Lecture 4: Dependent types and the Calculus of Constructions

Łukasz Czajka

Dependent types

- In the formal systems of logic we considered up till now, the proof terms were in a separate syntactic category.

Dependent types

- In the formal systems of logic we considered up till now, the proof terms were in a separate syntactic category.
  - Proof terms could not occur in types/formulas or in object terms/programs.

Dependent types

- In the formal systems of logic we considered up till now, the proof terms were in a separate syntactic category.
  - Proof terms could not occur in types/formulas or in object terms/programs.
  - One could "erase" the judgements $\Gamma \vdash M : \varphi$ of intuitionistic first-order logic to $|\Gamma| \vdash \varphi$ and still have a reasonable formal system.

Dependent types

- In the formal systems of logic we considered up till now, the proof terms were in a separate syntactic category.
  - Proof terms could not occur in types/formulas or in object terms/programs.
  - One could "erase" the judgements $\Gamma \vdash M : \varphi$ of intuitionistic first-order logic to $|\Gamma| \vdash \varphi$ and still have a reasonable formal system. Analogously with higher-order logic.

Dependent types

- In the formal systems of logic we considered up till now, the proof terms were in a separate syntactic category.
  - Proof terms could not occur in types/formulas or in object terms/programs.
  - One could "erase" the judgements $\Gamma \vdash M : \varphi$ of intuitionistic first-order logic to $|\Gamma| \vdash \varphi$ and still have a reasonable formal system. Analogously with higher-order logic.
- Full dependent types abolish the a priori distinction between proof terms (proofs) and object terms (programs).

Dependent types

- In the formal systems of logic we considered up till now, the proof terms were in a separate syntactic category.
  - Proof terms could not occur in types/formulas or in object terms/programs.
  - One could "erase" the judgements $\Gamma \vdash M : \varphi$ of intuitionistic first-order logic to $|\Gamma| \vdash \varphi$ and still have a reasonable formal system. Analogously with higher-order logic.
- Full dependent types abolish the a priori distinction between proof terms (proofs) and object terms (programs).
- It becomes possible to quantify over proofs (which <u>are</u> programs), and proofs (programs) may occur in types (formulas).

Dependent types

- $\forall x : \sigma.\tau$ is the type of functions which take an argument t of type σ and produce a result of type $\tau[t/x]$.

Dependent types

- $\forall x : \sigma.\tau$ is the type of functions which take an argument t of type σ and produce a result of type $\tau[t/x]$.
- The type of the result depends on the value of the argument!

Dependent types

- $\forall x : \sigma.\tau$ is the type of functions which take an argument t of type σ and produce a result of type $\tau[t/x]$.
- The type of the result depends on the value of the argument!
- $\sigma \to \tau$ is a special case of $\forall x : \sigma.\tau$ when $x \notin \mathrm{FV}(\tau)$ (i.e. x does not occur free in τ).

Intermission: the simply-typed lambda-calculus

Simple types: $\mathcal{T} ::= \mathcal{B} \mid \mathcal{T} \to \mathcal{T}$ where $\mathcal{B}$ is a fixed finite set of type constants.

Intermission: the simply-typed lambda-calculus

Simple types: $\mathcal{T} ::= \mathcal{B} \mid \mathcal{T} \to \mathcal{T}$ where $\mathcal{B}$ is a fixed finite set of type constants.

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \tau}$$

$$\frac{\Gamma \cup \{x : \tau\} \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau . t : \tau \to \sigma} \qquad \frac{\Gamma \vdash t_1 : \tau \to \sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \sigma}$$

Intermission: the simply-typed lambda-calculus

Simple types: $\mathcal{T} ::= \mathcal{B} \mid \mathcal{T} \to \mathcal{T}$ where $\mathcal{B}$ is a fixed finite set of type constants.

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \tau}$$

$$\frac{\Gamma \cup \{x : \tau\} \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau . t : \tau \to \sigma} \qquad \frac{\Gamma \vdash t_1 : \tau \to \sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \sigma}$$

- β-reduction: $(\lambda x : \tau . t) t' \to_\beta t[t'/x]$.

Intermission: the simply-typed lambda-calculus

Simple types: $\mathcal{T} ::= \mathcal{B} \mid \mathcal{T} \to \mathcal{T}$ where $\mathcal{B}$ is a fixed finite set of type constants.

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \tau}$$

$$\frac{\Gamma \cup \{x : \tau\} \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau . t : \tau \to \sigma} \qquad \frac{\Gamma \vdash t_1 : \tau \to \sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \sigma}$$

- β-reduction: $(\lambda x : \tau . t)t' \to_\beta t[t'/x]$.
- Subject reduction theorem: if $\Gamma \vdash t : \tau$ and $t \to_\beta^* t'$ then $\Gamma \vdash t' : \tau$.

Intermission: the simply-typed lambda-calculus

Simple types: $\mathcal{T} ::= \mathcal{B} \mid \mathcal{T} \to \mathcal{T}$ where $\mathcal{B}$ is a fixed finite set of type constants.

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \tau}$$

$$\frac{\Gamma \cup \{x : \tau\} \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau.t : \tau \to \sigma} \qquad \frac{\Gamma \vdash t_1 : \tau \to \sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \sigma}$$

- β-reduction: $(\lambda x : \tau.t)t' \to_\beta t[t'/x]$.
- Subject reduction theorem: if $\Gamma \vdash t : \tau$ and $t \to_\beta^* t'$ then $\Gamma \vdash t' : \tau$.
- Strong normalisation theorem: if $\Gamma \vdash t : \tau$ then every reduction sequence starting from t ends in a β-normal form (i.e., in a term with no β-redexes).

Intermission: the simply-typed lambda-calculus

Simple types: $\mathcal{T} ::= \mathcal{B} \mid \mathcal{T} \to \mathcal{T}$ where $\mathcal{B}$ is a fixed finite set of type constants.

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \tau}$$

$$\frac{\Gamma \cup \{x : \tau\} \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau . t : \tau \to \sigma} \qquad \frac{\Gamma \vdash t_1 : \tau \to \sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \sigma}$$

- β-reduction: $(\lambda x : \tau . t)t' \to_\beta t[t'/x]$.
- Subject reduction theorem: if $\Gamma \vdash t : \tau$ and $t \to_\beta^* t'$ then $\Gamma \vdash t' : \tau$.
- Strong normalisation theorem: if $\Gamma \vdash t : \tau$ then every reduction sequence starting from t ends in a β-normal form (i.e., in a term with no β-redexes).
- Uniqueness of normal forms: if t_1, t_2 are in β-normal form and $t_1 =_\beta t_2$, then $t_1 = t_2$.

Intermission: the simply-typed lambda-calculus

Simple types: $\mathcal{T} ::= \mathcal{B} \mid \mathcal{T} \to \mathcal{T}$ where $\mathcal{B}$ is a fixed finite set of type constants.

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \tau}$$

$$\frac{\Gamma \cup \{x : \tau\} \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau . t : \tau \to \sigma} \qquad \frac{\Gamma \vdash t_1 : \tau \to \sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \sigma}$$

- β-reduction: $(\lambda x : \tau . t) t' \to_\beta t[t'/x]$.
- Subject reduction theorem: if $\Gamma \vdash t : \tau$ and $t \to_\beta^* t'$ then $\Gamma \vdash t' : \tau$.
- Strong normalisation theorem: if $\Gamma \vdash t : \tau$ then every reduction sequence starting from t ends in a β-normal form (i.e., in a term with no β-redexes).
- Uniqueness of normal forms: if t_1, t_2 are in β-normal form and $t_1 =_\beta t_2$, then $t_1 = t_2$.
- Exercise: β-equality on simply-typed terms is decidable.

Intermission: the simply-typed lambda-calculus

Let's assume the elements of $\mathcal{B}$ are ordinary variables and $t_1 \to t_2$ is just another form of terms. Let $*$ be the universe of types.

$$\frac{\alpha \in \mathcal{B}}{\Gamma \vdash \alpha : *} \qquad \frac{\Gamma \vdash \tau : * \quad \Gamma \vdash \sigma : *}{\Gamma \vdash \tau \to \sigma : *}$$

Intermission: the simply-typed lambda-calculus

Let's assume the elements of $\mathcal{B}$ are ordinary variables and $t_1 \to t_2$ is just another form of terms. Let $*$ be the universe of types.

$$\frac{\alpha \in \mathcal{B}}{\Gamma \vdash \alpha : *} \qquad \frac{\Gamma \vdash \tau : * \quad \Gamma \vdash \sigma : *}{\Gamma \vdash \tau \to \sigma : *}$$

Let the contexts be <u>sequences</u> instead of sets.

$$\frac{\Gamma \vdash \tau : * \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \sigma : * \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \sigma \vdash t : \tau}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau . t : \tau \to \sigma} \qquad \frac{\Gamma \vdash t_1 : \tau \to \sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \sigma}$$

Dependent types: the system λP

- A term t, τ, σ is a variable x, y, z, α, β, a universe $u \in \mathcal{U}$, an application $t_1 t_2$, a lambda-abstraction $\lambda x : \tau.t$, or a dependent function type $\forall x : \sigma.\tau$.

Dependent types: the system λP

- A term t, τ, σ is a variable x, y, z, α, β, a universe $u \in \mathcal{U}$, an application $t_1 t_2$, a lambda-abstraction $\lambda x : \tau.t$, or a dependent function type $\forall x : \sigma.\tau$.
- A context Γ is a finite sequence of declarations $x : \tau$.

Dependent types: the system λP

- A <u>term</u> t, τ, σ is a variable x, y, z, α, β, a universe $u \in \mathcal{U}$, an application $t_1 t_2$, a lambda-abstraction $\lambda x : \tau . t$, or a dependent function type $\forall x : \sigma . \tau$.
- A <u>context</u> Γ is a finite <u>sequence</u> of declarations $x : \tau$.
  - The <u>order</u> matters!

Dependent types: the system λP

- A term t, τ, σ is a variable x, y, z, α, β, a universe $u \in \mathcal{U}$, an application $t_1 t_2$, a lambda-abstraction $\lambda x : \tau.t$, or a dependent function type $\forall x : \sigma.\tau$.
- A context Γ is a finite sequence of declarations $x : \tau$.
  - The order matters!
  - We denote the empty sequence by $\langle\rangle$.

