An Introduction to Dependent Type Theory

Leon Vatthauer

24.07.2025

Seminar Wissensrepräsentation und -verarbeitung

Roadmap

Roadmap

In this talk we will:

- Extend $\lambda\!\rightarrow$ to first-order logic (resulting in $\lambda\mathrm{P}$)

Roadmap

In this talk we will:

- Extend $\lambda\!\to$ to first-order logic (resulting in $\lambda \mathrm{P}$)
- Discuss properties of $\lambda \mathrm{P}$

Roadmap

In this talk we will:

- Extend $\lambda\!\rightarrow$ to first-order logic (resulting in $\lambda\mathrm{P}$)
- Discuss properties of $\lambda\mathrm{P}$
- Look at applications of dependent type theory

Recall: Propositional Logic to FOL

Definition

The syntax of propositional logic is defined by:

$$\varphi, \psi ::= \bot \mid P \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \to \psi \qquad P \in \mathcal{V}$$

where $\mathcal{V}$ is a set of propositional variables.

Recall: Propositional Logic to FOL

Definition

The syntax of propositional logic is defined by:

$$\varphi, \psi ::= \bot \mid P \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \to \psi \qquad P \in \mathcal{V}$$

where $\mathcal{V}$ is a set of propositional variables.

Definition

First-order logic extends propositional logic with predicates and quantifiers:

$$\varphi, \psi ::= \bot \mid P(t_1, \ldots, t_n) \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \to \psi \mid \forall x.\varphi \mid \exists x.\varphi$$

where $t_1, \ldots, t_n$ are terms and P is a n-ary predicate.

Recall: Propositional Logic to FOL

Definition

The syntax of propositional logic is defined by:

$$\varphi, \psi ::= \bot \mid P \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \qquad P \in \mathcal{V}$$

where $\mathcal{V}$ is a set of propositional variables.

Definition

First-order logic extends propositional logic with predicates and quantifiers:

$$\varphi, \psi ::= \bot \mid P(t_1, \ldots, t_n) \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \forall x.\varphi \mid \exists x.\varphi$$

where $t_1, \ldots, t_n$ are terms and P is a n-ary predicate.

Extending $\lambda{\rightarrow}$ with Dependent Types

Recall: The System $\lambda{\rightarrow}$

The syntax of $\lambda{\rightarrow}$ consists of:

$$
\begin{aligned}
t, s &::= x \mid \lambda(x : \varphi).t \mid t\ s && \text{(Terms)} \\
\varphi, \psi &::= X \mid \varphi \to \psi && \text{(Types)} \\
\Gamma &::= \emptyset \mid \Gamma, (x : \varphi) && \text{(Contexts)}
\end{aligned}
$$

Recall: The System $\lambda{\rightarrow}$

The syntax of $\lambda{\rightarrow}$ consists of:

$$
\begin{aligned}
t, s &::= x \mid \lambda(x : \varphi).t \mid t\ s && \text{(Terms)} \\
\varphi, \psi &::= X \mid \varphi \to \psi && \text{(Types)} \\
\Gamma &::= \emptyset \mid \Gamma, (x : \varphi) && \text{(Contexts)}
\end{aligned}
$$

Judgements of the form $\Gamma \vdash t : \varphi$ can be derived via:

$$
\frac{}{\Gamma, x : \varphi \vdash x : \varphi}\ (Ax)
$$

$$
\frac{\Gamma, x : \varphi \vdash t : \psi}{\Gamma \vdash \lambda(x : \varphi).t : \varphi \to \psi}\ (\to_i)
\qquad
\frac{\Gamma \vdash t : \varphi \to \psi \qquad \Gamma \vdash s : \varphi}{\Gamma \vdash t\ s : \psi}\ (\to_e)
$$

What is a Dependent Type?

Definition

A **dependent type** is a type that depends on terms.

What is a Dependent Type?

Definition

A **dependent type** is a type that depends on terms.

Example

- The type $\mathbf{Vec}\mathbb{N}$ **n** of lists of natural numbers with length **n**.
- The type **Fin n** of numbers smaller than **n**.

(where **n** is a natural number)

What is a Dependent Type?

