

# Lecture 3: Higher-order logic

Łukasz Czajka

# Higher-order logic

- In first-order logic there are function and predicate symbols, but no quantification over them.

# Higher-order logic

- In first-order logic there are function and predicate symbols, but no quantification over them.
- $\forall x R(f(x))$  is a first-order formula.

# Higher-order logic

- In first-order logic there are function and predicate symbols, but no quantification over them.
- $\forall x R(f(x))$  is a first-order formula.
- $\exists R \forall f \forall x R(f(x))$  is not.

# Higher-order logic

- In first-order logic there are function and predicate symbols, but no quantification over them.
- $\forall x R(f(x))$  is a first-order formula.
- $\exists R \forall f \forall x R(f(x))$  is not.
- Second-order logic: quantification over first-order predicates is allowed.

# Higher-order logic

- In first-order logic there are function and predicate symbols, but no quantification over them.
- $\forall x R(f(x))$  is a first-order formula.
- $\exists R \forall f \forall x R(f(x))$  is not.
- Second-order logic: quantification over first-order predicates is allowed.
- Second-order predicates: e.g.  $Q(R) := \forall x R x$ .

# Higher-order logic

- In first-order logic there are function and predicate symbols, but no quantification over them.
- $\forall x R(f(x))$  is a first-order formula.
- $\exists R \forall f \forall x R(f(x))$  is not.
- Second-order logic: quantification over first-order predicates is allowed.
- Second-order predicates: e.g.  $Q(R) := \forall x R x$ .
- Third-order logic: quantification over first- and second-order predicates allowed.

# Higher-order logic

- In first-order logic there are function and predicate symbols, but no quantification over them.
- $\forall x R(f(x))$  is a first-order formula.
- $\exists R \forall f \forall x R(f(x))$  is not.
- Second-order logic: quantification over first-order predicates is allowed.
- Second-order predicates: e.g.  $Q(R) := \forall x R x$ .
- Third-order logic: quantification over first- and second-order predicates allowed.
- Higher-order logic: why not go all the way up?



# Higher-order logic: object types

## Definition

The object types (or domains)  $A, B, C$  are given by

$$\mathcal{D} ::= \mathcal{B} \mid \text{Prop} \mid \mathcal{D} \rightarrow \mathcal{D}$$

where  $\mathcal{B}$  is a fixed set of basic domains.

# Higher-order logic: object types

## Definition

The object types (or domains)  $A, B, C$  are given by

$$\mathcal{D} ::= \mathcal{B} \mid \text{Prop} \mid \mathcal{D} \rightarrow \mathcal{D}$$

where  $\mathcal{B}$  is a fixed set of basic domains.

Examples (assuming  $\text{nat}, \text{bool} \in \mathcal{B}$ ):

- first-order predicates:  $\text{nat} \rightarrow \text{Prop}, \text{bool} \rightarrow \text{nat} \rightarrow \text{Prop};$

# Higher-order logic: object types

## Definition

The object types (or domains)  $A, B, C$  are given by

$$\mathcal{D} ::= \mathcal{B} \mid \text{Prop} \mid \mathcal{D} \rightarrow \mathcal{D}$$

where  $\mathcal{B}$  is a fixed set of basic domains.

Examples (assuming  $\text{nat}, \text{bool} \in \mathcal{B}$ ):

- first-order predicates:  $\text{nat} \rightarrow \text{Prop}, \text{bool} \rightarrow \text{nat} \rightarrow \text{Prop};$
- first-order functions:  $\text{bool} \rightarrow \text{bool}, \text{nat} \rightarrow \text{bool} \rightarrow \text{nat};$

# Higher-order logic: object types

## Definition

The object types (or domains)  $A, B, C$  are given by

$$\mathcal{D} ::= \mathcal{B} \mid \text{Prop} \mid \mathcal{D} \rightarrow \mathcal{D}$$

where  $\mathcal{B}$  is a fixed set of basic domains.

Examples (assuming  $\text{nat}, \text{bool} \in \mathcal{B}$ ):

- first-order predicates:  $\text{nat} \rightarrow \text{Prop}, \text{bool} \rightarrow \text{nat} \rightarrow \text{Prop};$
- first-order functions:  $\text{bool} \rightarrow \text{bool}, \text{nat} \rightarrow \text{bool} \rightarrow \text{nat};$
- higher-order predicates:  $(\text{nat} \rightarrow \text{Prop}) \rightarrow \text{Prop};$   
 $(\text{Prop} \rightarrow \text{Prop}) \rightarrow \text{Prop};$

# Higher-order logic: object types

## Definition

The object types (or domains)  $A, B, C$  are given by

$$\mathcal{D} ::= \mathcal{B} \mid \text{Prop} \mid \mathcal{D} \rightarrow \mathcal{D}$$

where  $\mathcal{B}$  is a fixed set of basic domains.

Examples (assuming  $\text{nat}, \text{bool} \in \mathcal{B}$ ):

- first-order predicates:  $\text{nat} \rightarrow \text{Prop}, \text{bool} \rightarrow \text{nat} \rightarrow \text{Prop};$
- first-order functions:  $\text{bool} \rightarrow \text{bool}, \text{nat} \rightarrow \text{bool} \rightarrow \text{nat};$
- higher-order predicates:  $(\text{nat} \rightarrow \text{Prop}) \rightarrow \text{Prop};$   
 $(\text{Prop} \rightarrow \text{Prop}) \rightarrow \text{Prop};$
- higher-order functions:  $(\text{nat} \rightarrow \text{bool}) \rightarrow \text{nat};$   
 $((\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool}) \rightarrow \text{nat};$

# Higher-order logic: object types

## Definition

The object types (or domains)  $A, B, C$  are given by

$$\mathcal{D} ::= \mathcal{B} \mid \text{Prop} \mid \mathcal{D} \rightarrow \mathcal{D}$$

where  $\mathcal{B}$  is a fixed set of basic domains.

