Lambda calculus in Coq

Shin Tanaka, Paul Seip

July 4, 2022

Goals

Three-level goals:

1. Implement simply typed lambda calculus in Coq,
2. Prove some basic lemma's and theorems: subject reduction, generation lemma, etc.
3. Prove (weak or strong) normalization theorem.

Goals

Three-level goals:

1. Implement simply typed lambda calculus in Coq,
2. Prove some basic lemma's and theorems: subject reduction, generation lemma, etc.
3. Prove (weak or strong) normalization theorem.

(too ambitious!) ↑

Introduction

We compare two different implementations of the simply typed lambda calculus in Coq:

- Intrinsically typed (à la Church)
- Extrinsically typed (à la Curry).

Intrinsic Lambda Calculus

(Simple) **types** are defined inductively by

$$\begin{aligned} &\mathit{base} : \mathit{type} \\ &\mathit{arrow} : \mathit{type} \to \mathit{type} \to \mathit{type} \end{aligned}$$

Contexts are defined inductively by

$$\begin{aligned} &\mathit{empty} : \mathit{context} \\ &\mathit{append} : \mathit{context} \to \mathit{type} \to \mathit{context} \end{aligned}$$

Denote Γ, A for the context Γ extended with A, and $A \Rightarrow B$ for an arrow type.

We inductively define a predicate

$$- \in - : \mathit{type} \to \mathit{context} \to \mathit{prop}$$

that returns whether a variable $x : A$ is in context Γ,

$$\begin{aligned} \forall A, B : & \mathit{type}, \forall\ \Gamma : \mathit{context}, \\ & \mathit{Last} : A \in \Gamma, A. \\ & \mathit{Middle} : A \in \Gamma \to A \in \Gamma, B. \end{aligned}$$

We inductively define a predicate **judgement**

$$- \vdash - : \mathit{type} \to \mathit{context} \to \mathit{prop}$$

$$\begin{aligned} \forall A, B : & \mathit{type}, \forall\ \Gamma : \mathit{context}, \\ & \mathit{var} : A \in \Gamma \to \Gamma \vdash A, \\ & \mathit{abs} : \Gamma, A \vdash B \to \Gamma \vdash A \Rightarrow B, \\ & \mathit{app} : \Gamma \vdash A \Rightarrow B \to \Gamma \vdash A \to \Gamma \vdash B. \end{aligned}$$

A map between two variables of different contexts can be extended to a larger context.

Lemma (extension lemma)

$\forall\ \Gamma, \Delta : \mathit{context},$

$$\frac{\forall\ A : \mathit{type},\ A \in \Gamma \to A \in \Delta}{\forall\ A : \mathit{type},\ A \in \Gamma, B \to A \in \Delta, B}$$

We can "rebase" a type derivation to a different context which has the same variables.

Lemma (renaming lemma)

$\forall\ \Gamma, \Delta : \mathit{context},$

$$\frac{\forall\ A : \mathit{type},\ A \in \Gamma \to A \in \Delta}{\forall\ A : \mathit{type},\ \Gamma \vdash A \to \Delta \vdash A}$$

Two corollaries of renaming:

- *Thinning / Weakening*: $\Gamma \vdash A \to \Delta \vdash A$ if $\Gamma \subseteq \Delta$.
- *Permutation*: $\Gamma \vdash A \to \Gamma' \vdash A$ if Γ' is a permutation of Γ.

A map from variables of one context to terms in another context can be extended to a larger context.

Lemma (extension lemma 2)

$\forall\ \Gamma, \Delta : \mathit{context},$

$$\frac{\forall\ A : \mathit{type},\ A \in \Gamma \to \Delta \vdash A}{\forall\ A : \mathit{type},\ A \in \Gamma, B \to \Delta, B \vdash A}$$

Using this we can define simultaneous substitution, if variables in one context map to terms in another context, then terms in the first context map to terms in the second context.