Dependent types: the system λP

- A term t, τ, σ is a variable x, y, z, α, β, a universe $u \in \mathcal{U}$, an application $t_1 t_2$, a lambda-abstraction $\lambda x : \tau . t$, or a dependent function type $\forall x : \sigma . \tau$.
- A context Γ is a finite sequence of declarations $x : \tau$.
  - The order matters!
  - We denote the empty sequence by $\langle\rangle$.
  - By $\mathrm{dom}(\Gamma)$ we denote the set of all variables declared in Γ.

Dependent types: the system λP

- A <u>term</u> t, τ, σ is a variable x, y, z, α, β, a universe $u \in \mathcal{U}$, an application $t_1 t_2$, a lambda-abstraction $\lambda x : \tau . t$, or a dependent function type $\forall x : \sigma . \tau$.
- A <u>context</u> Γ is a finite <u>sequence</u> of declarations $x : \tau$.
  - The <u>order</u> matters!
  - We denote the empty sequence by $\langle\rangle$.
  - By $\mathrm{dom}(\Gamma)$ we denote the set of all variables declared in Γ.
- A <u>judgement</u> has the form $\Gamma \vdash t : \tau$ with Γ context, t, τ terms.

Dependent types: definitional equality

- β-reduction: $(\lambda x : \tau.t)t' \to_\beta t[t'/x]$.

Dependent types: definitional equality

- β-reduction: $(\lambda x : \tau.t)t' \to_\beta t[t'/x]$.
- η-reduction: $\lambda x : \tau.tx \to_\eta t$ if $x \notin \mathrm{FV}(t)$.

Dependent types: definitional equality

- β-reduction: $(\lambda x : \tau.t)t' \to_\beta t[t'/x]$.
- η-reduction: $\lambda x : \tau.tx \to_\eta t$ if $x \notin \mathrm{FV}(t)$.
- Definitional equality $\equiv$ is defined as $\beta\eta$-equality.

Dependent types: the system λP

$$\frac{(\mathcal{U}_1, \mathcal{U}_2) \in \mathcal{A}}{\langle\rangle \vdash \mathcal{U}_1 : \mathcal{U}_2}$$

$$\frac{\Gamma \vdash \tau : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \sigma : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \sigma \vdash t : \tau}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma \quad \Gamma \vdash (\forall x : \tau.\sigma) : \mathcal{U}}{\Gamma \vdash (\lambda x : \tau.t) : \forall x : \tau.\sigma} \qquad \frac{\Gamma \vdash t_1 : \forall x : \tau.\sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \sigma[t_2/x]}$$

$$\frac{\Gamma \vdash \tau : \mathcal{U}_1 \quad \Gamma, x : \tau \vdash \sigma : \mathcal{U}_2 \quad (\mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3) \in \mathcal{R}}{\Gamma \vdash (\forall x : \tau.\sigma) : \mathcal{U}_3}$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \tau' : \mathcal{U} \quad \tau \equiv \tau'}{\Gamma \vdash t : \tau'}$$

- Universes: $\mathcal{U} = \{*, \Box\}$.
- Axioms: $\mathcal{A} = \{(*, \Box)\}$.
- Rules: $\mathcal{R} = \{(*, *, *), (*, \Box, \Box)\}$.

Notation

- We write $\Gamma \vdash t_1 : t_2 : t_3$ if $\Gamma \vdash t_1 : t_2$ and $\Gamma \vdash t_2 : t_3$.

Notation

- We write $\Gamma \vdash t_1 : t_2 : t_3$ if $\Gamma \vdash t_1 : t_2$ and $\Gamma \vdash t_2 : t_3$.
- We omit Γ when obvious or irrelevant, writing e.g. $t_1 : t_2$, $t_1 : t_2 : t_3$.

Notation

- We write $\Gamma \vdash t_1 : t_2 : t_3$ if $\Gamma \vdash t_1 : t_2$ and $\Gamma \vdash t_2 : t_3$.
- We omit Γ when obvious or irrelevant, writing e.g. $t_1 : t_2$, $t_1 : t_2 : t_3$.
- Unless stated otherwise, we consider only legal terms and contexts (i.e. those which appear in some derivable judgement).

Dependent types: universes of λP

- Universe of types: $*$.

Dependent types: universes of λP

- Universe of types: $*$.
  - Objects (proofs) have types (formulas/propositions).

Dependent types: universes of λP

- Universe of types: $*$.
  - Objects (proofs) have types (formulas/propositions).
  - If $t : \tau : *$ then t is an object of type τ (t is a proof of τ).

Dependent types: universes of λP

- Universe of types: $*$.
    - Objects (proofs) have types (formulas/propositions).
    - If $t : \tau : *$ then t is an object of type τ (t is a proof of τ).
    - E.g.: $\alpha : * \vdash (\lambda x : \alpha . x) : \alpha \to \alpha : *$.

Dependent types: universes of λP

- Universe of types: $*$.
  - Objects (proofs) have types (formulas/propositions).
  - If $t : \tau : *$ then t is an object of type τ (t is a proof of τ).
  - E.g.: $\alpha : * \vdash (\lambda x : \alpha . x) : \alpha \to \alpha : *$.
- Universe of kinds: $\Box$.

Dependent types: universes of λP

- Universe of types: $*$.
  - Objects (proofs) have types (formulas/propositions).
  - If $t : \tau : *$ then t is an object of type τ (t is a proof of τ).
  - E.g.: $\alpha : * \vdash (\lambda x : \alpha . x) : \alpha \to \alpha : *$.
- Universe of kinds: $\Box$.
  - Type constructors (predicates) have kinds.

Dependent types: universes of λP

- Universe of types: $*$.
  - Objects (proofs) have types (formulas/propositions).
  - If $t : \tau : *$ then t is an object of type τ (t is a proof of τ).
  - E.g.: $\alpha : * \vdash (\lambda x : \alpha . x) : \alpha \to \alpha : *$.
- Universe of kinds: $\Box$.
  - Type constructors (predicates) have kinds.
  - If $\tau : \kappa : \Box$ then τ is a type constructor (predicate) of kind κ.

Dependent types: universes of λP

- Universe of types: $*$.
  - Objects (proofs) have types (formulas/propositions).
  - If $t : \tau : *$ then t is an object of type τ (t is a proof of τ).
  - E.g.: $\alpha : * \vdash (\lambda x : \alpha . x) : \alpha \to \alpha : *$.
- Universe of kinds: $\Box$.
  - Type constructors (predicates) have kinds.
  - If $\tau : \kappa : \Box$ then τ is a type constructor (predicate) of kind κ.
  - E.g.: $\alpha : *, P : \alpha \to \alpha \to * \vdash (\lambda x : \alpha . Pxx) : \alpha \to * : \Box$.

Dependent types: universes of λP

- Universe of types: $*$.
  - Objects (proofs) have types (formulas/propositions).
  - If $t : \tau : *$ then t is an object of type τ (t is a proof of τ).
  - E.g.: $\alpha : * \vdash (\lambda x : \alpha . x) : \alpha \to \alpha : *$.
- Universe of kinds: $\Box$.
  - Type constructors (predicates) have kinds.
  - If $\tau : \kappa : \Box$ then τ is a type constructor (predicate) of kind κ.
  - E.g.: $\alpha : *, P : \alpha \to \alpha \to * \vdash (\lambda x : \alpha . Pxx) : \alpha \to * : \Box$.
- The universe of types is a kind: $* : \Box$ because $(*, \Box) \in \mathcal{A}$.

Dependent types: universes of λP

- Universe of types: $*$.
  - Objects (proofs) have types (formulas/propositions).
  - If $t : \tau : *$ then t is an object of type τ (t is a proof of τ).
  - E.g.: $\alpha : * \vdash (\lambda x : \alpha.x) : \alpha \to \alpha : *$.
- Universe of kinds: $\Box$.
  - Type constructors (predicates) have kinds.
  - If $\tau : \kappa : \Box$ then τ is a type constructor (predicate) of kind κ.
  - E.g.: $\alpha : *, P : \alpha \to \alpha \to * \vdash (\lambda x : \alpha.Pxx) : \alpha \to * : \Box$.
- The universe of types is a kind: $* : \Box$ because $(*, \Box) \in \mathcal{A}$. So each type (formula/proposition) is a type constructor (nullary predicate).

Dependent types: λP

$\Box$						
$*$			$\alpha \to *$	$(\alpha \to \alpha) \to \alpha \to *$	$\dots$	
α	$(\forall x : \alpha.Px) \to Py$	$\dots$	$\lambda x : \alpha.Px$	$\lambda f : \alpha \to \alpha.\lambda x : \alpha.P(fx)$	$\dots$	
y	$\lambda f : \forall x : \alpha.Px.fy$	$\dots$	—			

In the context: $\alpha : *, P : \alpha \to *, y : \alpha, p : \forall x : \alpha.Px$.

Dependent types: rules of λP

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

Dependent types: rules of λP

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

Let $\Gamma = \alpha : *, P : \alpha \to *$.

Dependent types: rules of λP

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

Let $\Gamma = \alpha : *, P : \alpha \to *$.

- $\Gamma \vdash \alpha : *$.

Dependent types: rules of λP

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

Let $\Gamma = \alpha : *, P : \alpha \to *$.

- $\Gamma \vdash \alpha : *$.

$$\frac{\Gamma \vdash \tau : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau}$$

Dependent types: rules of λP

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

Let $\Gamma = \alpha : *, P : \alpha \to *$.

- $\Gamma \vdash \alpha : *$.

$$\frac{\Gamma \vdash \tau : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau}$$

- $\Gamma, x : \alpha \vdash x : \alpha$.

Dependent types: rules of λP

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

Let $\Gamma = \alpha : *, P : \alpha \to *$.

- $\Gamma \vdash \alpha : *$.

$$\frac{\Gamma \vdash \tau : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau}$$

- $\Gamma, x : \alpha \vdash x : \alpha$.
- $\Gamma \vdash (\forall x : \alpha . Px) : *$.

Dependent types: rules of λP

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau.\sigma) : *}$$

Let $\Gamma = \alpha : *, P : \alpha \to *$.

- $\Gamma \vdash \alpha : *$.

$$\frac{\Gamma \vdash \tau : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau}$$

- $\Gamma, x : \alpha \vdash x : \alpha$.
- $\Gamma \vdash (\forall x : \alpha.Px) : *$. For this we need:
  - $\Gamma \vdash \alpha : *$, and
  - $\Gamma, x : \alpha \vdash Px : *$.