Examples (assuming  $\text{nat}, \text{bool} \in \mathcal{B}$ ):

- first-order predicates:  $\text{nat} \rightarrow \text{Prop}, \text{bool} \rightarrow \text{nat} \rightarrow \text{Prop};$
- first-order functions:  $\text{bool} \rightarrow \text{bool}, \text{nat} \rightarrow \text{bool} \rightarrow \text{nat};$
- higher-order predicates:  $(\text{nat} \rightarrow \text{Prop}) \rightarrow \text{Prop};$   
 $(\text{Prop} \rightarrow \text{Prop}) \rightarrow \text{Prop};$
- higher-order functions:  $(\text{nat} \rightarrow \text{bool}) \rightarrow \text{nat};$   
 $((\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool}) \rightarrow \text{nat};$
- functions with predicate arguments:  $(\text{nat} \rightarrow \text{Prop}) \rightarrow \text{nat};$   
 $\text{Prop} \rightarrow \text{bool}.$

## Higher-order logic: object terms

- An object term  $t$  is an object variable  $x, y, z$ , an application  $t_1 t_2$ , an abstraction  $\lambda x : A. t'$ , an implication  $t_1 \Rightarrow t_2$ , or a universal quantification  $\forall x : A. t$ .

## Higher-order logic: object terms

- An object term  $t$  is an object variable  $x, y, z$ , an application  $t_1 t_2$ , an abstraction  $\lambda x : A. t'$ , an implication  $t_1 \Rightarrow t_2$ , or a universal quantification  $\forall x : A. t$ .
- An object context  $\Gamma$  is a finite set of unique declarations of the form  $x : A$ .



## Higher-order logic: object terms

- An object term  $t$  is an object variable  $x, y, z$ , an application  $t_1 t_2$ , an abstraction  $\lambda x : A. t'$ , an implication  $t_1 \Rightarrow t_2$ , or a universal quantification  $\forall x : A. t$ .
- An object context  $\Gamma$  is a finite set of unique declarations of the form  $x : A$ . We write  $\Gamma, x : A$  for  $\Gamma \cup \{x : A\}$ .

## Higher-order logic: object terms

- An object term  $t$  is an object variable  $x, y, z$ , an application  $t_1 t_2$ , an abstraction  $\lambda x : A. t'$ , an implication  $t_1 \Rightarrow t_2$ , or a universal quantification  $\forall x : A. t$ .
- An object context  $\Gamma$  is a finite set of unique declarations of the form  $x : A$ . We write  $\Gamma, x : A$  for  $\Gamma \cup \{x : A\}$ .
- An object term  $t$  has type  $A \in \mathcal{D}$  in  $\Gamma$  if  $\Gamma \vdash t : A$  can be derived using the following rules.

$$\overline{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A. t : A \rightarrow B} \qquad \frac{\Gamma \vdash t_1 : A \rightarrow B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B}$$

$$\frac{\Gamma \vdash \varphi : \mathbf{Prop} \quad \Gamma \vdash \psi : \mathbf{Prop}}{\Gamma \vdash \varphi \Rightarrow \psi : \mathbf{Prop}} \qquad \frac{\Gamma, x : A \vdash \varphi : \mathbf{Prop}}{\Gamma \vdash \forall x : A. \varphi : \mathbf{Prop}}$$

## Higher-order logic: object terms

- An object term  $t$  is an object variable  $x, y, z$ , an application  $t_1 t_2$ , an abstraction  $\lambda x : A. t'$ , an implication  $t_1 \Rightarrow t_2$ , or a universal quantification  $\forall x : A. t$ .
- An object context  $\Gamma$  is a finite set of unique declarations of the form  $x : A$ . We write  $\Gamma, x : A$  for  $\Gamma \cup \{x : A\}$ .
- An object term  $t$  has type  $A \in \mathcal{D}$  in  $\Gamma$  if  $\Gamma \vdash t : A$  can be derived using the following rules.

$$\overline{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A. t : A \rightarrow B} \quad \frac{\Gamma \vdash t_1 : A \rightarrow B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B}$$

$$\frac{\Gamma \vdash \varphi : \mathbf{Prop} \quad \Gamma \vdash \psi : \mathbf{Prop}}{\Gamma \vdash \varphi \Rightarrow \psi : \mathbf{Prop}} \quad \frac{\Gamma, x : A \vdash \varphi : \mathbf{Prop}}{\Gamma \vdash \forall x : A. \varphi : \mathbf{Prop}}$$

We consider only well-typed object terms.

## Higher-order logic: object terms

- An object term  $t$  is an object variable  $x, y, z$ , an application  $t_1 t_2$ , an abstraction  $\lambda x : A. t'$ , an implication  $t_1 \Rightarrow t_2$ , or a universal quantification  $\forall x : A. t$ .
- An object context  $\Gamma$  is a finite set of unique declarations of the form  $x : A$ . We write  $\Gamma, x : A$  for  $\Gamma \cup \{x : A\}$ .
- An object term  $t$  has type  $A \in \mathcal{D}$  in  $\Gamma$  if  $\Gamma \vdash t : A$  can be derived using the following rules.

$$\overline{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A. t : A \rightarrow B} \quad \frac{\Gamma \vdash t_1 : A \rightarrow B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B}$$

$$\frac{\Gamma \vdash \varphi : \mathbf{Prop} \quad \Gamma \vdash \psi : \mathbf{Prop}}{\Gamma \vdash \varphi \Rightarrow \psi : \mathbf{Prop}} \quad \frac{\Gamma, x : A \vdash \varphi : \mathbf{Prop}}{\Gamma \vdash \forall x : A. \varphi : \mathbf{Prop}}$$

We consider only well-typed object terms.

- A formula  $\varphi, \psi$  is an object term of type  $\mathbf{Prop}$ .