Lemma (simultaneous substitution)

$\forall\ \Gamma, \Delta : \mathit{context},$

$$\frac{\forall\ A : \mathit{type},\ A \in \Gamma \to \Delta \vdash A}{\forall\ A : \mathit{type},\ \Gamma \vdash A \to \Delta \vdash A}$$

We get single substitution as a special case,

Lemma (single substitution)

$\forall\ \Gamma : \mathit{context},\ \forall\ A, B : \mathit{type},$

$$\frac{\Gamma, B \vdash A \to \Gamma \vdash B}{\Gamma \vdash A}$$

We inductively define a predicate β-reduction

$$\to_\beta : (\Gamma \vdash A) \to (\Gamma \vdash A) \to \mathit{Prop}$$

By the following four rules

- $(L \to_\beta L') \to (L \cdot M \to_\beta L' \cdot M)$,
- $(M \to_\beta M') \to (L \cdot M \to_\beta L \cdot M')$,
- $(M \to_\beta N) \to (\lambda x.M \to_\beta \lambda x.N)$,
- $(\lambda x.M)N \to_\beta M[x := N]$.

We define

$$\twoheadrightarrow_\beta : (\Gamma \vdash A) \to (\Gamma \vdash A) \to \mathit{Prop}$$

as the reflexive and transitive closure of this relation,

- refl: $M \twoheadrightarrow_\beta M$,
- trans: $(L \to_\beta M) \to (M \twoheadrightarrow_\beta N) \to (L \twoheadrightarrow_\beta N)$.

Normalization

We define inductively a predicate for normal forms,
$NF : (\Gamma \vdash A) \to Prop$,

- nf_var: $NF\ x$,
- nf_abs: $NF\ M \to NF\ (\lambda x.M)$,
- nf_app: $NF\ M \to NF\ N \to M \neq (\lambda x.L) \to NF\ M \cdot N$.

Normalization

We define inductively a predicate for normal forms,
$NF : (\Gamma \vdash A) \to Prop$,

- nf_var: $NF\ x$,
- nf_abs: $NF\ M \to NF\ (\lambda x.M)$,
- nf_app: $NF\ M \to NF\ N \to M \neq (\lambda x.L) \to NF\ M \cdot N$.

Finally, we can state weak normalization:

Theorem (weak normalization)

$\forall\ (M : \Gamma \vdash A),\ \exists\ (N : \Gamma \vdash A),\ (NF\ N)\ \wedge\ M \twoheadrightarrow_\beta N.$

Difficulties and reflections

- *Problem* : We want a function
  $lookup : context \to nat \to type$, returning the n-th variable in a context. What to do with $lookup\ \emptyset\ n$? In Coq we cannot throw an error.
- *Solution*: dependent types!

$$lookup : (context\ n) \to (m : nat) \to \{m \leq n\} \to type,$$

where $context\ n$ is the type of contexts with n elements.

Difficulties and reflections

- *Problem* : We want a function
  $lookup : context \to nat \to type$, returning the n-th variable in a context. What to do with $lookup\ \emptyset\ n$? In Coq we cannot throw an error.
- *Solution*: dependent types!

$$lookup : (context\ n) \to (m : nat) \to \{m \leq n\} \to type,$$

  where $context\ n$ is the type of contexts with n elements.
- *Reflection* : in the intrinsically typed system, our goal (subject reduction, generation lemma) is inherently true!
- We developed a second implementation of an extrinsically typed system.

Extrinsic Lambda Calculus

Unlike Intrinsic system, we define **terms** and **types** separately, then make a connection between the two via **contexts** and **judgments**.

types

$$\mathit{Type\_Base} : \mathit{type}$$

$$\mathit{Type\_Arrow} : \mathit{type} \to \mathit{type} \to \mathit{type}$$

- For types σ and τ, we have $\sigma \twoheadrightarrow \tau := \mathit{Type\_Arrow}(\sigma, \tau)$.

terms

$$\mathit{Term\_Var} : \mathit{string} \to \mathit{term}$$

$$\mathit{Term\_App} : \mathit{term} \to \mathit{term} \to \mathit{term}$$

$$\mathit{Term\_Abs} : \mathit{string} \to \mathit{type} \to \mathit{term} \to \mathit{term}$$

- For each string x, we have a variable $\mathit{Term\_Var}(x)$ (we can mentally regard strings as variables).
- For terms M and N, we have $MN := \mathit{Term\_App}(M, N)$.
- For a 'variable' x, type τ and a term M, we have $\lambda x : \tau.M := \mathit{Term\_Abs}(x, \tau, M)$.

