarrow_\beta u'$
- $t \succ u \wedge t \sim_\Phi t' \wedge u \sim_\Phi u' \implies t' \succ u'$

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 Φ : Var $\longrightarrow$ 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:

- $t \sim_\Phi t' \wedge \gamma \sim_\Phi \gamma' \implies t\gamma \sim_\Phi t'\gamma'$
- $t \rightarrow_\beta u \wedge t \sim_\Phi t' \wedge u \sim_\Phi u' \implies t' \rightarrow_\beta u'$
- $t \succ u \wedge t \sim_\Phi t' \wedge u \sim_\Phi u' \implies t' \succ u'$

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 Φ : Var $\longrightarrow$ 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:

- $t \sim_\Phi t' \wedge \gamma \sim_\Phi \gamma' \implies t\gamma \sim_\Phi t'\gamma'$
- $t \rightarrow_\beta u \wedge t \sim_\Phi t' \wedge u \sim_\Phi u' \implies t' \rightarrow_\beta u'$
- $t \succ u \wedge t \sim_\Phi t' \wedge u \sim_\Phi u' \implies t' \succ u'$

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 Φ : Var $\longrightarrow$ 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:

- $t \sim_\Phi t' \wedge \gamma \sim_\Phi \gamma' \implies t\gamma \sim_\Phi t'\gamma'$
- $t \rightarrow_\beta u \wedge t \sim_\Phi t' \wedge u \sim_\Phi u' \implies t' \rightarrow_\beta u'$
- $t \succ u \wedge t \sim_\Phi t' \wedge u \sim_\Phi u' \implies t' \succ u'$

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 Φ : Var $\longrightarrow$ 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:

- $t \sim_\Phi t' \wedge \gamma \sim_\Phi \gamma' \implies t\gamma \sim_\Phi t'\gamma'$
- $t \rightarrow_\beta u \wedge t \sim_\Phi t' \wedge u \sim_\Phi u' \implies t' \rightarrow_\beta u'$
- $t \succ u \wedge t \sim_\Phi t' \wedge u \sim_\Phi u' \implies t' \succ u'$

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

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
}.
```

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
}.
```

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
}.
```

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
}.
```

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
}.
```

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
}.
```

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.

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.