Definition

A **dependent type** is a type that depends on terms.

Example

- The type $\mathbf{Vec}\mathbb{N}$ **n** of lists of natural numbers with length **n**.
- The type **Fin n** of numbers smaller than **n**.

(where **n** is a natural number)

Remark

$\mathbf{Vec}\mathbb{N}$ and **Fin** themselves are **not** types, but type-families (indexed over natural numbers).

However, all of $\mathbf{Vec}\mathbb{N}$ **2**, **Fin 42**, $\mathbf{Vec}\mathbb{N}$ **123** are types.

The System $\lambda\mathrm{P}$

The syntax of $\lambda\mathrm{P}$ consists of:

$$t, s ::= x \mid \lambda(x : \varphi).t \mid t\ s \qquad \text{(Terms)}$$

The System λP

The syntax of λP consists of:

$$t, s ::= x \mid \lambda(x : \varphi).t \mid t\ s \qquad \text{(Terms)}$$

$$\varphi, \psi ::= X \mid \forall(x : \varphi).\psi \mid \varphi\ t \qquad \text{(Types)}$$

The System $\lambda\mathrm{P}$

The syntax of $\lambda\mathrm{P}$ consists of:

$$t, s ::= x \mid \lambda(x : \varphi).t \mid t\ s \qquad \text{(Terms)}$$

$$\varphi, \psi ::= X \mid \forall(x : \varphi).\psi \mid \varphi\ t \qquad \text{(Types)}$$

Shorthands:

$$\varphi \to \psi \text{ instead of } \forall(x : \varphi).\psi \text{ if } x \notin FV(\psi)$$

The syntax of $\lambda\mathrm{P}$ consists of:

$$t, s ::= x \mid \lambda(x : \varphi).t \mid t\ s \qquad \text{(Terms)}$$

$$\varphi, \psi ::= X \mid \forall(x : \varphi).\psi \mid \varphi\ t \qquad \text{(Types)}$$

$$\kappa ::= * \mid \mathit{\Pi}(x : \varphi).\kappa \qquad \text{(Kinds)}$$

Shorthands:

$$\varphi \to \psi \text{ instead of } \forall(x : \varphi).\psi \text{ if } x \notin FV(\psi)$$

The syntax of λP consists of:

$$t, s ::= x \mid \lambda(x : \varphi).t \mid t\ s \qquad \text{(Terms)}$$

$$\varphi, \psi ::= X \mid \forall(x : \varphi).\psi \mid \varphi\ t \qquad \text{(Types)}$$

$$\kappa ::= * \mid \Pi(x : \varphi).\kappa \qquad \text{(Kinds)}$$

Shorthands:

$$\varphi \to \psi \text{ instead of } \forall(x : \varphi).\psi \text{ if } x \notin FV(\psi)$$

$$\varphi \Rightarrow \kappa \text{ instead of } \Pi(x : \varphi).\kappa \text{ if } x \notin FV(\kappa)$$

The System $\lambda\mathrm{P}$

The syntax of $\lambda\mathrm{P}$ consists of:

$$t, s ::= x \mid \lambda(x : \varphi).t \mid t\ s \qquad \text{(Terms)}$$

$$\varphi, \psi ::= X \mid \forall(x : \varphi).\psi \mid \varphi\ t \qquad \text{(Types)}$$

$$\kappa ::= * \mid \Pi(x : \varphi).\kappa \qquad \text{(Kinds)}$$

$$\Gamma ::= \emptyset \mid \Gamma, (x : \varphi) \mid \Gamma, (X : \kappa) \qquad \text{(Contexts)}$$

Shorthands:

$$\varphi \to \psi \text{ instead of } \forall(x : \varphi).\psi \text{ if } x \notin FV(\psi)$$

$$\varphi \Rightarrow \kappa \text{ instead of } \Pi(x : \varphi).\kappa \text{ if } x \notin FV(\kappa)$$