Dependent types: rules of λP

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau.\sigma) : *}$$

Let $\Gamma = \alpha : *, P : \alpha \to *$.

- $\Gamma \vdash \alpha : *$.

$$\frac{\Gamma \vdash \tau : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau}$$

- $\Gamma, x : \alpha \vdash x : \alpha$.
- $\Gamma \vdash (\forall x : \alpha.Px) : *$. For this we need:
  - $\Gamma \vdash \alpha : *$, and
  - $\Gamma, x : \alpha \vdash Px : *$. For this we need:
    - $\Gamma, x : \alpha \vdash P : \alpha \to *$, and
    - $\Gamma, x : \alpha \vdash x : \alpha$.

Dependent types: rules of λP

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau.\sigma) : *}$$

Let $\Gamma = \alpha : *, P : \alpha \to *$.

- $\Gamma \vdash \alpha : *$.

$$\frac{\Gamma \vdash \tau : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau}$$

- $\Gamma, x : \alpha \vdash x : \alpha$.
- $\Gamma \vdash (\forall x : \alpha.Px) : *$. For this we need:
  - $\Gamma \vdash \alpha : *$, and
  - $\Gamma, x : \alpha \vdash Px : *$. For this we need:
    - $\Gamma, x : \alpha \vdash P : \alpha \to *$, and
    - $\Gamma, x : \alpha \vdash x : \alpha$.

But how do we actually derive $\Gamma \vdash P : \alpha \to *$?

Dependent types: rules of λP

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

Let $\Gamma = \alpha : *, P : \alpha \to *$.

- $\Gamma \vdash \alpha : *$.

$$\frac{\Gamma \vdash \tau : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau}$$

- $\Gamma, x : \alpha \vdash x : \alpha$.
- $\Gamma \vdash (\forall x : \alpha . Px) : *$. For this we need:
  - $\Gamma \vdash \alpha : *$, and
  - $\Gamma, x : \alpha \vdash Px : *$. For this we need:
    - $\Gamma, x : \alpha \vdash P : \alpha \to *$, and
    - $\Gamma, x : \alpha \vdash x : \alpha$.

But how do we actually derive $\Gamma \vdash P : \alpha \to *$?

- For this we need $\alpha : * \vdash (\alpha \to *) : \Box$.

Dependent types: rules of λP

Types depend on objects: $(*, \Box, \Box) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau . \sigma) : \Box}$$

Dependent types: rules of λP

Types depend on objects: $(*, \Box, \Box) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau.\sigma) : \Box}$$

$$\frac{\dfrac{\vdots}{\alpha : * \vdash \alpha : *} \quad \dfrac{\vdots}{\alpha : *, x : \alpha \vdash * : \Box}}{\alpha : * \vdash \alpha \to * : \Box}$$

Dependent types: rules of λP

Types depend on objects: $(*, \Box, \Box) \in \mathcal{R}$.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau . \sigma) : \Box}$$

$$\dfrac{\dfrac{\vdots}{\alpha : * \vdash \alpha : *} \quad \dfrac{\vdots}{\alpha : *, x : \alpha \vdash * : \Box}}{\alpha : * \vdash \alpha \to * : \Box}$$

The rule $(*, \Box, \Box)$ allows us to have "predicates" in the context (but not to quantify over them). λP is essentially a "first-order" system.

λP vs first-order logic

Consider the universal-implicational fragment FOL$\forall\to$ of the system of inituitionistic first-order logic from the second lecture.

$$\frac{}{\Gamma, X : \varphi \vdash X : \varphi}$$

$$\frac{\Gamma, X : \varphi_1 \vdash M : \varphi_2}{\Gamma \vdash (\lambda X : \varphi_1.M) : \varphi_1 \to \varphi_2} \qquad \frac{\Gamma \vdash M_1 : \varphi \to \psi \quad \Gamma \vdash M_2 : \varphi}{\Gamma \vdash M_1 M_2 : \psi}$$

$$\frac{\Gamma \vdash M : \varphi \quad x : A \quad x \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash (\lambda x : A.M) : \forall x : A.\varphi} \qquad \frac{\Gamma \vdash M : \forall x : A.\varphi \quad t : A}{\Gamma \vdash Mt : \varphi[t/x]}$$

λP vs first-order logic

Consider the universal-implicational fragment FOL$\forall\!\to$ of the system of inituitionistic first-order logic from the second lecture.

$$\frac{}{\Gamma, X : \varphi \vdash X : \varphi}$$

$$\frac{\Gamma, X : \varphi_1 \vdash M : \varphi_2}{\Gamma \vdash (\lambda X : \varphi_1 . M) : \varphi_1 \to \varphi_2} \qquad \frac{\Gamma \vdash M_1 : \varphi \to \psi \quad \Gamma \vdash M_2 : \varphi}{\Gamma \vdash M_1 M_2 : \psi}$$

$$\frac{\Gamma \vdash M : \varphi \quad x : A \quad x \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash (\lambda x : A . M) : \forall x : A . \varphi} \qquad \frac{\Gamma \vdash M : \forall x : A . \varphi \quad t : A}{\Gamma \vdash M t : \varphi[t/x]}$$

Assuming the proof and object variables and the domains of FOL$\forall\!\to$ are variables in λP, we define a translation from FOL$\forall\!\to$ to λP:

- $\lceil X \rceil = X$, $\lceil x \rceil = x$, $\lceil M_1 M_2 \rceil = \lceil M_1 \rceil \lceil M_2 \rceil$, $\lceil M t \rceil = \lceil M \rceil \lceil t \rceil$, $\lceil \lambda x : A . M \rceil = \lambda x : A . \lceil M \rceil$, $\lceil \lambda X : \varphi . M \rceil = \lambda X : \lceil \varphi \rceil . \lceil M \rceil$.
- $\lceil A \rceil = A$, $\lceil \varphi \to \psi \rceil = \lceil \varphi \rceil \to \lceil \psi \rceil$, $\lceil \forall x : A . \varphi \rceil = \forall x : A . \lceil \varphi \rceil$.

λP vs first-order logic

$$\begin{aligned}&\lceil \Gamma \vdash M : \varphi \rceil = \\ &\quad A_1 : *, \ldots, A_n : *, a_1 : A_1, \ldots, a_n : A_n, x_1 : A_{x_1}, \ldots, x_m : A_{x_m}, \\ &\qquad X_1 : \lceil \psi_1 \rceil, \ldots, X_k : \lceil \psi_k \rceil \vdash \lceil M \rceil : \lceil \varphi \rceil\end{aligned}$$

λP vs first-order logic

$$\begin{aligned}&\lceil \Gamma \vdash M : \varphi \rceil = \\ &\quad A_1 : *, \dots, A_n : *, a_1 : A_1, \dots, a_n : A_n, x_1 : A_{x_1}, \dots, x_m : A_{x_m}, \\ &\qquad X_1 : \lceil \psi_1 \rceil, \dots, X_k : \lceil \psi_k \rceil \vdash \lceil M \rceil : \lceil \varphi \rceil\end{aligned}$$

where:

- $A_1, \dots, A_n$ are all of the domains of object variables (free and bound) occurring in Γ, M, φ,

λP vs first-order logic

$$
\begin{aligned}
&\lceil \Gamma \vdash M : \varphi \rceil = \\
&\quad A_1 : *, \ldots, A_n : *, a_1 : A_1, \ldots, a_n : A_n, x_1 : A_{x_1}, \ldots, x_m : A_{x_m}, \\
&\qquad X_1 : \lceil \psi_1 \rceil, \ldots, X_k : \lceil \psi_k \rceil \vdash \lceil M \rceil : \lceil \varphi \rceil
\end{aligned}
$$

where:

- $A_1, \ldots, A_n$ are all of the domains of object variables (free and bound) occurring in Γ, M, φ,
- $a_1, \ldots, a_n$ are distinct fresh variables,

λP vs first-order logic

$$
\begin{aligned}
&\lceil \Gamma \vdash M : \varphi \rceil = \\
&\quad A_1 : *, \ldots, A_n : *, a_1 : A_1, \ldots, a_n : A_n, x_1 : A_{x_1}, \ldots, x_m : A_{x_m}, \\
&\qquad X_1 : \lceil \psi_1 \rceil, \ldots, X_k : \lceil \psi_k \rceil \vdash \lceil M \rceil : \lceil \varphi \rceil
\end{aligned}
$$

where:

- $A_1, \ldots, A_n$ are all of the domains of object variables (free and bound) occurring in Γ, M, φ,
- $a_1, \ldots, a_n$ are distinct fresh variables,
- $x_1, \ldots, x_n$ are all of the free object variables occurring in Γ, M, φ with domains $A_{x_1}, \ldots, A_{x_m}$ respectively,

λP vs first-order logic

$$
\begin{aligned}
&\lceil \Gamma \vdash M : \varphi \rceil = \\
&\quad A_1 : *, \ldots, A_n : *, a_1 : A_1, \ldots, a_n : A_n, x_1 : A_{x_1}, \ldots, x_m : A_{x_m}, \\
&\qquad X_1 : \lceil \psi_1 \rceil, \ldots, X_k : \lceil \psi_k \rceil \vdash \lceil M \rceil : \lceil \varphi \rceil
\end{aligned}
$$

where:

- $A_1, \ldots, A_n$ are all of the domains of object variables (free and bound) occurring in Γ, M, φ,
- $a_1, \ldots, a_n$ are distinct fresh variables,
- $x_1, \ldots, x_n$ are all of the free object variables occurring in Γ, M, φ with domains $A_{x_1}, \ldots, A_{x_m}$ respectively,
- $\Gamma = \{X_1 : \psi_1, \ldots, X_k : \psi_1\}$.

λP vs first-order logic

$$\begin{aligned}&\lceil \Gamma \vdash M : \varphi \rceil = \\ &\quad A_1 : *, \ldots, A_n : *, a_1 : A_1, \ldots, a_n : A_n, x_1 : A_{x_1}, \ldots, x_m : A_{x_m}, \\ &\qquad X_1 : \lceil \psi_1 \rceil, \ldots, X_k : \lceil \psi_k \rceil \vdash \lceil M \rceil : \lceil \varphi \rceil\end{aligned}$$

where:

- $A_1, \ldots, A_n$ are all of the domains of object variables (free and bound) occurring in Γ, M, φ,
- $a_1, \ldots, a_n$ are distinct fresh variables,
- $x_1, \ldots, x_n$ are all of the free object variables occurring in Γ, M, φ with domains $A_{x_1}, \ldots, A_{x_m}$ respectively,
- $\Gamma = \{X_1 : \psi_1, \ldots, X_k : \psi_1\}$.