# Higher-order logic: object terms

Examples:

- $f : A \rightarrow A \vdash \lambda x : A \lambda y : B. f(fx) : A \rightarrow B \rightarrow A;$

# Higher-order logic: object terms

Examples:

- $f : A \rightarrow A \vdash \lambda x : A \lambda y : B. f(fx) : A \rightarrow B \rightarrow A;$
- $f : A \rightarrow A \vdash \lambda x : A. (\lambda g : A \rightarrow A. g(fx)) f : A \rightarrow A;$

# Higher-order logic: object terms

Examples:

- $f : A \rightarrow A \vdash \lambda x : A \lambda y : B. f(fx) : A \rightarrow B \rightarrow A;$
- $f : A \rightarrow A \vdash \lambda x : A. (\lambda g : A \rightarrow A. g(fx)) f : A \rightarrow A;$
- $f : A \rightarrow A, R : A \rightarrow \mathbf{Prop} \vdash \forall x : A. R(fx) \Rightarrow R(f(fx)) : \mathbf{Prop};$

# Higher-order logic: object terms

Examples:

- $f : A \rightarrow A \vdash \lambda x : A \lambda y : B. f(fx) : A \rightarrow B \rightarrow A;$
- $f : A \rightarrow A \vdash \lambda x : A. (\lambda g : A \rightarrow A. g(fx)) f : A \rightarrow A;$
- $f : A \rightarrow A, R : A \rightarrow \mathbf{Prop} \vdash \forall x : A. R(fx) \Rightarrow R(f(fx)) : \mathbf{Prop};$
- $R : (A \rightarrow B) \rightarrow \mathbf{Prop} \vdash \forall f : A \rightarrow B. Rf \Rightarrow R(\lambda x : A. fx) : \mathbf{Prop};$



# Higher-order logic: object terms

Examples:

- $f : A \rightarrow A \vdash \lambda x : A \lambda y : B. f(fx) : A \rightarrow B \rightarrow A;$
- $f : A \rightarrow A \vdash \lambda x : A. (\lambda g : A \rightarrow A. g(fx)) f : A \rightarrow A;$
- $f : A \rightarrow A, R : A \rightarrow \mathbf{Prop} \vdash \forall x : A. R(fx) \Rightarrow R(f(fx)) : \mathbf{Prop};$
- $R : (A \rightarrow B) \rightarrow \mathbf{Prop} \vdash \forall f : A \rightarrow B. Rf \Rightarrow R(\lambda x : A. fx) : \mathbf{Prop};$
- $x : A, y : A \vdash \forall R : A \rightarrow \mathbf{Prop}. Rx \Rightarrow Ry : \mathbf{Prop}.$

## Higher-order logic: computation

- $\beta$ -reduction “implements” applying a function to an argument:

$$(\lambda x : A. t) t' \rightarrow_{\beta} t[t'/x]$$

## Higher-order logic: computation

- $\beta$ -reduction “implements” applying a function to an argument:

$$(\lambda x : A.t)t' \rightarrow_{\beta} t[t'/x]$$

- Example:  $\lambda x : A.(\lambda y : A.y)x \rightarrow_{\beta} \lambda x : A.x$ .

## Higher-order logic: computation

- $\beta$ -reduction “implements” applying a function to an argument:

$$(\lambda x : A.t)t' \rightarrow_{\beta} t[t'/x]$$

- Example:  $\lambda x : A.(\lambda y : A.y)x \rightarrow_{\beta} \lambda x : A.x$ .
- The relation  $=_{\beta}$  of  $\beta$ -equality is the least equivalence relation including  $\beta$ -reduction.

## Higher-order logic: computation

- $\beta$ -reduction “implements” applying a function to an argument:

$$(\lambda x : A.t)t' \rightarrow_{\beta} t[t'/x]$$

- Example:  $\lambda x : A.(\lambda y : A.y)x \rightarrow_{\beta} \lambda x : A.x$ .
- The relation  $=_{\beta}$  of  $\beta$ -equality is the least equivalence relation including  $\beta$ -reduction.
- $\eta$ -reduction “implements” syntactic extensionality of functions:

$$(\lambda x : A.tx) \rightarrow_{\eta} t \quad \text{if } x \notin \text{FV}(t)$$

## Higher-order logic: computation

- $\beta$ -reduction “implements” applying a function to an argument:

$$(\lambda x : A.t)t' \rightarrow_{\beta} t[t'/x]$$

- Example:  $\lambda x : A.(\lambda y : A.y)x \rightarrow_{\beta} \lambda x : A.x$ .
- The relation  $=_{\beta}$  of  $\beta$ -equality is the least equivalence relation including  $\beta$ -reduction.
- $\eta$ -reduction “implements” syntactic extensionality of functions:

$$(\lambda x : A.tx) \rightarrow_{\eta} t \quad \text{if } x \notin \text{FV}(t)$$

- The relation  $=_{\beta\eta}$  of  $\beta\eta$ -equality is the least equivalence relation including  $\beta$ - and  $\eta$ -reduction.

## Higher-order logic: computation

- $\beta$ -reduction “implements” applying a function to an argument:

$$(\lambda x : A.t)t' \rightarrow_{\beta} t[t'/x]$$

- Example:  $\lambda x : A.(\lambda y : A.y)x \rightarrow_{\beta} \lambda x : A.x$ .
- The relation  $=_{\beta}$  of  $\beta$ -equality is the least equivalence relation including  $\beta$ -reduction.
- $\eta$ -reduction “implements” syntactic extensionality of functions:

$$(\lambda x : A.tx) \rightarrow_{\eta} t \quad \text{if } x \notin \text{FV}(t)$$

- The relation  $=_{\beta\eta}$  of  $\beta\eta$ -equality is the least equivalence relation including  $\beta$ - and  $\eta$ -reduction.
- The relation  $\equiv$  of definitional equality (also called computational equality) is defined to be  $\beta\eta$ -equality.

## Higher-order logic: computation

- $\beta$ -reduction “implements” applying a function to an argument:

$$(\lambda x : A.t)t' \rightarrow_{\beta} t[t'/x]$$

- Example:  $\lambda x : A.(\lambda y : A.y)x \rightarrow_{\beta} \lambda x : A.x$ .
- The relation  $=_{\beta}$  of  $\beta$ -equality is the least equivalence relation including  $\beta$ -reduction.
- $\eta$ -reduction “implements” syntactic extensionality of functions:

$$(\lambda x : A.tx) \rightarrow_{\eta} t \quad \text{if } x \notin \text{FV}(t)$$

- The relation  $=_{\beta\eta}$  of  $\beta\eta$ -equality is the least equivalence relation including  $\beta$ - and  $\eta$ -reduction.
- The relation  $\equiv$  of definitional equality (also called computational equality) is defined to be  $\beta\eta$ -equality.
  - Definitional equality is different for different systems.