The System $\lambda\mathrm{P}$

The syntax of $\lambda\mathrm{P}$ consists of:

$$t, s ::= x \mid \lambda(x : \varphi).t \mid t\ s \qquad \text{(Terms)}$$

$$\varphi, \psi ::= X \mid \forall(x : \varphi).\psi \mid \varphi\ t \qquad \text{(Types)}$$

$$\kappa ::= * \mid \Pi(x : \varphi).\kappa \qquad \text{(Kinds)}$$

$$\Gamma ::= \emptyset \mid \Gamma, (x : \varphi) \mid \Gamma, (X : \kappa) \qquad \text{(Contexts)}$$

Shorthands:

$$\varphi \to \psi \text{ instead of } \forall(x : \varphi).\psi \text{ if } x \notin FV(\psi)$$

$$\varphi \Rightarrow \kappa \text{ instead of } \Pi(x : \varphi).\kappa \text{ if } x \notin FV(\kappa)$$

Judgements are of the form:

$\Gamma \vdash t : \varphi$ (Typing) $\qquad$ $\Gamma \vdash \varphi : \kappa$ (Kinding)

$\Gamma \vdash \kappa$ (Kind formation)

The System λP

Typing rules:

$$\frac{\Gamma \vdash \varphi : *}{\Gamma, x : \varphi \vdash x : \varphi}\ (Ax_t)$$

$$\frac{\Gamma, x : \varphi \vdash t : \psi}{\Gamma \vdash \lambda(x : \varphi).t : \forall(x : \varphi).\psi}\ (\forall_i) \qquad \frac{\Gamma \vdash t : \forall(x : \varphi).\psi \qquad \Gamma \vdash s : \varphi}{\Gamma \vdash t\ s : \psi[x := s]}\ (\forall_e)$$

The System $\lambda\mathrm{P}$

Typing rules:

$$\frac{\Gamma \vdash \varphi : *}{\Gamma, x : \varphi \vdash x : \varphi}\ (Ax_t)$$

$$\frac{\Gamma, x : \varphi \vdash t : \psi}{\Gamma \vdash \lambda(x : \varphi).t : \forall(x : \varphi).\psi}\ (\forall_i) \qquad \frac{\Gamma \vdash t : \forall(x : \varphi).\psi \qquad \Gamma \vdash s : \varphi}{\Gamma \vdash t\ s : \psi[x := s]}\ (\forall_e)$$

Kinding rules:

$$\frac{\Gamma \vdash \kappa}{\Gamma, X : \kappa \vdash X : \kappa}\ (Ax_\varphi)$$

$$\frac{\Gamma, x : \varphi \vdash \psi : \star}{\Gamma \vdash \forall(x : \varphi).\psi : \star}\ (\Pi_i) \qquad \frac{\Gamma \vdash \varphi : (\Pi x : \psi).\kappa \qquad \Gamma \vdash t : \psi}{\Gamma \vdash \varphi\ t : \kappa[x := t]}\ (\Pi_e)$$

Kind formation rules:

$$\frac{}{\Gamma \vdash *}\,(Ax_\kappa) \qquad \frac{\Gamma, x : \varphi \vdash \kappa}{\Gamma \vdash \Pi(x : \varphi).\kappa}\,(\kappa_i)$$

Example
Let us derive

$$\Gamma \vdash sum\ 4\ v : \mathbb{N}$$

with

$$\begin{aligned}\Gamma = \{&\mathbb{N} : \star,\\ &Vec\mathbb{N} : \mathbb{N} \Rightarrow \star,\\ &sum : \forall(n : \mathbb{N}).\, Vec\mathbb{N}\ n \to \mathbb{N},\\ &4 : \mathbb{N},\\ &v :\ Vec\mathbb{N}\ 4\}\end{aligned}$$

Example
Let us derive

$$\Gamma \vdash \forall(n : \mathbb{N}).\forall(m : \mathbb{N}).eq_{\mathbb{N}}\ (add\ n\ m)\ (add\ m\ n) : \star$$

with

$$\begin{aligned}\Gamma = \{&\mathbb{N} : \star, \\ &add : \mathbb{N} \to \mathbb{N} \to \mathbb{N}, \\ &eq_{\mathbb{N}} : \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \star\}\end{aligned}$$

The Curry-Howard Isomorphism

The Curry-Howard Isomorphism

Let $\Sigma = \Sigma_P \cup \Sigma_f$ be a FO-signature. We define a context Γ_Σ:

The Curry-Howard Isomorphism

Let $\Sigma = \Sigma_P \cup \Sigma_f$ be a FO-signature. We define a context Γ_Σ:

- There is only one proper type: $(0 : \star) \in \Gamma_\Sigma$.