Extrinsic Lambda Calculus

Unlike Intrinsic system, we define **terms** and **types** separately, then make a connection between the two via **contexts** and **judgments**.

types

$$\mathit{Type\_Base} : \mathit{type}$$

$$\mathit{Type\_Arrow} : \mathit{type} \to \mathit{type} \to \mathit{type}$$

- For types σ and τ, we have $\sigma \twoheadrightarrow \tau := \mathit{Type\_Arrow}(\sigma, \tau)$

terms

$$\mathit{Term\_Var} : \mathit{string} \to \mathit{term}$$

$$\mathit{Term\_App} : \mathit{term} \to \mathit{term} \to \mathit{term}$$

$$\mathit{Term\_Abs} : \mathit{string} \to \mathit{type} \to \mathit{term} \to \mathit{term}$$

Remark: a variable is not assigned a type.

- For each string x, we have a variable $\mathit{Term\_Var}(x)$ (we can mentally regard strings as variables).
- For terms M and N, we have $MN := \mathit{Term\_App}(M, N)$.
- For a 'variable' x, type τ and a term M, we have $\lambda x : \tau.M := \mathit{Term\_Abs}(x, \tau, M)$.

Give types to terms via judgments

Usually, a context Γ is a list of $x : \sigma$. A judgement $\Gamma \vdash M : \tau$ and the derivation rules are defined as follows:

(var) If $x : \sigma \in \Gamma$, then $\Gamma \vdash x : \sigma$.

(app) If $\Gamma \vdash M : \sigma \to \tau$ and $\Gamma \vdash N : \sigma$, then $\Gamma \vdash MN : \tau$.

(abs) If $\Gamma, x : \sigma \vdash M : \tau$ then $\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau$.

However, our current system extrinsically assigns types to terms. How?

Give types to terms via judgments

Usually, a context Γ is a list of $x : \sigma$. A judgement $\Gamma \vdash M : \tau$ and the derivation rules are defined as follows:

(var) If $x : \sigma \in \Gamma$, then $\Gamma \vdash x : \sigma$.

(app) If $\Gamma \vdash M : \sigma \to \tau$ and $\Gamma \vdash N : \sigma$, then $\Gamma \vdash MN : \tau$.

(abs) If $\Gamma, x : \sigma \vdash M : \tau$ then $\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau$.

However, our current system extrinsically assigns types to terms.
How?
Define Γ as a (partial) function from 'variables' to types instead of a list. This roughly means

$$\Gamma : \mathit{string} \to \mathit{type}.$$

(Recall that to get a variable we need to apply $\mathit{Term\_Var}$. So, for a string x, a variable $\mathit{Term\_Var}(x)$ is assigned a type $\Gamma(x)$.)

Give types to terms via judgments

Usually, a context Γ is a list of $x : \sigma$. A judgement $\Gamma \vdash M : \tau$ and the derivation rules are defined as follows:

(var) If $x : \sigma \in \Gamma$, then $\Gamma \vdash x : \sigma$.

(app) If $\Gamma \vdash M : \sigma \twoheadrightarrow \tau$ and $\Gamma \vdash N : \sigma$, then $\Gamma \vdash MN : \tau$.

(abs) If $\Gamma, x : \sigma \vdash M : \tau$ then $\Gamma \vdash \lambda x : \sigma.M : \sigma \twoheadrightarrow \tau$.

However, our current system extrinsically assigns types to terms.
How?
Define Γ as a (partial) function from 'variables' to types instead of a list. This roughly means

$\Gamma : string \to type$. Later, we'll modify Γ a bit .