## Higher-order logic: computation

- $\beta$ -reduction “implements” applying a function to an argument:

$$(\lambda x : A.t)t' \rightarrow_{\beta} t[t'/x]$$

- Example:  $\lambda x : A.(\lambda y : A.y)x \rightarrow_{\beta} \lambda x : A.x$ .
- The relation  $=_{\beta}$  of  $\beta$ -equality is the least equivalence relation including  $\beta$ -reduction.
- $\eta$ -reduction “implements” syntactic extensionality of functions:

$$(\lambda x : A.tx) \rightarrow_{\eta} t \quad \text{if } x \notin \text{FV}(t)$$

- The relation  $=_{\beta\eta}$  of  $\beta\eta$ -equality is the least equivalence relation including  $\beta$ - and  $\eta$ -reduction.
- The relation  $\equiv$  of definitional equality (also called computational equality) is defined to be  $\beta\eta$ -equality.
  - Definitional equality is different for different systems.
  - Definitional equality is an equivalence relation compatible with the structure of terms.

## Higher-order logic: computation

- $\beta$ -reduction “implements” applying a function to an argument:

$$(\lambda x : A.t)t' \rightarrow_{\beta} t[t'/x]$$

- Example:  $\lambda x : A.(\lambda y : A.y)x \rightarrow_{\beta} \lambda x : A.x$ .
- The relation  $=_{\beta}$  of  $\beta$ -equality is the least equivalence relation including  $\beta$ -reduction.
- $\eta$ -reduction “implements” syntactic extensionality of functions:

$$(\lambda x : A.tx) \rightarrow_{\eta} t \quad \text{if } x \notin \text{FV}(t)$$

- The relation  $=_{\beta\eta}$  of  $\beta\eta$ -equality is the least equivalence relation including  $\beta$ - and  $\eta$ -reduction.
- The relation  $\equiv$  of definitional equality (also called computational equality) is defined to be  $\beta\eta$ -equality.
  - Definitional equality is different for different systems.
  - Definitional equality is an equivalence relation compatible with the structure of terms.
    - E.g. if  $t \equiv t'$  then  $\lambda x : A.f t x \equiv \lambda x : A.f t' x$ .

## Higher-order logic: computation

- $\beta$ -reduction “implements” applying a function to an argument:

$$(\lambda x : A.t)t' \rightarrow_{\beta} t[t'/x]$$

- Example:  $\lambda x : A.(\lambda y : A.y)x \rightarrow_{\beta} \lambda x : A.x$ .
- The relation  $=_{\beta}$  of  $\beta$ -equality is the least equivalence relation including  $\beta$ -reduction.
- $\eta$ -reduction “implements” syntactic extensionality of functions:

$$(\lambda x : A.tx) \rightarrow_{\eta} t \quad \text{if } x \notin \text{FV}(t)$$

- The relation  $=_{\beta\eta}$  of  $\beta\eta$ -equality is the least equivalence relation including  $\beta$ - and  $\eta$ -reduction.
- The relation  $\equiv$  of definitional equality (also called computational equality) is defined to be  $\beta\eta$ -equality.
  - Definitional equality is different for different systems.
  - Definitional equality is an equivalence relation compatible with the structure of terms.
    - E.g. if  $t \equiv t'$  then  $\lambda x : A.f t x \equiv \lambda x : A.f t' x$ .
  - Definitional equality is decidable.

# Syntactic functional extensionality and $\eta$ -reduction

## Definition

Syntactic functional extensionality for  $\Gamma, A, B$  is the following (meta) statement:

- for any  $f, g$  with  $\Gamma \vdash f : A \rightarrow B$  and  $\Gamma \vdash g : A \rightarrow B$ , if  $ft \equiv gt$  for every  $t$  with  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ , then  $f \equiv g$ .

# Syntactic functional extensionality and $\eta$ -reduction

## Definition

Syntactic functional extensionality for  $\Gamma, A, B$  is the following (meta) statement:

- for any  $f, g$  with  $\Gamma \vdash f : A \rightarrow B$  and  $\Gamma \vdash g : A \rightarrow B$ , if  $ft \equiv gt$  for every  $t$  with  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ , then  $f \equiv g$ .

## Fact

*If definitional equality includes  $\eta$ -reduction then syntactic functional extensionality holds.*

# Syntactic functional extensionality and $\eta$ -reduction

## Definition

Syntactic functional extensionality for  $\Gamma, A, B$  is the following (meta) statement:

- for any  $f, g$  with  $\Gamma \vdash f : A \rightarrow B$  and  $\Gamma \vdash g : A \rightarrow B$ , if  $ft \equiv gt$  for every  $t$  with  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ , then  $f \equiv g$ .

## Fact

*If definitional equality includes  $\eta$ -reduction then syntactic functional extensionality holds.*

## Proof.

Let  $f, g : A \rightarrow B$  in  $\Gamma$ . Assume  $ft \equiv gt$  for all  $t$  such that  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ .

# Syntactic functional extensionality and $\eta$ -reduction

## Definition

Syntactic functional extensionality for  $\Gamma, A, B$  is the following (meta) statement:

- for any  $f, g$  with  $\Gamma \vdash f : A \rightarrow B$  and  $\Gamma \vdash g : A \rightarrow B$ , if  $ft \equiv gt$  for every  $t$  with  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ , then  $f \equiv g$ .

## Fact

*If definitional equality includes  $\eta$ -reduction then syntactic functional extensionality holds.*

## Proof.

Let  $f, g : A \rightarrow B$  in  $\Gamma$ . Assume  $ft \equiv gt$  for all  $t$  such that  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ . Take a fresh variable  $x \notin \text{FV}(f, g, \Gamma)$  and let  $\Gamma' = \Gamma, x : A$ .

# Syntactic functional extensionality and $\eta$ -reduction

## Definition

Syntactic functional extensionality for  $\Gamma, A, B$  is the following (meta) statement:

- for any  $f, g$  with  $\Gamma \vdash f : A \rightarrow B$  and  $\Gamma \vdash g : A \rightarrow B$ , if  $ft \equiv gt$  for every  $t$  with  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ , then  $f \equiv g$ .

## Fact

*If definitional equality includes  $\eta$ -reduction then syntactic functional extensionality holds.*

## Proof.

Let  $f, g : A \rightarrow B$  in  $\Gamma$ . Assume  $ft \equiv gt$  for all  $t$  such that  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ . Take a fresh variable  $x \notin \text{FV}(f, g, \Gamma)$  and let  $\Gamma' = \Gamma, x : A$ . Then  $\Gamma' \vdash x : A$ , so  $fx \equiv gx$ .