Theorem (Soundness of translation from FOL$\forall\!\to$ to λP)

If $\Gamma \vdash M : \varphi$ *is derivable in FOL*$\forall\!\to$ *then* $\lceil \Gamma \vdash M : \varphi \rceil$ *is derivable in* λP.

λP vs first-order logic

But there are terms and types of λP which have no counterpart in first-order logic!

λP vs first-order logic

But there are terms and types of λP which have no counterpart in first-order logic!

- Quantification over higher-order functions (but not predicates!), e.g.:

$$\alpha : *, P : \alpha \to * \vdash_{\lambda\mathrm{P}} (\forall f : (\alpha \to \alpha) \to \alpha . P(f(\lambda x : \alpha . x))) : *$$

λP vs first-order logic

But there are terms and types of λP which have no counterpart in first-order logic!

- Quantification over higher-order functions (but not predicates!), e.g.:

$$\alpha : *, P : \alpha \to * \vdash_{\lambda \mathrm{P}} (\forall f : (\alpha \to \alpha) \to \alpha . P(f(\lambda x : \alpha . x))) : *$$

- Formulas (types) can refer to properties of proofs (dependently typed programs), e.g.:

$$\alpha : *, P : \alpha \to *, Q : (\forall y : \alpha . Py) \to * \vdash_{\lambda \mathrm{P}} (\forall x : (\forall y : \alpha . Py) . Qx) : *$$

λP vs first-order logic

But there are terms and types of λP which have no counterpart in first-order logic!

- Quantification over higher-order functions (but not predicates!), e.g.:

$$\alpha : *, P : \alpha \to * \vdash_{\lambda P} (\forall f : (\alpha \to \alpha) \to \alpha . P(f(\lambda x : \alpha . x))) : *$$

- Formulas (types) can refer to properties of proofs (dependently typed programs), e.g.:

$$\alpha : *, P : \alpha \to *, Q : (\forall y : \alpha . Py) \to * \vdash_{\lambda P} (\forall x : (\forall y : \alpha . Py) . Qx) : *$$

- Domains of quantifications may be empty, in contrast to "ordinary" first-order logic where they are implicitly assumed to be non-empty. E.g.: $(\forall x : \tau . \psi) \to \psi$ with $x \notin \mathrm{FV}(\psi)$ is not inhabited unless we can construct an element of τ, even though the corresponding first-order formula $\forall x \psi \to \psi$ is an intuitionistic tautology when $x \notin \mathrm{FV}(\psi)$.

Pure Type Systems

$$\frac{(u_1, u_2) \in \mathcal{A}}{\langle\rangle \vdash u_1 : u_2}$$

$$\frac{\Gamma \vdash \tau : u \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \sigma : u \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \sigma \vdash t : \tau}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma \quad \Gamma \vdash (\forall x : \tau.\sigma) : u}{\Gamma \vdash (\lambda x : \tau.t) : \forall x : \tau.\sigma} \qquad \frac{\Gamma \vdash t_1 : \forall x : \tau.\sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \tau[t_2/x]}$$

$$\frac{\Gamma \vdash \tau : u_1 \quad \Gamma, x : \tau \vdash \sigma : u_2 \quad (u_1, u_2, u_3) \in \mathcal{R}}{\Gamma \vdash (\forall x : \tau.\sigma) : u_3}$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \tau' : u \quad \tau \equiv \tau'}{\Gamma \vdash t : \tau'}$$

Pure Type Systems

$$\frac{(u_1, u_2) \in \mathcal{A}}{\langle\rangle \vdash u_1 : u_2}$$

$$\frac{\Gamma \vdash \tau : u \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \sigma : u \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \sigma \vdash t : \tau}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma \quad \Gamma \vdash (\forall x : \tau.\sigma) : u}{\Gamma \vdash (\lambda x : \tau.t) : \forall x : \tau.\sigma} \qquad \frac{\Gamma \vdash t_1 : \forall x : \tau.\sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \tau[t_2/x]}$$

$$\frac{\Gamma \vdash \tau : u_1 \quad \Gamma, x : \tau \vdash \sigma : u_2 \quad (u_1, u_2, u_3) \in \mathcal{R}}{\Gamma \vdash (\forall x : \tau.\sigma) : u_3}$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \tau' : u \quad \tau \equiv \tau'}{\Gamma \vdash t : \tau'}$$

- Simply-typed lambda-calculus $\lambda{\rightarrow}$: $\mathcal{U} = \{*, \Box\}$, $\mathcal{A} = \{(*, \Box)\}$, $\mathcal{R} = \{(*, *, *)\}$.

Pure Type Systems

$$\frac{(\mathcal{U}_1, \mathcal{U}_2) \in \mathcal{A}}{\langle\rangle \vdash \mathcal{U}_1 : \mathcal{U}_2}$$

$$\frac{\Gamma \vdash \tau : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \sigma : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \sigma \vdash t : \tau}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma \quad \Gamma \vdash (\forall x : \tau.\sigma) : \mathcal{U}}{\Gamma \vdash (\lambda x : \tau.t) : \forall x : \tau.\sigma} \qquad \frac{\Gamma \vdash t_1 : \forall x : \tau.\sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \tau[t_2/x]}$$

$$\frac{\Gamma \vdash \tau : \mathcal{U}_1 \quad \Gamma, x : \tau \vdash \sigma : \mathcal{U}_2 \quad (\mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3) \in \mathcal{R}}{\Gamma \vdash (\forall x : \tau.\sigma) : \mathcal{U}_3}$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \tau' : \mathcal{U} \quad \tau \equiv \tau'}{\Gamma \vdash t : \tau'}$$

- Simply-typed lambda-calculus $\lambda\!\!\rightarrow$: $\mathcal{U} = \{*, \Box\}$, $\mathcal{A} = \{(*, \Box)\}$, $\mathcal{R} = \{(*, *, *)\}$.
- Intuitionistic higher-order logic λHOL: $\mathcal{U} = \{*, \Box, \triangle\}$, $\mathcal{A} = \{(*, \Box), (\Box, \triangle)\}$, $\mathcal{R} = \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box)\}$.

Pure Type Systems: reduction

- The (many-step) $\beta\eta$-reduction relation $\to^*_{\beta\eta}$ is the transitive-reflexive closure of $\to_{\beta\eta} = \to_\beta \cup \to_\eta$.

Pure Type Systems: reduction

- The (many-step) $\beta\eta$-reduction relation $\rightarrow^*_{\beta\eta}$ is the transitive-reflexive closure of $\rightarrow_{\beta\eta} = \rightarrow_\beta \cup \rightarrow_\eta$.
- A term t is in <u>$\beta\eta$-normal form</u> if it contains no <u>$\beta\eta$-redexes</u>, i.e., there is no term t' such that $t \rightarrow_{\beta\eta} t'$.

Pure Type Systems: reduction

- The (many-step) $\beta\eta$-reduction relation $\to^*_{\beta\eta}$ is the transitive-reflexive closure of $\to_{\beta\eta} = \to_\beta \cup \to_\eta$.
- A term t is in <u>$\beta\eta$-normal form</u> if it contains no <u>$\beta\eta$-redexes</u>, i.e., there is no term t' such that $t \to_{\beta\eta} t'$.
- A term t is <u>weakly $\beta\eta$-normalising</u> (has a $\beta\eta$-normal form), denoted $\mathrm{WN}_{\beta\eta}(t)$, if there is a $\beta\eta$-normal form t' such that $t \to^*_{\beta\eta} t'$. Analogously $\mathrm{WN}_\beta(t)$.

Pure Type Systems: reduction

- The (many-step) $\beta\eta$-reduction relation $\rightarrow^*_{\beta\eta}$ is the transitive-reflexive closure of $\rightarrow_{\beta\eta} = \rightarrow_\beta \cup \rightarrow_\eta$.
- A term t is in <u>$\beta\eta$-normal form</u> if it contains no <u>$\beta\eta$-redexes</u>, i.e., there is no term t' such that $t \rightarrow_{\beta\eta} t'$.
- A term t is <u>weakly $\beta\eta$-normalising</u> (has a $\beta\eta$-normal form), denoted $\mathrm{WN}_{\beta\eta}(t)$, if there is a $\beta\eta$-normal form t' such that $t \rightarrow^*_{\beta\eta} t'$. Analogously $\mathrm{WN}_\beta(t)$.
- A term t is <u>strongly $\beta\eta$-normalising</u>, denoted $\mathrm{SN}_{\beta\eta}(t)$, if there are no infinite $\beta\eta$-reduction sequences starting from t, i.e., no infinite sequences of terms $\{t_i\}_{i\in\mathbb{N}}$ such that $t_0 = t$ and $t_i \rightarrow_{\beta\eta} t_{i+1}$ for $n \in \mathbb{N}$.

Pure Type Systems: reduction

- The (many-step) $\beta\eta$-reduction relation $\rightarrow^*_{\beta\eta}$ is the transitive-reflexive closure of $\rightarrow_{\beta\eta} = \rightarrow_\beta \cup \rightarrow_\eta$.
- A term t is in <u>$\beta\eta$-normal form</u> if it contains no <u>$\beta\eta$-redexes</u>, i.e., there is no term t' such that $t \rightarrow_{\beta\eta} t'$.
- A term t is <u>weakly $\beta\eta$-normalising</u> (has a $\beta\eta$-normal form), denoted $\mathrm{WN}_{\beta\eta}(t)$, if there is a $\beta\eta$-normal form t' such that $t \rightarrow^*_{\beta\eta} t'$. Analogously $\mathrm{WN}_\beta(t)$.
- A term t is <u>strongly $\beta\eta$-normalising</u>, denoted $\mathrm{SN}_{\beta\eta}(t)$, if there are no infinite $\beta\eta$-reduction sequences starting from t, i.e., no infinite sequences of terms $\{t_i\}_{i\in\mathbb{N}}$ such that $t_0 = t$ and $t_i \rightarrow_{\beta\eta} t_{i+1}$ for $n \in \mathbb{N}$.
- A PTS is strongly (resp. weakly) normalising if every legal term is strongly (resp. weakly) normalising.