(Recall that to get a variable we need to apply *Term_Var*. So, for a string x, a variable $Term\_Var(x)$ is assigned a type $\Gamma(x)$.)

Give types to terms via judgments

Usually, a context Γ is a list of $x : \sigma$. A judgement $\Gamma \vdash M : \tau$ and the derivation rules are defined as follows:

(*var*) If $x : \sigma \in \Gamma$, then $\Gamma \vdash x : \sigma$.

(*app*) If $\Gamma \vdash M : \sigma \to \tau$ and $\Gamma \vdash N : \sigma$, then $\Gamma \vdash MN : \tau$.

(*abs*) If $\Gamma, x : \sigma \vdash M : \tau$ then $\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau$.

However, our current system extrinsically assigns types to terms.
How?
Define Γ as a (partial) function from 'variables' to types instead of a list. This roughly means

$$\Gamma : \mathit{string} \to \mathit{type}.$$

Later, we'll modify Γ a bit .

(Recall that to get a variable we need to apply *Term.Var*. So for a string x, a variab

Since Γ is partial, $\Gamma(x)$ may not be defined.

Context as function

Using this partial map Γ, instead of $x : \sigma \in \Gamma$, our derivation rules look like:

(*var*)	If $\Gamma(x) = \texttt{Some}\ \ \sigma$, then $\Gamma \vdash \mathit{Term\_Var}(x) : \sigma$.
(*app*)	If $\Gamma \vdash M : \sigma \twoheadrightarrow \tau$ and $\Gamma \vdash N : \sigma$, then $\Gamma \vdash MN : \tau$.
(*abs*)	If $\Gamma, x : \sigma \vdash M : \tau$ then $\Gamma \vdash \lambda x : \sigma.M : \sigma \twoheadrightarrow \tau$.

Context as function

Using this partial map Γ, instead of $x : \sigma \in \Gamma$, our derivation rules look like:

(*var*)	If $\Gamma(x) = \texttt{Some}\ \ \sigma$, then $\Gamma \vdash \mathit{Term\_Var}(x) : \sigma$.
(*app*)	If $\Gamma \vdash M : \sigma \to \tau$ and $\Gamma \vdash N : \sigma$, then $\Gamma \vdash MN : \tau$.
(*abs*)	If $\Gamma, x : \sigma \vdash M : \tau$ then $\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau$.

What is `Some`?

Context as function

Using this partial map Γ, instead of $x : \sigma \in \Gamma$, our derivation rules look like:

(*var*) If $\Gamma(x) = \texttt{Some}\ \ \sigma$, then $\Gamma \vdash \mathit{Term\_Var}(x) : \sigma$.

(*app*) If $\Gamma \vdash M : \sigma \twoheadrightarrow \tau$ and $\Gamma \vdash N : \sigma$, then $\Gamma \vdash MN : \tau$.

(*abs*) If $\Gamma, x : \sigma \vdash M : \tau$ then $\Gamma \vdash \lambda x : \sigma.M : \sigma \twoheadrightarrow \tau$.

What is `Some`?

Recall from Paul's part that *lookup* : *context* $\rightarrow$ *nat* $\rightarrow$ *type*, returning the *n*-th variable in a context. What to do with *lookup* $\varnothing$ *n* ? In Coq we cannot throw an error. Ans. dependent types.

`Some` is another way to solve this issue. Let us look at this a bit more.

Use option (`Some` and `None`) to define partial map

$$
\begin{aligned}
&\mathit{Inductive}\ \mathit{option}\ (A : \mathit{Type}) : \mathit{Type} :=\\
&\quad \mid\ \mathit{Some} : A \to \mathit{option}\ A\\
&\quad \mid\ \mathit{None} : \mathit{option}\ A.
\end{aligned}
$$

Use option (`Some` and `None`) to define partial map

$$
\begin{aligned}
&\mathit{Inductive}\ \mathit{option}\ (A : \mathit{Type}) : \mathit{Type} :=\\
&\quad \mid\ \mathit{Some} : A \to \mathit{option}\ A\\
&\quad \mid\ \mathit{None} : \mathit{option}\ A.
\end{aligned}
$$

Coq can return an error by using `None`.