# Syntactic functional extensionality and $\eta$ -reduction

## Definition

Syntactic functional extensionality for  $\Gamma, A, B$  is the following (meta) statement:

- for any  $f, g$  with  $\Gamma \vdash f : A \rightarrow B$  and  $\Gamma \vdash g : A \rightarrow B$ , if  $ft \equiv gt$  for every  $t$  with  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ , then  $f \equiv g$ .

## Fact

*If definitional equality includes  $\eta$ -reduction then syntactic functional extensionality holds.*

## Proof.

Let  $f, g : A \rightarrow B$  in  $\Gamma$ . Assume  $ft \equiv gt$  for all  $t$  such that  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ . Take a fresh variable  $x \notin \text{FV}(f, g, \Gamma)$  and let  $\Gamma' = \Gamma, x : A$ . Then  $\Gamma' \vdash x : A$ , so  $fx \equiv gx$ . Hence also  $\lambda x : A. fx \equiv \lambda x : A. gx$ .

# Syntactic functional extensionality and $\eta$ -reduction

## Definition

Syntactic functional extensionality for  $\Gamma, A, B$  is the following (meta) statement:

- for any  $f, g$  with  $\Gamma \vdash f : A \rightarrow B$  and  $\Gamma \vdash g : A \rightarrow B$ , if  $ft \equiv gt$  for every  $t$  with  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ , then  $f \equiv g$ .

## Fact

*If definitional equality includes  $\eta$ -reduction then syntactic functional extensionality holds.*

## Proof.

Let  $f, g : A \rightarrow B$  in  $\Gamma$ . Assume  $ft \equiv gt$  for all  $t$  such that  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ . Take a fresh variable  $x \notin \text{FV}(f, g, \Gamma)$  and let  $\Gamma' = \Gamma, x : A$ . Then  $\Gamma' \vdash x : A$ , so  $fx \equiv gx$ . Hence also  $\lambda x : A. fx \equiv \lambda x : A. gx$ . But  $\lambda x : A. fx \rightarrow_{\eta} f$  and  $\lambda x : A. gx \rightarrow_{\eta} g$  (recall  $x \notin \text{FV}(f, g)$ ).

# Syntactic functional extensionality and $\eta$ -reduction

## Definition

Syntactic functional extensionality for  $\Gamma, A, B$  is the following (meta) statement:

- for any  $f, g$  with  $\Gamma \vdash f : A \rightarrow B$  and  $\Gamma \vdash g : A \rightarrow B$ , if  $ft \equiv gt$  for every  $t$  with  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ , then  $f \equiv g$ .

## Fact

*If definitional equality includes  $\eta$ -reduction then syntactic functional extensionality holds.*

## Proof.

Let  $f, g : A \rightarrow B$  in  $\Gamma$ . Assume  $ft \equiv gt$  for all  $t$  such that  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ . Take a fresh variable  $x \notin \text{FV}(f, g, \Gamma)$  and let  $\Gamma' = \Gamma, x : A$ . Then  $\Gamma' \vdash x : A$ , so  $fx \equiv gx$ . Hence also  $\lambda x : A. fx \equiv \lambda x : A. gx$ . But  $\lambda x : A. fx \rightarrow_{\eta} f$  and  $\lambda x : A. gx \rightarrow_{\eta} g$  (recall  $x \notin \text{FV}(f, g)$ ). Then  $\lambda x : A. fx \equiv f$  and  $\lambda x : A. gx \equiv g$  because  $\equiv$  includes  $\eta$ -reduction.

# Syntactic functional extensionality and $\eta$ -reduction

## Definition

Syntactic functional extensionality for  $\Gamma, A, B$  is the following (meta) statement:

- for any  $f, g$  with  $\Gamma \vdash f : A \rightarrow B$  and  $\Gamma \vdash g : A \rightarrow B$ , if  $ft \equiv gt$  for every  $t$  with  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ , then  $f \equiv g$ .

## Fact

*If definitional equality includes  $\eta$ -reduction then syntactic functional extensionality holds.*

## Proof.

Let  $f, g : A \rightarrow B$  in  $\Gamma$ . Assume  $ft \equiv gt$  for all  $t$  such that  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ . Take a fresh variable  $x \notin \text{FV}(f, g, \Gamma)$  and let  $\Gamma' = \Gamma, x : A$ . Then  $\Gamma' \vdash x : A$ , so  $fx \equiv gx$ . Hence also  $\lambda x : A. fx \equiv \lambda x : A. gx$ . But  $\lambda x : A. fx \rightarrow_{\eta} f$  and  $\lambda x : A. gx \rightarrow_{\eta} g$  (recall  $x \notin \text{FV}(f, g)$ ). Then  $\lambda x : A. fx \equiv f$  and  $\lambda x : A. gx \equiv g$  because  $\equiv$  includes  $\eta$ -reduction. This implies  $f \equiv g$ . □

# Syntactic functional extensionality and $\eta$ -reduction

## Definition

Syntactic functional extensionality for  $\Gamma, A, B$  is the following (meta) statement:

- for any  $f, g$  with  $\Gamma \vdash f : A \rightarrow B$  and  $\Gamma \vdash g : A \rightarrow B$ , if  $ft \equiv gt$  for every  $t$  with  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ , then  $f \equiv g$ .

## Fact

*If definitional equality includes  $\eta$ -reduction then syntactic functional extensionality holds.*

## Proof.

Let  $f, g : A \rightarrow B$  in  $\Gamma$ . Assume  $ft \equiv gt$  for all  $t$  such that  $\Gamma' \vdash t : A$  for some  $\Gamma' \supseteq \Gamma$ . Take a fresh variable  $x \notin \text{FV}(f, g, \Gamma)$  and let  $\Gamma' = \Gamma, x : A$ . Then  $\Gamma' \vdash x : A$ , so  $fx \equiv gx$ . Hence also  $\lambda x : A. fx \equiv \lambda x : A. gx$ . But  $\lambda x : A. fx \rightarrow_{\eta} f$  and  $\lambda x : A. gx \rightarrow_{\eta} g$  (recall  $x \notin \text{FV}(f, g)$ ). Then  $\lambda x : A. fx \equiv f$  and  $\lambda x : A. gx \equiv g$  because  $\equiv$  includes  $\eta$ -reduction. This implies  $f \equiv g$ . □

Trivially, if syntactic functional extensionality holds and definitional equality includes  $\beta$ -reduction, then it also includes  $\eta$ -reduction (exercise).