Exercise: postponement of η-reduction

Proposition

If t is strongly (resp. weakly) β-normalising, then it is strongly (resp. weakly) $\beta\eta$-normalising.

Exercise: postponement of η-reduction

Proposition

If t is strongly (resp. weakly) β-normalising, then it is strongly (resp. weakly) $\beta\eta$-normalising.

Proof (sketch).

For strong normalisation, show that if $t_1 \to_\eta t_2 \to_\beta t_3$ then there is t' with $t_1 \to_\beta^+ t' \to_\eta^* t_3$. For weak normalisation, it suffices to prove that η-reduction is normalising and that η-reducing a β-normal form produces a β-normal form. $\square$

Pure Type Systems: properties

Theorem (Subject reduction for β)

In any PTS, if $\Gamma \vdash t : \tau$ *and* $t \rightarrow_\beta^* t'$ *then* $\Gamma \vdash t' : \tau$.

Pure Type Systems: properties

Theorem (Subject reduction for β)

In any PTS, if $\Gamma \vdash t : \tau$ *and* $t \to^*_\beta t'$ *then* $\Gamma \vdash t' : \tau$.

Theorem (Subject reduction for $\beta\eta$)

In any weakly normalising PTS, if $\Gamma \vdash t : \tau$ *and* $t \to^*_{\beta\eta} t'$ *then* $\Gamma \vdash t' : \tau$.

Pure Type Systems: properties

Theorem (Subject reduction for β)

In any PTS, if $\Gamma \vdash t : \tau$ *and* $t \to_\beta^* t'$ *then* $\Gamma \vdash t' : \tau$*.*

Theorem (Subject reduction for $\beta\eta$)

In any weakly normalising PTS, if $\Gamma \vdash t : \tau$ *and* $t \to_{\beta\eta}^* t'$ *then* $\Gamma \vdash t' : \tau$*.*

Theorem (Uniqueness of normal forms)

In any PTS, if t_1, t_2 *are* <u>*legal*</u> *(well-typed)* $\beta\eta$*-normal forms such that* $t_1 =_{\beta\eta} t_2$*, then* $t_1 = t_2$*.*

Pure Type Systems: benefits of normalisation

Theorem (Decidability of type checking)

In any weakly normalising PTS, type checking is decidable.

Pure Type Systems: benefits of normalisation

Theorem (Decidability of type checking)

In any weakly normalising PTS, type checking is decidable.

Theorem (Consistency)

Any weakly normalising PTS is consistent, i.e., there is no term t with $\vdash t : \forall p : *.p$.

Calculus of Constructions

$$\frac{(\mathcal{U}_1, \mathcal{U}_2) \in \mathcal{A}}{\langle\rangle \vdash \mathcal{U}_1 : \mathcal{U}_2}$$

$$\frac{\Gamma \vdash \tau : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \sigma : \mathcal{U} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : \sigma \vdash t : \tau}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma \quad \Gamma \vdash (\forall x : \tau.\sigma) : \mathcal{U}}{\Gamma \vdash (\lambda x : \tau.t) : \forall x : \tau.\sigma} \qquad \frac{\Gamma \vdash t_1 : \forall x : \tau.\sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \tau[t_2/x]}$$

$$\frac{\Gamma \vdash \tau : \mathcal{U}_1 \quad \Gamma, x : \tau \vdash \sigma : \mathcal{U}_2 \quad (\mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3) \in \mathcal{R}}{\Gamma \vdash (\forall x : \tau.\sigma) : \mathcal{U}_3}$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \tau' : \mathcal{U} \quad \tau \equiv \tau'}{\Gamma \vdash t : \tau'}$$

Calculus of Constructions λC: $\mathcal{U} = \{*, \Box\}$, $\mathcal{A} = \{(*, \Box)\}$,
$\mathcal{R} = \{(*, *, *), (*, \Box, \Box), (\Box, *, *), (\Box, \Box, \Box)\}$.

Calculus of Constructions

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

Calculus of Constructions

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

- Logic view: implication.
- Programming view: simple function types.

Calculus of Constructions

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

- Logic view: implication.
- Programming view: simple function types.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

Calculus of Constructions

Objects depend on objects: $(*, *, *) \in \mathcal{R}$.

- Logic view: implication.
- Programming view: simple function types.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

$$\alpha : * \vdash (\lambda x : \alpha . x) : \alpha \to \alpha : *$$

Calculus of Constructions

Types depend on objects: $(*, \Box, \Box) \in \mathcal{R}$.

Calculus of Constructions

Types depend on objects: $(*, \Box, \Box) \in \mathcal{R}$.

- Logic view: quantification over objects; formation of first-order predicates.
- Programming view: dependent function types (the type of the result may depend on the value of the argument); type constructors with object arguments.

Calculus of Constructions

Types depend on objects: $(*, \Box, \Box) \in \mathcal{R}$.

- Logic view: quantification over objects; formation of first-order predicates.
- Programming view: dependent function types (the type of the result may depend on the value of the argument); type constructors with object arguments.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau . \sigma) : \Box}$$

Calculus of Constructions

Types depend on objects: $(*, \Box, \Box) \in \mathcal{R}$.

- Logic view: quantification over objects; formation of first-order predicates.
- Programming view: dependent function types (the type of the result may depend on the value of the argument); type constructors with object arguments.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau . \sigma) : \Box}$$

$$\alpha : * \vdash \alpha \to \alpha \to * : \Box$$

Calculus of Constructions

Types depend on objects: $(*, \Box, \Box) \in \mathcal{R}$.

- Logic view: quantification over objects; formation of first-order predicates.
- Programming view: dependent function types (the type of the result may depend on the value of the argument); type constructors with object arguments.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau . \sigma) : \Box}$$

$$\alpha : * \vdash \alpha \to \alpha \to * : \Box$$
$$\alpha : *, p : \alpha \to \alpha \to * \vdash (\lambda x : \alpha . pxx) : \alpha \to * : \Box$$

Calculus of Constructions

Types depend on objects: $(*, \Box, \Box) \in \mathcal{R}$.

- Logic view: quantification over objects; formation of first-order predicates.
- Programming view: dependent function types (the type of the result may depend on the value of the argument); type constructors with object arguments.

$$\frac{\Gamma \vdash \tau : * \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau . \sigma) : \Box}$$

$$\begin{array}{l}
\alpha : * \vdash \alpha \to \alpha \to * : \Box \\
\alpha : *, p : \alpha \to \alpha \to * \vdash (\lambda x : \alpha . pxx) : \alpha \to * : \Box \\
\alpha : *, p : \alpha \to * \vdash (\lambda x : \alpha . \lambda q : px . q) : (\forall x : \alpha . px \to px) : *
\end{array}$$

Calculus of Constructions

Objects depend on types: $(\Box, *, *) \in \mathcal{R}$.

Calculus of Constructions

Objects depend on types: $(\Box, *, *) \in \mathcal{R}$.

- Logic view: quantification over predicates.
- Programming view: impredicative polymorphism.

Calculus of Constructions

Objects depend on types: $(\square, *, *) \in \mathcal{R}$.

- Logic view: quantification over predicates.
- Programming view: impredicative polymorphism.

$$\frac{\Gamma \vdash \tau : \square \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

Calculus of Constructions

Objects depend on types: $(\Box, *, *) \in \mathcal{R}$.

- Logic view: quantification over predicates.
- Programming view: impredicative polymorphism.

$$\frac{\Gamma \vdash \tau : \Box \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

$(\lambda p : *. \lambda x : p.x) : (\forall p : *. p \to p) : *$

Calculus of Constructions

Objects depend on types: $(\Box, *, *) \in \mathcal{R}$.

- Logic view: quantification over predicates.
- Programming view: impredicative polymorphism.

$$\frac{\Gamma \vdash \tau : \Box \quad \Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

$(\lambda p : *.\lambda x : p.x) : (\forall p : *.p \to p) : *$
$(\lambda \alpha : *.\lambda p : \alpha \to *.\lambda x : \alpha.\lambda q : px.q) : (\forall \alpha : *.\forall p : \alpha \to *.\forall x : \alpha.px \to px) : *$

Calculus of Constructions

Types depend on types: $(\Box, \Box, \Box) \in \mathcal{R}$.

Calculus of Constructions

Types depend on types: $(\Box, \Box, \Box) \in \mathcal{R}$.

- Logic view: formation of higher-order predicates.
- Programming view: type constructors with type arguments.

Calculus of Constructions

Types depend on types: $(\Box, \Box, \Box) \in \mathcal{R}$.

- Logic view: formation of higher-order predicates.
- Programming view: type constructors with type arguments.

$$\frac{\Gamma \vdash \tau : \Box \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau . \sigma) : \Box}$$

Calculus of Constructions

Types depend on types: $(\Box, \Box, \Box) \in \mathcal{R}$.

- Logic view: formation of higher-order predicates.
- Programming view: type constructors with type arguments.

$$\frac{\Gamma \vdash \tau : \Box \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau . \sigma) : \Box}$$

$(\lambda p : *. p \to p) : * \to * : \Box$

Calculus of Constructions

Types depend on types: $(\square, \square, \square) \in \mathcal{R}$.

- Logic view: formation of higher-order predicates.
- Programming view: type constructors with type arguments.

$$\frac{\Gamma \vdash \tau : \square \quad \Gamma, x : \tau \vdash \sigma : \square}{\Gamma \vdash (\forall x : \tau . \sigma) : \square}$$

$(\lambda p : *. p \to p) : * \to * : \square$
$\alpha : * \vdash (\lambda p : \alpha \to \alpha \to *. \lambda x : \alpha . pxx) : (\alpha \to \alpha \to *) \to \alpha \to * : \square$

Calculus of Constructions

Types depend on types: $(\Box, \Box, \Box) \in \mathcal{R}$.

- Logic view: formation of higher-order predicates.
- Programming view: type constructors with type arguments.

$$\frac{\Gamma \vdash \tau : \Box \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau . \sigma) : \Box}$$

$(\lambda p : *. p \to p) : * \to * : \Box$
$\alpha : * \vdash (\lambda p : \alpha \to \alpha \to *. \lambda x : \alpha . pxx) : (\alpha \to \alpha \to *) \to \alpha \to * : \Box$
$(\lambda \alpha : *. \lambda p : \alpha \to \alpha \to *. \lambda x : \alpha . pxx) : \forall \alpha : *. (\alpha \to \alpha \to *) \to \alpha \to * : \Box$

Calculus of Constructions

Types depend on types: $(\Box, \Box, \Box) \in \mathcal{R}$.