Use option (`Some` and `None`) to define partial map

$$
\begin{aligned}
&\mathit{Inductive}\ \mathit{option}\ (A : \mathit{Type}) : \mathit{Type} := \\
&\quad | \quad \mathit{Some} : A \to \mathit{option}\ A \\
&\quad | \quad \mathit{None} : \mathit{option}\ A.
\end{aligned}
$$

Coq can return an error by using `None`.

We defined $\Gamma : \mathit{string} \to \mathit{type}$. When Γ is a total map, it is OK, but since it is partial, we want to modify it as

$$\Gamma : \mathit{string} \to \mathit{option}(\mathit{type}),$$

Use option (`Some` and `None`) to define partial map

$$\begin{aligned} &\mathit{Inductive\ option}\ (A : \mathit{Type}) : \mathit{Type} := \\ &\quad | \quad \mathit{Some} : A \to \mathit{option}\ A \\ &\quad | \quad \mathit{None} : \mathit{option}\ A. \end{aligned}$$

Coq can return an error by using `None`.

We defined $\Gamma : \mathit{string} \to \mathit{type}$. When Γ is a total map, it is OK, but since it is partial, we want to modify it as

$$\Gamma : \mathit{string} \to \mathit{option}(\mathit{type}),$$

so that

$$\Gamma(x) = \begin{cases} \texttt{Some}\ \ \sigma & \text{if } x \in \mathit{dom}(\Gamma) \text{ and we want to assign } x \text{ a type } \sigma. \\ \texttt{None} & \text{if } x \notin \mathit{dom}(\Gamma). \end{cases}$$

Use option (Some and None) to define partial map

$$\begin{aligned} &\mathit{Inductive\ option}\ (A : \mathit{Type}) : \mathit{Type} := \\ &\quad | \quad \mathit{Some} : A \to \mathit{option}\ A \\ &\quad | \quad \mathit{None} : \mathit{option}\ A. \end{aligned}$$

Coq can return an error by using None.

We defined $\Gamma : \mathit{string} \to \mathit{type}$. When Γ is a total map, it is OK, but since it is partial, we want to modify it as

$$\Gamma : \mathit{string} \to \mathit{option}(\mathit{type}),$$

so that

$$\Gamma(x) = \begin{cases} \texttt{Some}\ \ \sigma & \text{if } x \in \mathit{dom}(\Gamma) \text{ and we want to assign } x \text{ a type } \sigma. \\ \texttt{None} & \text{if } x \notin \mathit{dom}(\Gamma). \end{cases}$$

Remark: option is the same as Maybe monad in Haskell (Just/Nothing instead of Some/None).

Context as function (Recap)

$$\Gamma : \mathit{string} \to \mathit{option}(\mathit{type}),$$

$$\Gamma(x) = \begin{cases} \texttt{Some}\ \ \sigma & \text{if } x \in \mathit{dom}(\Gamma) \text{ and we want to assign } x \text{ a type } \sigma. \\ \texttt{None} & \text{if } x \notin \mathit{dom}(\Gamma). \end{cases}$$

Using this partial map Γ, instead of $x : \sigma \in \Gamma$, our derivation rules look like:

(*var*)	If $\Gamma(x) = \texttt{Some}\ \ \sigma$, then $\Gamma \vdash \mathit{Term\_Var}(x) : \sigma$.
(*app*)	If $\Gamma \vdash M : \sigma \to \tau$ and $\Gamma \vdash N : \sigma$, then $\Gamma \vdash MN : \tau$.
(*abs*)	If $\Gamma, x : \sigma \vdash M : \tau$ then $\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau$.

Context Γ extrinsically assigns a type to a variable!

Update context (partial map extension)

When extending context Γ to context $\Gamma, x : \sigma$, we update this partial map Γ using *update* function.