## Intermission: the simply-typed lambda-calculus

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

# Intermission: the simply-typed lambda-calculus

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

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

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

# Intermission: the simply-typed lambda-calculus

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

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

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

- $\beta$ -reduction:  $(\lambda x : \tau. t)t' \rightarrow_{\beta} t[t'/x]$ .



# Intermission: the simply-typed lambda-calculus

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

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

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

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

# Intermission: the simply-typed lambda-calculus

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

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

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

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

# Intermission: the simply-typed lambda-calculus

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

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

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

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

# Intermission: the simply-typed lambda-calculus

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

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

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

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

## Higher-order logic: proof terms

- A proof term  $M, N$  is a proof variable  $X, Y, Z$ , a lambda abstraction  $\lambda X : \varphi.M$  or  $\lambda x : A.M$ , or an application  $M_1 M_2$  or  $Mt$ .

# Higher-order logic: proof terms

- A proof term  $M, N$  is a proof variable  $X, Y, Z$ , a lambda abstraction  $\lambda X : \varphi. M$  or  $\lambda x : A. M$ , or an application  $M_1 M_2$  or  $M t$ .
- A proof context  $\Delta$  is a finite set of unique declarations of the form  $X : \varphi$ .

# Higher-order logic: proof terms

- A proof term  $M, N$  is a proof variable  $X, Y, Z$ , a lambda abstraction  $\lambda X : \varphi. M$  or  $\lambda x : A. M$ , or an application  $M_1 M_2$  or  $M t$ .
- A proof context  $\Delta$  is a finite set of unique declarations of the form  $X : \varphi$ .
- A judgement has the form  $\Gamma; \Delta \vdash M : \varphi$ .

## Intermission: derivation rules

$$\frac{J_1 \quad \dots \quad J_n}{J} S$$

- If we have derived the judgements  $J_1, \dots, J_n$  and the side condition  $S$  holds, then we can derive the judgement  $J$ .



## Intermission: derivation rules

$$\frac{J_1 \quad \dots \quad J_n}{J} S$$

- If we have derived the judgements  $J_1, \dots, J_n$  and the side condition  $S$  holds, then we can derive the judgement  $J$ .
- Sometimes we write the side condition(s) above the line together with the judgements  $J_1, \dots, J_n$ .

## Intermission: derivation trees

$$\frac{\frac{\overline{J_3} \quad \frac{\overline{J_5}}{\overline{J_4}}}{\overline{J_1}} \quad \overline{J_2}}{\overline{J}}$$

- To derive a judgement  $J$  we build a derivation tree using the derivation rules: each node is a valid application of a derivation rule.

## Intermission: derivation trees

$$\frac{\frac{\overline{J_3} \quad \frac{\overline{J_5}}{\overline{J_4}}}{\overline{J_1}} \quad \overline{J_2}}{\overline{J}}$$

- To derive a judgement  $J$  we build a derivation tree using the derivation rules: each node is a valid application of a derivation rule.
- At the leaves of the tree we need rules with no judgements above the line.

# Intuitionistic higher-order logic: rules

$$\overline{\Gamma; \Delta, X : \varphi \vdash X : \varphi} \text{ (Ax)}$$

$$\frac{\Gamma; \Delta, X : \varphi \vdash M : \psi}{\Gamma; \Delta \vdash \lambda X : \varphi. M : \varphi \Rightarrow \psi} \text{ (}\Rightarrow\text{I)} \quad \frac{\Gamma; \Delta \vdash M : \varphi \Rightarrow \psi \quad \Gamma; \Delta \vdash N : \varphi}{\Gamma; \Delta \vdash MN : \psi} \text{ (}\Rightarrow\text{E)}$$

$$\frac{\Gamma, x : A; \Delta \vdash M : \varphi \quad x \notin \text{FV}(\Delta)}{\Gamma; \Delta \vdash \lambda x : A. M : \forall x : A. \varphi} \text{ (}\forall\text{I)} \quad \frac{\Gamma; \Delta \vdash M : \forall x : A. \varphi \quad \Gamma \vdash t : A}{\Gamma; \Delta \vdash Mt : \varphi[t/x]} \text{ (}\forall\text{E)}$$

$$\frac{\Gamma; \Delta \vdash M : \varphi \quad \Gamma \vdash \psi : \mathbf{Prop} \quad \varphi \equiv \psi}{\Gamma; \Delta \vdash M : \psi} \text{ (conv)}$$

# Intuitionistic higher-order logic: example derivation

$$\frac{\frac{\overline{\Gamma; \Delta \vdash X_1 : \forall x : A. Px \Rightarrow Q} \quad \overline{\Gamma \vdash x : A}}{\Gamma; \Delta \vdash X_1 x : Px \Rightarrow Q} \quad \frac{\overline{\Gamma; \Delta \vdash X_2 : \forall x : A. Px} \quad \overline{\Gamma \vdash x : A}}{\Gamma; \Delta \vdash X_2 x : Px}}{\Gamma; \Delta \vdash X_1 x (X_2 x) : Q}$$

- $\Gamma = P : A \rightarrow \mathbf{Prop}, \quad Q : \mathbf{Prop}, \quad x : A.$
- $\Delta = X_1 : \forall x : A. Px \Rightarrow Q, \quad X_2 : \forall x : A. Px.$

## Higher-order logic: expressiveness

- A second-order predicate expressing the transitivity of a binary relation:

$$\text{Trans} := \lambda R : A \rightarrow A \rightarrow \text{Prop}. \forall xyz : A. Rxy \Rightarrow Ryz \Rightarrow Rxz$$

## Higher-order logic: expressiveness

- A second-order predicate expressing the transitivity of a binary relation:

$$\text{Trans} := \lambda R : A \rightarrow A \rightarrow \text{Prop}. \forall xyz : A. Rxy \Rightarrow Ryz \Rightarrow Rxz$$

- A binary relation  $R$  is included in  $S$  if for all  $x, y$ ,  $Rxy$  implies  $Sxy$ :

$$\text{Subrel} := \lambda RS : A \rightarrow A \rightarrow \text{Prop}. \forall xy : A. Rxy \Rightarrow Sxy$$

## Higher-order logic: expressiveness

- A second-order predicate expressing the transitivity of a binary relation:

$$\text{Trans} := \lambda R : A \rightarrow A \rightarrow \text{Prop}. \forall xyz : A. Rxy \Rightarrow Ryz \Rightarrow Rxz$$

- A binary relation  $R$  is included in  $S$  if for all  $x, y$ ,  $Rxy$  implies  $Sxy$ :

$$\text{Subrel} := \lambda RS : A \rightarrow A \rightarrow \text{Prop}. \forall xy : A. Rxy \Rightarrow Sxy$$

- The transitive closure of a binary relation  $R$  is the least transitive relation including  $R$ .