- Logic view: formation of higher-order predicates.
- Programming view: type constructors with type arguments.

$$\frac{\Gamma \vdash \tau : \Box \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau.\sigma) : \Box}$$

$$\begin{array}{l}
(\lambda p : *.p \to p) : * \to * : \Box \\
\alpha : * \vdash (\lambda p : \alpha \to \alpha \to *.\lambda x : \alpha.pxx) : (\alpha \to \alpha \to *) \to \alpha \to * : \Box \\
(\lambda \alpha : *.\lambda p : \alpha \to \alpha \to *.\lambda x : \alpha.pxx) : \forall \alpha : *.(\alpha \to \alpha \to *) \to \alpha \to * : \Box \\
(\forall q : (\forall \alpha : *.\alpha \to \alpha \to *).\forall \beta : *.\forall x : \beta.q\beta xx \to *) : \Box
\end{array}$$

Calculus of Constructions

Types depend on types: $(\Box, \Box, \Box) \in \mathcal{R}$.

- Logic view: formation of higher-order predicates.
- Programming view: type constructors with type arguments.

$$\frac{\Gamma \vdash \tau : \Box \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau . \sigma) : \Box}$$

$$
\begin{array}{l}
(\lambda p : *.p \to p) : * \to * : \Box \\
\alpha : * \vdash (\lambda p : \alpha \to \alpha \to *.\lambda x : \alpha.pxx) : (\alpha \to \alpha \to *) \to \alpha \to * : \Box \\
(\lambda \alpha : *.\lambda p : \alpha \to \alpha \to *.\lambda x : \alpha.pxx) : \forall \alpha : *.(\alpha \to \alpha \to *) \to \alpha \to * : \Box \\
(\forall q : (\forall \alpha : *.\alpha \to \alpha \to *).\forall \beta : *.\forall x : \beta.q\beta xx \to *) : \Box \\
(\lambda \alpha : *.\lambda p : \alpha \to *.\lambda r : (\alpha \to *) \to *.\lambda q : rp.q) : \\
\quad (\forall \alpha : *.\forall p : \alpha \to *.\forall r : (\alpha \to *) \to *.rp \to r(\lambda x : \alpha.px)) : *
\end{array}
$$

Calculus of Constructions

Types depend on types: $(\Box, \Box, \Box) \in \mathcal{R}$.

- Logic view: formation of higher-order predicates.
- Programming view: type constructors with type arguments.

$$\frac{\Gamma \vdash \tau : \Box \quad \Gamma, x : \tau \vdash \sigma : \Box}{\Gamma \vdash (\forall x : \tau . \sigma) : \Box}$$

$$(\lambda p : *. p \to p) : * \to * : \Box$$
$$\alpha : * \vdash (\lambda p : \alpha \to \alpha \to *. \lambda x : \alpha . pxx) : (\alpha \to \alpha \to *) \to \alpha \to * : \Box$$
$$(\lambda \alpha : *. \lambda p : \alpha \to \alpha \to *. \lambda x : \alpha . pxx) : \forall \alpha : *. (\alpha \to \alpha \to *) \to \alpha \to * : \Box$$
$$(\forall q : (\forall \alpha : *. \alpha \to \alpha \to *). \forall \beta : *. \forall x : \beta . q\beta xx \to *) : \Box$$
$$(\lambda \alpha : *. \lambda p : \alpha \to *. \lambda r : (\alpha \to *) \to *. \lambda q : rp.q) :$$
$$\quad (\forall \alpha : *. \forall p : \alpha \to *. \forall r : (\alpha \to *) \to *. rp \to r(\lambda x : \alpha . px)) : *$$
$$(\lambda \alpha : *. \forall p : \alpha \to \alpha \to *. \forall r : (\alpha \to *) \to *. r(\lambda x : \alpha . pxx) \to r(\lambda x : \alpha . pxx)) :$$
$$\quad * \to *$$

Calculus of Constructions vs higher-order logic

- Calculus of Constructions λC: $\mathcal{U} = \{*, \Box\}$, $\mathcal{A} = \{(*, \Box)\}$, $\mathcal{R} = \{(*, *, *), (*, \Box, \Box), (\Box, *, *), (\Box, \Box, \Box)\}$.
- Intuitionistic higher-order logic λHOL: $\mathcal{U} = \{*, \Box, \triangle\}$, $\mathcal{A} = \{(*, \Box), (\Box, \triangle)\}$, $\mathcal{R} = \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box)\}$.

Calculus of Constructions vs higher-order logic

- Calculus of Constructions λC: $\mathcal{U} = \{*, \square\}$, $\mathcal{A} = \{(*, \square)\}$, $\mathcal{R} = \{(*, *, *), (*, \square, \square), (\square, *, *), (\square, \square, \square)\}$.
- Intuitionistic higher-order logic λHOL: $\mathcal{U} = \{*, \square, \triangle\}$, $\mathcal{A} = \{(*, \square), (\square, \triangle)\}$, $\mathcal{R} = \{(*, *, *), (\square, *, *), (\square, \square, \square)\}$.

A sound translation from λHOL to λC:

- $\lceil * \rceil = *$, $\lceil \square \rceil = *$, $\lceil \triangle \rceil = \square$, $\lceil x \rceil = x$, $\lceil t_1 t_2 \rceil = \lceil t_1 \rceil \lceil t_2 \rceil$, $\lceil \lambda x : \tau . t \rceil = \lambda x : \lceil \tau \rceil . \lceil t \rceil$.

Calculus of Constructions vs higher-order logic

- Calculus of Constructions λC: $\mathcal{U} = \{*, \Box\}$, $\mathcal{A} = \{(*, \Box)\}$, $\mathcal{R} = \{(*, *, *), (*, \Box, \Box), (\Box, *, *), (\Box, \Box, \Box)\}$.
- Intuitionistic higher-order logic λHOL: $\mathcal{U} = \{*, \Box, \triangle\}$, $\mathcal{A} = \{(*, \Box), (\Box, \triangle)\}$, $\mathcal{R} = \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box)\}$.

A sound translation from λHOL to λC:

- $\lceil * \rceil = *$, $\lceil \Box \rceil = *$, $\lceil \triangle \rceil = \Box$, $\lceil x \rceil = x$, $\lceil t_1 t_2 \rceil = \lceil t_1 \rceil \lceil t_2 \rceil$, $\lceil \lambda x : \tau . t \rceil = \lambda x : \lceil \tau \rceil . \lceil t \rceil$.
- $\lceil \langle\rangle \rceil = \langle\rangle$, $\lceil \Gamma, x : \tau \rceil = \lceil \Gamma \rceil, x : \lceil \tau \rceil$.

Calculus of Constructions vs higher-order logic

- Calculus of Constructions λC: $\mathcal{U} = \{*, \Box\}$, $\mathcal{A} = \{(*, \Box)\}$, $\mathcal{R} = \{(*, *, *), (*, \Box, \Box), (\Box, *, *), (\Box, \Box, \Box)\}$.
- Intuitionistic higher-order logic λHOL: $\mathcal{U} = \{*, \Box, \triangle\}$, $\mathcal{A} = \{(*, \Box), (\Box, \triangle)\}$, $\mathcal{R} = \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box)\}$.

A sound translation from λHOL to λC:

- $\lceil * \rceil = *$, $\lceil \Box \rceil = *$, $\lceil \triangle \rceil = \Box$, $\lceil x \rceil = x$, $\lceil t_1 t_2 \rceil = \lceil t_1 \rceil \lceil t_2 \rceil$, $\lceil \lambda x : \tau . t \rceil = \lambda x : \lceil \tau \rceil . \lceil t \rceil$.
- $\lceil \langle\rangle \rceil = \langle\rangle$, $\lceil \Gamma, x : \tau \rceil = \lceil \Gamma \rceil, x : \lceil \tau \rceil$.
- $\lceil \Gamma \vdash t : \tau \rceil = \lceil \Gamma \rceil \vdash \lceil t \rceil : \lceil \tau \rceil$ if $\tau \notin \{\Box, \triangle\}$.

Calculus of Constructions vs higher-order logic

- Calculus of Constructions λC: $\mathcal{U} = \{*, \Box\}$, $\mathcal{A} = \{(*, \Box)\}$, $\mathcal{R} = \{(*, *, *), (*, \Box, \Box), (\Box, *, *), (\Box, \Box, \Box)\}$.
- Intuitionistic higher-order logic λHOL: $\mathcal{U} = \{*, \Box, \triangle\}$, $\mathcal{A} = \{(*, \Box), (\Box, \triangle)\}$, $\mathcal{R} = \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box)\}$.

A sound translation from λHOL to λC:

- $\lceil * \rceil = *$, $\lceil \Box \rceil = *$, $\lceil \triangle \rceil = \Box$, $\lceil x \rceil = x$, $\lceil t_1 t_2 \rceil = \lceil t_1 \rceil \lceil t_2 \rceil$, $\lceil \lambda x : \tau . t \rceil = \lambda x : \lceil \tau \rceil . \lceil t \rceil$.
- $\lceil \langle\rangle \rceil = \langle\rangle$, $\lceil \Gamma, x : \tau \rceil = \lceil \Gamma \rceil, x : \lceil \tau \rceil$.
- $\lceil \Gamma \vdash t : \tau \rceil = \lceil \Gamma \rceil \vdash \lceil t \rceil : \lceil \tau \rceil$ if $\tau \notin \{\Box, \triangle\}$.
- $\lceil \Gamma \vdash \tau : \Box \rceil = \lceil \Gamma \rceil \vdash \lceil \tau \rceil : *$ if $\tau = \tau_1 \to \ldots \to \tau_n \to \alpha$ with α a variable.

Calculus of Constructions vs higher-order logic

- Calculus of Constructions λC: $\mathcal{U} = \{*, \Box\}$, $\mathcal{A} = \{(*, \Box)\}$, $\mathcal{R} = \{(*, *, *), (*, \Box, \Box), (\Box, *, *), (\Box, \Box, \Box)\}$.
- Intuitionistic higher-order logic λHOL: $\mathcal{U} = \{*, \Box, \triangle\}$, $\mathcal{A} = \{(*, \Box), (\Box, \triangle)\}$, $\mathcal{R} = \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box)\}$.