The Curry-Howard Isomorphism

Let $\Sigma = \Sigma_P \cup \Sigma_f$ be a FO-signature. We define a context Γ_Σ:

- There is only one proper type: $(0 : \star) \in \Gamma_\Sigma$.
- For every $P/n \in \Sigma_P$ we have $(P : 0 \Rightarrow^n 0) \in \Gamma_\Sigma$.

The Curry-Howard Isomorphism

Let $\Sigma = \Sigma_P \cup \Sigma_f$ be a FO-signature. We define a context Γ_Σ:

- There is only one proper type: $(0 : \star) \in \Gamma_\Sigma$.
- For every $P/n \in \Sigma_P$ we have $(P : 0 \Rightarrow^n 0) \in \Gamma_\Sigma$.
- For every $f/n \in \Sigma_f$ we have $(f : 0 \rightarrow^n 0) \in \Gamma_\Sigma$.

The Curry-Howard Isomorphism

Let $\Sigma = \Sigma_P \cup \Sigma_f$ be a FO-signature. We define a context Γ_Σ:

- There is only one proper type: $(0 : \star) \in \Gamma_\Sigma$.
- For every $P/n \in \Sigma_P$ we have $(P : 0 \Rightarrow^n 0) \in \Gamma_\Sigma$.
- For every $f/n \in \Sigma_f$ we have $(f : 0 \to^n 0) \in \Gamma_\Sigma$.

Theorem

Let φ be a first-order formula consisting only of $\to$ and $\forall$. There exists a λ-term t such that $\Gamma_\Sigma \vdash t : \varphi$ iff φ is a theorem of intuitionistic FOL with signature Σ.

Let $\Sigma = \Sigma_P \cup \Sigma_f$ be a FO-signature. We define a context Γ_Σ:

- There is only one proper type: $(0 : \star) \in \Gamma_\Sigma$.
- For every $P/n \in \Sigma_P$ we have $(P : 0 \Rightarrow^n 0) \in \Gamma_\Sigma$.
- For every $f/n \in \Sigma_f$ we have $(f : 0 \rightarrow^n 0) \in \Gamma_\Sigma$.

Theorem

Let φ be a first-order formula consisting only of $\rightarrow$ and $\forall$. There exists a λ-term t such that $\Gamma_\Sigma \vdash t : \varphi$ iff φ is a theorem of intuitionistic FOL with signature Σ.

Remark (for further reading)

The Curry-Howard isomorphism for "full" FOL requires us to extend $\lambda\mathrm{P}$ with the following constructs:

- Product, sum and empty type (corresponding to $\wedge, \vee, \bot$)
- Dependent sum type (corresponding to existential quantification)

Properties of λP

Expressiveness of $\lambda\mathrm{P}$

Let us define a translation from $\lambda\mathrm{P}$ to $\lambda{\to}$:

Terms

$$\overline{x} := x$$

$$\overline{t\ s} := \bar{t}\ \overline{s}$$

$$\overline{\lambda(x:\varphi).t} := \lambda(x:\overline{\varphi}).\bar{t}$$

Expressiveness of $\lambda\mathrm{P}$

Let us define a translation from $\lambda\mathrm{P}$ to $\lambda{\to}$:

Terms	**Types**
$\overline{x} := x$	$\overline{X} := X$
$\overline{t\ s} := \bar{t}\ \overline{s}$	$\overline{\varphi\ t} := \overline{\varphi}$
$\overline{\lambda(x:\varphi).t} := \lambda(x:\overline{\varphi}).\bar{t}$	$\overline{\forall(x:\varphi).\psi} := \overline{\varphi} \to \overline{\psi}$