## Higher-order logic: expressiveness

- A second-order predicate expressing the transitivity of a binary relation:

$$\text{Trans} := \lambda R : A \rightarrow A \rightarrow \text{Prop}. \forall xyz : A. Rxy \Rightarrow Ryz \Rightarrow Rxz$$

- A binary relation  $R$  is included in  $S$  if for all  $x, y$ ,  $Rxy$  implies  $Sxy$ :

$$\text{Subrel} := \lambda R S : A \rightarrow A \rightarrow \text{Prop}. \forall xy : A. Rxy \Rightarrow Sxy$$

- The transitive closure of a binary relation  $R$  is the least transitive relation including  $R$ . This can be defined as the intersection of all transitive relations including  $R$ :

$$\begin{aligned} \text{TC} := \lambda R : A \rightarrow A \rightarrow \text{Prop}. & \lambda xy : A. \forall S : A \rightarrow A \rightarrow \text{Prop}. \\ & \text{Trans}(S) \Rightarrow \text{Subrel } R S \Rightarrow Sxy \end{aligned}$$

## Higher-order logic: expressiveness

$$\text{TC} := \lambda R : A \rightarrow A \rightarrow \text{Prop}. \lambda xy : A. \forall S : A \rightarrow A \rightarrow \text{Prop}. \\ \text{Trans}(S) \Rightarrow \text{Subrel } R S \Rightarrow Sxy$$

Exercise: for arbitrary  $R : A \rightarrow A \rightarrow \text{Prop}$  prove that  $\text{TC}(R)$  is indeed the least transitive relation including  $R$ , i.e.,

- it is transitive:

$$\text{Trans}(\text{TC}(R))$$

- it includes  $R$ :

$$\text{Subrel } R (\text{TC}(R))$$

- every other transitive relation which includes  $R$  also includes  $\text{TC}(R)$ :

$$\forall S : A \rightarrow A \rightarrow \text{Prop}. \text{Trans}(S) \Rightarrow \text{Subrel } R S \Rightarrow \text{Subrel } (\text{TC}(R)) S$$

## Higher-order logic: expressiveness

Induction principle for natural numbers:

$$\forall P : \mathbf{nat} \rightarrow \mathbf{Prop}. P0 \Rightarrow (\forall n : \mathbf{nat}. Pn \Rightarrow P(Sn)) \Rightarrow \forall n : \mathbf{nat}. Pn$$

## Higher-order encodings of logical connectives

In higher-order logic all other connectives and equality may be defined using  $\forall$  and  $\Rightarrow$ .

## Higher-order encodings of logical connectives

In higher-order logic all other connectives and equality may be defined using  $\forall$  and  $\Rightarrow$ .

- Truth:  $\top := \forall P : \mathbf{Prop}. P \Rightarrow P$ .

# Higher-order encodings of logical connectives

In higher-order logic all other connectives and equality may be defined using  $\forall$  and  $\Rightarrow$ .

- Truth:  $\top := \forall P : \mathbf{Prop}. P \Rightarrow P$ .
- Falsity:  $\perp := \forall P : \mathbf{Prop}. P$ .

# Higher-order encodings of logical connectives

In higher-order logic all other connectives and equality may be defined using  $\forall$  and  $\Rightarrow$ .

- Truth:  $\top := \forall P : \mathbf{Prop}. P \Rightarrow P$ .
- Falsity:  $\perp := \forall P : \mathbf{Prop}. P$ .
- Conjunction:  $\varphi \wedge \psi := \forall P : \mathbf{Prop}. (\varphi \Rightarrow \psi \Rightarrow P) \Rightarrow P$ .

# Higher-order encodings of logical connectives

In higher-order logic all other connectives and equality may be defined using  $\forall$  and  $\Rightarrow$ .

- Truth:  $\top := \forall P : \mathbf{Prop}. P \Rightarrow P$ .
- Falsity:  $\perp := \forall P : \mathbf{Prop}. P$ .
- Conjunction:  $\varphi \wedge \psi := \forall P : \mathbf{Prop}. (\varphi \Rightarrow \psi \Rightarrow P) \Rightarrow P$ .
- Disjunction:  $\varphi \vee \psi := \forall P : \mathbf{Prop}. (\varphi \Rightarrow P) \Rightarrow (\psi \Rightarrow P) \Rightarrow P$ .



# Higher-order encodings of logical connectives

In higher-order logic all other connectives and equality may be defined using  $\forall$  and  $\Rightarrow$ .

- Truth:  $\top := \forall P : \mathbf{Prop}. P \Rightarrow P$ .
- Falsity:  $\perp := \forall P : \mathbf{Prop}. P$ .
- Conjunction:  $\varphi \wedge \psi := \forall P : \mathbf{Prop}. (\varphi \Rightarrow \psi \Rightarrow P) \Rightarrow P$ .
- Disjunction:  $\varphi \vee \psi := \forall P : \mathbf{Prop}. (\varphi \Rightarrow P) \Rightarrow (\psi \Rightarrow P) \Rightarrow P$ .
- Existential quantification:  
 $\exists x : A. \varphi(x) := \forall P : \mathbf{Prop}. \forall x : A. (\varphi(x) \Rightarrow P) \Rightarrow P$ .

# Higher-order encodings of logical connectives

In higher-order logic all other connectives and equality may be defined using  $\forall$  and  $\Rightarrow$ .