A sound translation from λHOL to λC:

- $\lceil * \rceil = *$, $\lceil \Box \rceil = *$, $\lceil \triangle \rceil = \Box$, $\lceil x \rceil = x$, $\lceil t_1 t_2 \rceil = \lceil t_1 \rceil \lceil t_2 \rceil$, $\lceil \lambda x : \tau . t \rceil = \lambda x : \lceil \tau \rceil . \lceil t \rceil$.
- $\lceil \langle \rangle \rceil = \langle \rangle$, $\lceil \Gamma, x : \tau \rceil = \lceil \Gamma \rceil, x : \lceil \tau \rceil$.
- $\lceil \Gamma \vdash t : \tau \rceil = \lceil \Gamma \rceil \vdash \lceil t \rceil : \lceil \tau \rceil$ if $\tau \notin \{\Box, \triangle\}$.
- $\lceil \Gamma \vdash \tau : \Box \rceil = \lceil \Gamma \rceil \vdash \lceil \tau \rceil : *$ if $\tau = \tau_1 \to \ldots \to \tau_n \to \alpha$ with α a variable.
- $\lceil \Gamma \vdash \tau : \Box \rceil = \lceil \Gamma \rceil \vdash \lceil \tau \rceil : \Box$ if $\tau = \tau_1 \to \ldots \to \tau_n \to *$.

Calculus of Constructions vs higher-order logic

- Calculus of Constructions λC: $\mathcal{U} = \{*, \Box\}$, $\mathcal{A} = \{(*, \Box)\}$, $\mathcal{R} = \{(*, *, *), (*, \Box, \Box), (\Box, *, *), (\Box, \Box, \Box)\}$.
- Intuitionistic higher-order logic λHOL: $\mathcal{U} = \{*, \Box, \triangle\}$, $\mathcal{A} = \{(*, \Box), (\Box, \triangle)\}$, $\mathcal{R} = \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box)\}$.

A sound translation from λHOL to λC:

- $\lceil * \rceil = *$, $\lceil \Box \rceil = *$, $\lceil \triangle \rceil = \Box$, $\lceil x \rceil = x$, $\lceil t_1 t_2 \rceil = \lceil t_1 \rceil \lceil t_2 \rceil$, $\lceil \lambda x : \tau . t \rceil = \lambda x : \lceil \tau \rceil . \lceil t \rceil$.
- $\lceil \langle \rangle \rceil = \langle \rangle$, $\lceil \Gamma, x : \tau \rceil = \lceil \Gamma \rceil, x : \lceil \tau \rceil$.
- $\lceil \Gamma \vdash t : \tau \rceil = \lceil \Gamma \rceil \vdash \lceil t \rceil : \lceil \tau \rceil$ if $\tau \notin \{\Box, \triangle\}$.
- $\lceil \Gamma \vdash \tau : \Box \rceil = \lceil \Gamma \rceil \vdash \lceil \tau \rceil : *$ if $\tau = \tau_1 \to \ldots \to \tau_n \to \alpha$ with α a variable.
- $\lceil \Gamma \vdash \tau : \Box \rceil = \lceil \Gamma \rceil \vdash \lceil \tau \rceil : \Box$ if $\tau = \tau_1 \to \ldots \to \tau_n \to *$.
- $\lceil \Gamma \vdash \Box : \triangle \rceil = \lceil \Gamma \rceil \vdash * : \Box$.

`Prop` and `Set`

- Sometimes (e.g. for program extraction) it may be desirable to distinguish between proofs and programs.

Prop and Set

- Sometimes (e.g. for program extraction) it may be desirable to distinguish between proofs and programs.
- In Coq, the universe $*$ is split into two: `Prop` and `Set`.

Universes

- In Coq, $\Box$ is called $\texttt{Type}_1$.

Universes

- In Coq, $\Box$ is called $\texttt{Type}_1$.
- There is an infinite hierarchy of universes:
  $\texttt{Type}_1 : \texttt{Type}_2 : \texttt{Type}_3 : \ldots$

Universes

- In Coq, $\square$ is called $\texttt{Type}_1$.
- There is an infinite hierarchy of universes: $\texttt{Type}_1 : \texttt{Type}_2 : \texttt{Type}_3 : \dots$
- $\texttt{Prop} : \texttt{Type}_1$ and $\texttt{Set} : \texttt{Type}_1$.

Universes

- In Coq, $\Box$ is called $\texttt{Type}_1$.
- There is an infinite hierarchy of universes:
  $\texttt{Type}_1 : \texttt{Type}_2 : \texttt{Type}_3 : \ldots$
- $\texttt{Prop} : \texttt{Type}_1$ and $\texttt{Set} : \texttt{Type}_1$.
- The index i in $\texttt{Type}_i$ is implicit in Coq.

Universes

- In Coq, $\Box$ is called $\texttt{Type}_1$.
- There is an infinite hierarchy of universes:
  $\texttt{Type}_1 : \texttt{Type}_2 : \texttt{Type}_3 : \ldots$
- $\texttt{Prop} : \texttt{Type}_1$ and $\texttt{Set} : \texttt{Type}_1$.
- The index i in $\texttt{Type}_i$ is implicit in Coq.
- Let $\texttt{Type}_0 = \texttt{Set}$.

Universes: rules

- $(\texttt{Type}_i, \texttt{Type}_j, \texttt{Type}_{\max(i,j)}) \in \mathcal{R}$ for $i, j \geq 0$.

$$\frac{\Gamma \vdash \sigma : \texttt{Type}_i \quad \Gamma, x : \sigma \vdash \tau : \texttt{Type}_j}{\Gamma \vdash (\forall x : \sigma . \tau) : \texttt{Type}_{\max(i,j)}}$$

Universes: rules

- $(\texttt{Type}_i, \texttt{Type}_j, \texttt{Type}_{\max(i,j)}) \in \mathcal{R}$ for $i, j \geq 0$.

$$\frac{\Gamma \vdash \sigma : \texttt{Type}_i \quad \Gamma, x : \sigma \vdash \tau : \texttt{Type}_j}{\Gamma \vdash (\forall x : \sigma . \tau) : \texttt{Type}_{\max(i,j)}}$$

- $(\texttt{Prop}, \texttt{Type}_i, \texttt{Type}_i) \in \mathcal{R}$ for $i \geq 0$.

$$\frac{\Gamma \vdash \sigma : \texttt{Prop} \quad \Gamma, x : \sigma \vdash \tau : \texttt{Type}_i}{\Gamma \vdash (\forall x : \sigma . \tau) : \texttt{Type}_i}$$

Universes: rules

- $(\texttt{Type}_i, \texttt{Type}_j, \texttt{Type}_{\max(i,j)}) \in \mathcal{R}$ for $i, j \geq 0$.

$$\frac{\Gamma \vdash \sigma : \texttt{Type}_i \quad \Gamma, x : \sigma \vdash \tau : \texttt{Type}_j}{\Gamma \vdash (\forall x : \sigma.\tau) : \texttt{Type}_{\max(i,j)}}$$

- $(\texttt{Prop}, \texttt{Type}_i, \texttt{Type}_i) \in \mathcal{R}$ for $i \geq 0$.

$$\frac{\Gamma \vdash \sigma : \texttt{Prop} \quad \Gamma, x : \sigma \vdash \tau : \texttt{Type}_i}{\Gamma \vdash (\forall x : \sigma.\tau) : \texttt{Type}_i}$$

- $(\mathcal{U}, \texttt{Prop}, \texttt{Prop}) \in \mathcal{R}$ for any universe $\mathcal{U}$.

$$\frac{\Gamma \vdash \sigma : \mathcal{U} \quad \Gamma, x : \sigma \vdash \tau : \texttt{Prop}}{\Gamma \vdash (\forall x : \sigma.\tau) : \texttt{Prop}}$$

Impredicativity

Definition

A universe $\mathcal{U}_1$ with $\mathcal{U}_1 : \mathcal{U}_2$ is <u>impredicative</u> if $(\mathcal{U}_2, \mathcal{U}_1, \mathcal{U}_1) \in \mathcal{R}$, i.e.,

$$\frac{\Gamma \vdash \kappa : \mathcal{U}_2 \quad \Gamma, \alpha : \kappa \vdash \tau : \mathcal{U}_1}{\Gamma \vdash (\forall \alpha : \kappa . \tau) : \mathcal{U}_1}$$

is a typing rule.

Impredicativity

Definition

A universe $\mathcal{U}_1$ with $\mathcal{U}_1 : \mathcal{U}_2$ is impredicative if $(\mathcal{U}_2, \mathcal{U}_1, \mathcal{U}_1) \in \mathcal{R}$, i.e.,

$$\frac{\Gamma \vdash \kappa : \mathcal{U}_2 \quad \Gamma, \alpha : \kappa \vdash \tau : \mathcal{U}_1}{\Gamma \vdash (\forall \alpha : \kappa . \tau) : \mathcal{U}_1}$$

is a typing rule.

- In particular, in an impredicative universe $\mathcal{U}$ a type quantifying over the whole universe $\mathcal{U}$ can be formed.

Impredicativity

Definition

A universe $\mathcal{U}_1$ with $\mathcal{U}_1 : \mathcal{U}_2$ is <u>impredicative</u> if $(\mathcal{U}_2, \mathcal{U}_1, \mathcal{U}_1) \in \mathcal{R}$, i.e.,

$$\frac{\Gamma \vdash \kappa : \mathcal{U}_2 \quad \Gamma, \alpha : \kappa \vdash \tau : \mathcal{U}_1}{\Gamma \vdash (\forall \alpha : \kappa . \tau) : \mathcal{U}_1}$$

is a typing rule.

- In particular, in an impredicative universe $\mathcal{U}$ a type quantifying over the whole universe $\mathcal{U}$ can be formed.
- So $(\forall \alpha : \mathcal{U}.\tau) : \mathcal{U}$ can be instantiated with itself: if $t : \forall \alpha : \mathcal{U}.\tau$ then $t(\forall \alpha : \mathcal{U}.\tau) : \tau[(\forall \alpha : \mathcal{U}.\tau)/\alpha]$.