Expressiveness of $\lambda\mathrm{P}$

Let us define a translation from $\lambda\mathrm{P}$ to $\lambda{\to}$:

Terms	**Types**	**Contexts**
$\overline{x} := x$	$\overline{X} := X$	$\overline{\emptyset} := \emptyset$
$\overline{t\ s} := \bar{t}\ \overline{s}$	$\overline{\varphi\ t} := \overline{\varphi}$	$\overline{\Gamma, (x : \varphi)} := \overline{\Gamma}, (x : \overline{\varphi})$
$\overline{\lambda(x : \varphi).t} := \lambda(x : \overline{\varphi}).\bar{t}$	$\overline{\forall(x : \varphi).\psi} := \overline{\varphi} \to \overline{\psi}$	$\overline{\Gamma, (X : \kappa)} := \overline{\Gamma}$

Expressiveness of $\lambda\mathrm{P}$

Let us define a translation from $\lambda\mathrm{P}$ to $\lambda{\to}$:

Terms	Types	Contexts
$\overline{x} := x$	$\overline{X} := X$	$\overline{\emptyset} := \emptyset$
$\overline{t\ s} := \bar{t}\ \overline{s}$	$\overline{\varphi\ t} := \overline{\varphi}$	$\overline{\Gamma, (x : \varphi)} := \overline{\Gamma}, (x : \overline{\varphi})$
$\overline{\lambda(x : \varphi).t} := \lambda(x : \overline{\varphi}).\bar{t}$	$\overline{\forall(x : \varphi).\psi} := \overline{\varphi} \to \overline{\psi}$	$\overline{\Gamma, (X : \kappa)} := \overline{\Gamma}$

Lemma

If $\Gamma \vdash t : \varphi$ (in $\lambda\mathrm{P}$) then $\overline{\Gamma} \vdash \bar{t} : \overline{\varphi}$ (in $\lambda{\to}$).

Expressiveness of $\lambda\mathrm{P}$

Let us define a translation from $\lambda\mathrm{P}$ to $\lambda{\to}$:

Terms	**Types**	**Contexts**
$\overline{x} := x$	$\overline{X} := X$	$\overline{\emptyset} := \emptyset$
$\overline{t\ s} := \bar{t}\ \overline{s}$	$\overline{\varphi\ t} := \overline{\varphi}$	$\overline{\Gamma, (x : \varphi)} := \overline{\Gamma}, (x : \overline{\varphi})$
$\overline{\lambda(x : \varphi).t} := \lambda(x : \overline{\varphi}).\bar{t}$	$\overline{\forall(x : \varphi).\psi} := \overline{\varphi} \to \overline{\psi}$	$\overline{\Gamma, (X : \kappa)} := \overline{\Gamma}$

Lemma

If $\Gamma \vdash t : \varphi$ (in $\lambda\mathrm{P}$) then $\overline{\Gamma} \vdash \bar{t} : \overline{\varphi}$ (in $\lambda{\to}$).

Corollary

$\lambda\mathrm{P}$ can express (i.e. assign a type to) **exactly** *the same terms as $\lambda{\to}$.*

More Properties

Lemma

$\lambda\mathrm{P}$ *is strongly normalizing, i.e. there are no infinite reduction sequences.*

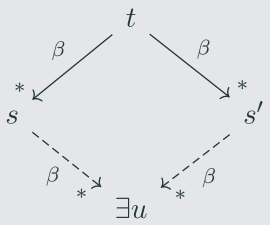
More Properties

Lemma

$\lambda\mathrm{P}$ *is strongly normalizing, i.e. there are no infinite reduction sequences.*

Lemma

$\lambda\mathrm{P}$ *has the Church-Rosser property, i.e.*



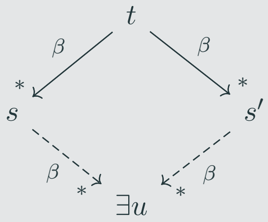
More Properties

Lemma

λP *is strongly normalizing, i.e. there are no infinite reduction sequences.*

Lemma

λP *has the Church-Rosser property, i.e.*



Corollary

For every term in λP *there* **exists** *a* **unique** *normal form.*

Applications of dependent types

The Curry-Howard Isomorphism in Practice

(Un)Safe Programming

Consider the following Haskell code:

```
data ListN where
  Nil  :: ListN
  Cons :: Nat -> ListN -> ListN
```