- Truth:  $\top := \forall P : \mathbf{Prop}. P \Rightarrow P$ .
- Falsity:  $\perp := \forall P : \mathbf{Prop}. P$ .
- Conjunction:  $\varphi \wedge \psi := \forall P : \mathbf{Prop}. (\varphi \Rightarrow \psi \Rightarrow P) \Rightarrow P$ .
- Disjunction:  $\varphi \vee \psi := \forall P : \mathbf{Prop}. (\varphi \Rightarrow P) \Rightarrow (\psi \Rightarrow P) \Rightarrow P$ .
- Existential quantification:  
 $\exists x : A. \varphi(x) := \forall P : \mathbf{Prop}. \forall x : A. (\varphi(x) \Rightarrow P) \Rightarrow P$ .
- Leibniz equality on  $A$ :  $x =_A y := \forall R : A \rightarrow \mathbf{Prop}. Rx \Rightarrow Ry$ .

# Higher-order encodings of logical connectives

In higher-order logic all other connectives and equality may be defined using  $\forall$  and  $\Rightarrow$ .

- Truth:  $\top := \forall P : \mathbf{Prop}. P \Rightarrow P$ .
- Falsity:  $\perp := \forall P : \mathbf{Prop}. P$ .
- Conjunction:  $\varphi \wedge \psi := \forall P : \mathbf{Prop}. (\varphi \Rightarrow \psi \Rightarrow P) \Rightarrow P$ .
- Disjunction:  $\varphi \vee \psi := \forall P : \mathbf{Prop}. (\varphi \Rightarrow P) \Rightarrow (\psi \Rightarrow P) \Rightarrow P$ .
- Existential quantification:  
 $\exists x : A. \varphi(x) := \forall P : \mathbf{Prop}. \forall x : A. (\varphi(x) \Rightarrow P) \Rightarrow P$ .
- Leibniz equality on  $A$ :  $x =_A y := \forall R : A \rightarrow \mathbf{Prop}. Rx \Rightarrow Ry$ .

The corresponding introduction and elimination rules are derivable.

# Classical higher-order logic

Excluded middle axiom:

$$\forall P : \mathbf{Prop}. P \vee \neg P$$

# Extensionality

- Functional extensionality axiom (scheme):

$$\forall f g : A \rightarrow B. (\forall x : A. f x = g x) \Rightarrow f = g.$$

# Extensionality

- Functional extensionality axiom (scheme):

$$\forall f g : A \rightarrow B. (\forall x : A. fx = gx) \Rightarrow f = g.$$

## NOTE:

- Syntactic functional extensionality does not imply functional extensionality!

# Extensionality

- Functional extensionality axiom (scheme):

$$\forall f g : A \rightarrow B. (\forall x : A. f x = g x) \Rightarrow f = g.$$

## NOTE:

- Syntactic functional extensionality does not imply functional extensionality!
  - More precisely: that a formal system of logic satisfies syntactic functional extensionality (a meta-theoretic property!) does not imply that the functional extensionality axiom is provable in the system.

# Extensionality

- Functional extensionality axiom (scheme):

$$\forall f g : A \rightarrow B. (\forall x : A. fx = gx) \Rightarrow f = g.$$

## NOTE:

- Syntactic functional extensionality does not imply functional extensionality!
  - More precisely: that a formal system of logic satisfies syntactic functional extensionality (a meta-theoretic property!) does not imply that the functional extensionality axiom is provable in the system.
- Functional extensionality does not imply syntactic functional extensionality either!



# Extensionality

- Functional extensionality axiom (scheme):

$$\forall fg : A \rightarrow B. (\forall x : A. fx = gx) \Rightarrow f = g.$$

## NOTE:

- Syntactic functional extensionality does not imply functional extensionality!
  - More precisely: that a formal system of logic satisfies syntactic functional extensionality (a meta-theoretic property!) does not imply that the functional extensionality axiom is provable in the system.
- Functional extensionality does not imply syntactic functional extensionality either!
- Propositional extensionality axiom:

$$\forall P_1 P_2 : \text{Prop}. (P_1 \Leftrightarrow P_2) \Rightarrow P_1 =_{\text{Prop}} P_2.$$

# Extensionality

- Functional extensionality axiom (scheme):

$$\forall fg : A \rightarrow B. (\forall x : A. fx = gx) \Rightarrow f = g.$$

## NOTE:

- Syntactic functional extensionality does not imply functional extensionality!
  - More precisely: that a formal system of logic satisfies syntactic functional extensionality (a meta-theoretic property!) does not imply that the functional extensionality axiom is provable in the system.
- Functional extensionality does not imply syntactic functional extensionality either!
- Propositional extensionality axiom:

$$\forall P_1 P_2 : \mathbf{Prop}. (P_1 \Leftrightarrow P_2) \Rightarrow P_1 =_{\mathbf{Prop}} P_2.$$

- Predicate extensionality axiom (scheme):

$$\forall R_1 R_2 : A \rightarrow \mathbf{Prop}. (\forall x : A. R_1 x \Leftrightarrow R_2 x) \Rightarrow R_1 = R_2.$$

# Choice

Axiom of choice (scheme):

$$(\forall x : A. \exists y : B. Rxy) \Rightarrow \exists f : A \rightarrow B. \forall x : A. Rx(fx).$$

# Church's Simple Type Theory

- Church's Simple Type Theory is essentially classical higher-order logic with extensionality and choice.

# Church's Simple Type Theory

- Church's Simple Type Theory is essentially classical higher-order logic with extensionality and choice.
- Alonzo Church, "A formulation of the simple theory of types", JSL 1940.

# Church's Simple Type Theory

- Church's Simple Type Theory is essentially classical higher-order logic with extensionality and choice.
- Alonzo Church, "A formulation of the simple theory of types", JSL 1940.
  - The simply-typed lambda-calculus originates from this paper, where it was used to define the object terms of Church's higher-order logic.

## Relativised choice

Relativised axiom of choice:

$$(\forall x : A. Qx \Rightarrow \exists y : B. Rxy) \Rightarrow \exists f : A \rightarrow B. \forall x : A. Qx \Rightarrow Rx(fx).$$

# Diaconescu's theorem

## Theorem (Diaconescu)

*In intuitionistic higher-order logic, the predicate extensionality axiom and the relativised axiom of choice together imply the excluded middle axiom.*