Impredicativity

Definition

A universe $\mathcal{U}_1$ with $\mathcal{U}_1 : \mathcal{U}_2$ is <u>impredicative</u> if $(\mathcal{U}_2, \mathcal{U}_1, \mathcal{U}_1) \in \mathcal{R}$, i.e.,

$$\frac{\Gamma \vdash \kappa : \mathcal{U}_2 \quad \Gamma, \alpha : \kappa \vdash \tau : \mathcal{U}_1}{\Gamma \vdash (\forall \alpha : \kappa . \tau) : \mathcal{U}_1}$$

is a typing rule.

- In particular, in an impredicative universe $\mathcal{U}$ a type quantifying over the whole universe $\mathcal{U}$ can be formed.
- So $(\forall \alpha : \mathcal{U}.\tau) : \mathcal{U}$ can be instantiated with itself: if $t : \forall \alpha : \mathcal{U}.\tau$ then $t(\forall \alpha : \mathcal{U}.\tau) : \tau[(\forall \alpha : \mathcal{U}.\tau)/\alpha]$.
- For a <u>predicative</u> universe $\mathcal{U}_1$ with $\mathcal{U}_1 : \mathcal{U}_2$ we have $(\mathcal{U}_2, \mathcal{U}_1, \mathcal{U}_2) \in \mathcal{R}$ instead, i.e.,

$$\frac{\Gamma \vdash \kappa : \mathcal{U}_2 \quad \Gamma, \alpha : \kappa \vdash \tau : \mathcal{U}_1}{\Gamma \vdash (\forall \alpha : \kappa . \tau) : \mathcal{U}_2}$$

Impredicativity

In Coq, only `Prop` is impredicative.

Impredicativity

In Coq, only `Prop` is impredicative. Can't we have more impredicative universes?

Impredicativity

In Coq, only `Prop` is impredicative. Can't we have more impredicative universes?

Definition

The PTS $\lambda\mathrm{U}^-$ is defined by: $\mathcal{U} = \{*, \Box, \triangle\}$, $\mathcal{A} = \{(*, \Box), (\Box, \triangle)\}$, $\mathcal{R} = \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box), (\triangle, \Box, \Box)\}$.

Impredicativity

In Coq, only `Prop` is impredicative. Can't we have more impredicative universes?

Definition

The PTS $\lambda\mathrm{U}^-$ is defined by: $\mathcal{U} = \{*, \Box, \triangle\}$, $\mathcal{A} = \{(*, \Box), (\Box, \triangle)\}$, $\mathcal{R} = \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box), (\triangle, \Box, \Box)\}$.

Theorem (Girard's paradox)

The system $\lambda\mathrm{U}^-$ *is inconsistent, i.e.,* $\vdash t : \forall p : *.p$ *is derivable for some* t*.*

Impredicativity

In Coq, only `Prop` is impredicative. Can't we have more impredicative universes?

Definition

The PTS $\lambda\mathrm{U}^-$ is defined by: $\mathcal{U} = \{*, \Box, \triangle\}$, $\mathcal{A} = \{(*, \Box), (\Box, \triangle)\}$, $\mathcal{R} = \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box), (\triangle, \Box, \Box)\}$.

Theorem (Girard's paradox)

The system $\lambda\mathrm{U}^-$ is inconsistent, i.e., $\vdash t : \forall p : *.p$ is derivable for some t.

Corollary

Any PTS with $(*, *) \in \mathcal{A}$ and $(*, *, *) \in \mathcal{R}$ is inconsistent.

Impredicativity

But maybe we could make `Set` impredicative?

Impredicativity

But maybe we could make `Set` impredicative?

- By Girard's paradox it is inconsistent to have two impredicative universes $\mathcal{U}_1 : \mathcal{U}_2$, one inside the other.

Impredicativity

But maybe we could make `Set` impredicative?

- By Girard's paradox it is inconsistent to have two impredicative universes $\mathcal{U}_1 : \mathcal{U}_2$, one inside the other.
- But `Set` and `Prop` are both in $\texttt{Type}_1$, and neither of them is inside the other one.

Impredicativity

But maybe we could make `Set` impredicative?

- By Girard's paradox it is inconsistent to have two impredicative universes $\mathcal{U}_1 : \mathcal{U}_2$, one inside the other.
- But `Set` and `Prop` are both in $\texttt{Type}_1$, and neither of them is inside the other one.
- Indeed, Coq with impredicative `Set` would be consistent.

Impredicativity

But maybe we could make `Set` impredicative?

- By Girard's paradox it is inconsistent to have two impredicative universes $\mathcal{U}_1 : \mathcal{U}_2$, one inside the other.
- But `Set` and `Prop` are both in $\texttt{Type}_1$, and neither of them is inside the other one.
- Indeed, Coq with impredicative `Set` would be consistent.
- But in Coq impredicative `Set` is inconsistent with the combination of classical logic and the axiom of choice!

Subtyping

- Coq's logic is not exactly a PTS (even without inductive types).

Subtyping

- Coq's logic is not exactly a PTS (even without inductive types).
- The conversion rule includes subtyping between universes:
  $\mathcal{U}_1 \leq \mathcal{U}_2$ means that if $t : \mathcal{U}_1$ then $t : \mathcal{U}_2$.

Subtyping

- Coq's logic is not exactly a PTS (even without inductive types).
- The conversion rule includes subtyping between universes: $\mathcal{U}_1 \leq \mathcal{U}_2$ means that if $t : \mathcal{U}_1$ then $t : \mathcal{U}_2$.
- In Coq, $\texttt{Prop} \leq \texttt{Type}_i$ for $i > 0$ and $\texttt{Type}_i \leq \texttt{Type}_j$ for $i \leq j$.

Subtyping

- Coq's logic is not exactly a PTS (even without inductive types).
- The conversion rule includes subtyping between universes:
  $\mathcal{U}_1 \leq \mathcal{U}_2$ means that if $t : \mathcal{U}_1$ then $t : \mathcal{U}_2$.
- In Coq, $\texttt{Prop} \leq \texttt{Type}_i$ for $i > 0$ and $\texttt{Type}_i \leq \texttt{Type}_j$ for $i \leq j$.
- In particular, $\texttt{Prop} : \texttt{Type}_i$ for $i > 0$ and $\texttt{Type}_i : \texttt{Type}_j$ for $i < j$.

Subtyping

- Coq's logic is not exactly a PTS (even without inductive types).
- The conversion rule includes subtyping between universes:
  $\mathcal{U}_1 \leq \mathcal{U}_2$ means that if $t : \mathcal{U}_1$ then $t : \mathcal{U}_2$.
- In Coq, $\texttt{Prop} \leq \texttt{Type}_i$ for $i > 0$ and $\texttt{Type}_i \leq \texttt{Type}_j$ for $i \leq j$.
- In particular, $\texttt{Prop} : \texttt{Type}_i$ for $i > 0$ and $\texttt{Type}_i : \texttt{Type}_j$ for $i < j$.
- One consequence of subtyping: subject reduction for η-reduction fails – η-expansion on legal terms is considered instead.

Proof irrelevance

Proof irrelevance axiom:

$$\forall A : \texttt{Prop}. \forall p_1 p_2 : A. p_1 = p_2$$

Proof irrelevance

Proof irrelevance axiom:

$$\forall A : \texttt{Prop}. \forall p_1 p_2 : A. p_1 = p_2$$

Theorem (Berardi)

- *In Coq, classical logic in impredicative* `Prop` *implies proof irrelevance.*

Proof irrelevance

Proof irrelevance axiom:

$$\forall A : \texttt{Prop}. \forall p_1 p_2 : A. p_1 = p_2$$

Theorem (Berardi)

- *In Coq, classical logic in impredicative* `Prop` *implies proof irrelevance.*
- *In λC, the excluded middle axiom and the axiom of choice together imply proof irrelevance.*

Proof irrelevance

Proof irrelevance axiom:

$$\forall A : \texttt{Prop}.\forall p_1 p_2 : A.p_1 = p_2$$

Theorem (Berardi)

- *In Coq, classical logic in impredicative* `Prop` *implies proof irrelevance.*
- *In λC, the excluded middle axiom and the axiom of choice together imply proof irrelevance.*

Decidability of equality (excluded middle for equality):

$$\forall A : \texttt{Type}.\forall xy : A.x = y \vee x \neq y.$$

Proof irrelevance

Proof irrelevance axiom:

$$\forall A : \texttt{Prop}.\forall p_1 p_2 : A.p_1 = p_2$$

Theorem (Berardi)

- *In Coq, classical logic in impredicative* `Prop` *implies proof irrelevance.*
- *In λC, the excluded middle axiom and the axiom of choice together imply proof irrelevance.*

Decidability of equality (excluded middle for equality):

$$\forall A : \texttt{Type}.\forall xy : A.x = y \vee x \neq y.$$

Theorem

In Coq, proof irrelevance and the axiom of choice together imply decidability of equality.

Axioms

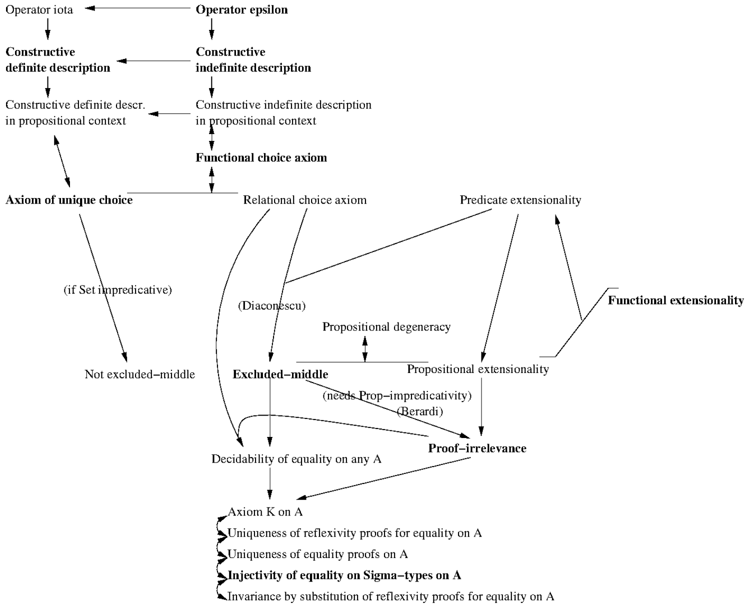

Image source: `https://github.com/coq/coq/wiki/The-Logic-of-Coq`