(Un)Safe Programming

Consider the following Haskell code:

```haskell
data ListN where
  Nil  :: ListN
  Cons :: Nat -> ListN -> ListN

head :: ListN -> Nat
head Nil = undefined
head (Cons n ns) = n
```

(Un)Safe Programming

Consider the following Haskell code:

```haskell
data ListN where
  Nil  :: ListN
  Cons :: Nat -> ListN -> ListN

head :: ListN -> Nat
head Nil = undefined
head (Cons n ns) = n
```

(Un)Safe Programming

Consider the following Haskell code:

```
data ListN where
  Nil  :: ListN
  Cons :: Nat -> ListN -> ListN

head :: ListN -> Nat
head Nil = undefined
head (Cons n ns) = n
```

This could benefit from dependent types!

Safe Programming

In Agda this can be expressed in a **safe** and **correct** way:

```
data VecN : ℕ → * where
  Nil  : VecN zero
  Cons : ∀ (i : ℕ) (n : ℕ) (ns : VecN i) → VecN (succ i)
```

Safe Programming

In Agda this can be expressed in a **safe** and **correct** way:

```
data VecN : ℕ → * where
  Nil  : VecN zero
  Cons : ∀ (i : ℕ) (n : ℕ) (ns : VecN i) → VecN (succ i)

head : ∀ (i : ℕ) → VecN (succ i) → ℕ
head (Cons i n ns) = n
```

Theorem Proving

We can use Agda to do first-order proofs (and more!):

```
data ℕ : * where
  zero : ℕ
  succ : ℕ → ℕ
```

Theorem Proving

We can use Agda to do first-order proofs (and more!):

```
data ℕ : * where
  zero : ℕ
  succ : ℕ → ℕ

data _≡_ : ℕ → ℕ → * where
  refl : ∀ {n : ℕ} → n ≡ n
```

Theorem Proving

We can use Agda to do first-order proofs (and more!):

```
data ℕ : * where
  zero : ℕ
  succ : ℕ → ℕ

data _≡_ : ℕ → ℕ → * where
  refl : ∀ {n : ℕ} → n ≡ n

_+_ : ℕ → ℕ → ℕ
zero     + n = n
(succ m) + n = succ (m + n)
```

Theorem Proving

We can use Agda to do first-order proofs (and more!):

```
data ℕ : * where
  zero : ℕ
  succ : ℕ → ℕ

data _≡_ : ℕ → ℕ → * where
  refl : ∀ {n : ℕ} → n ≡ n

_+_ : ℕ → ℕ → ℕ
zero     + n = n
(succ m) + n = succ (m + n)

+-assoc : ∀ (m : ℕ) (n : ℕ) (o : ℕ)
          → ((m + n) + o) ≡ (m + (n + o))
+-assoc zero     n o = refl
+-assoc (succ m) n o = cong succ (+-assoc m n o)
```

Conclusion

Conclusion

In this talk we have:

- Seen how to extend $\lambda\rightarrow$ to a dependent type system $\lambda\mathrm{P}$, that corresponds to FOL,

Conclusion

In this talk we have:

- Seen how to extend $\lambda\!\rightarrow$ to a dependent type system $\lambda\mathrm{P}$, that corresponds to FOL,
- Studied properties of $\lambda\mathrm{P}$,

Conclusion

In this talk we have:

- Seen how to extend $\lambda\!\rightarrow$ to a dependent type system $\lambda\mathrm{P}$, that corresponds to FOL,
- Studied properties of $\lambda\mathrm{P}$,
- Looked at applications of dependent types, by example of the dependently typed programming language Agda.

Conclusion

In this talk we have:

- Seen how to extend $\lambda\!\rightarrow$ to a dependent type system $\lambda\mathrm{P}$, that corresponds to FOL,
- Studied properties of $\lambda\mathrm{P}$,
- Looked at applications of dependent types, by example of the dependently typed programming language Agda.