Update context (partial map extension)

When extending context Γ to context $\Gamma, x : \sigma$, we update this partial map Γ using *update* function. For context Γ, string x and type σ, $\mathit{update}(\Gamma, x, \sigma) : \mathit{string} \to \mathit{option}(\mathit{type})$ is an updated context roughly corresponding to $\Gamma, x : \sigma$

Update context (partial map extension)

When extending context Γ to context $\Gamma, x : \sigma$, we update this partial map Γ using *update* function. For context Γ, string x and type σ, $\mathit{update}(\Gamma, x, \sigma) : \mathit{string} \to \mathit{option}(\mathit{type})$ is an updated context roughly corresponding to $\Gamma, x : \sigma$ s.t.

$$\mathit{update}(\Gamma, x, \sigma)(y) := \begin{cases} \texttt{Some}\ \ \sigma & \text{if } x = y. \\ \Gamma(y) & \text{if } x \neq y. \end{cases}$$

Update context (partial map extension)

When extending context Γ to context $\Gamma, x : \sigma$, we update this partial map Γ using *update* function. For context Γ, string x and type σ, $\mathit{update}(\Gamma, x, \sigma) : \mathit{string} \to \mathit{option}(\mathit{type})$ is an updated context roughly corresponding to $\Gamma, x : \sigma$ s.t.

$$\mathit{update}(\Gamma, x, \sigma)(y) := \begin{cases} \texttt{Some}\ \ \sigma & \text{if } x = y. \\ \Gamma(y) & \text{if } x \neq y. \end{cases}$$

Note that this allows an expression such as $\mathit{update}(\mathit{update}(\Gamma, x, \sigma), x, \tau)$, whereas a context like $\Gamma, x : \sigma, x : \tau$ is not usually allowed (especially when $\sigma \neq \tau$).

Axiom: Well-formed context

$\Gamma, x : \sigma \vdash M : \tau \to (\Gamma(x) = \mathit{None})$.

Recall $\Gamma(x) = \mathit{None}$ means $x \notin \mathit{dom}(\Gamma)$. The axiom says we update a context only by a fresh string ('variable').

Substitution and reduction

Recall substitution was quite abstract in the intrinsic system.

Lemma (single substitution)

$\forall\ \Gamma : \mathit{context},\ \forall\ A, B : \mathit{type},\quad \dfrac{\Gamma, B \vdash A \to \Gamma \vdash B}{\Gamma \vdash A}$

Since we directly deal with terms (as usual) in this extrinsic system, substitution is just as we expect.

```
Fixpoint subst (x :string) (s t:term) : term :=
match t with
| Term_Var y ⇒ if x = y then s else t
| t1 t2 ⇒ (subst x s t1 )(subst x s t2 )
| λy : σ.t1 ⇒ if x = y then t else λy : σ.(subst x s t1)
end.
```

$[x := s]t :=$ subst $(x\ s\ t)$ (i.e. substitute s for x in t)

β-reduction is defined in the same manner as in the intrinsic one.

Lemmas 1

x_1, x_2, x are all strings.

Lemma (Permutation)

$(update(update\,\Gamma\,x_1\,\sigma)\,x_2\,\tau) \vdash M : \rho$
$\rightarrow (update(update\,\Gamma\,x_2\,\tau)\,x_1\,\sigma) \vdash M : \rho.$

Or, in a more usual notation:

$\Gamma, x_1 : \sigma, x_2 : \tau \vdash M : \rho \rightarrow \Gamma, x_2 : \tau, x_1 : \sigma \vdash M : \rho.$

In the following, instead of *update*, we use usual context notations with occasional abuse of notation.

Lemma (Generation)

(*var*) $\Gamma \vdash Term\_Var(x) : \sigma \rightarrow \Gamma(x) = Some\ (\sigma).$
(*app*) $\Gamma \vdash MN : \sigma \rightarrow \exists \tau.\ (\Gamma \vdash M : \tau \twoheadrightarrow \sigma \wedge \Gamma \vdash N : \tau)\ .$
(*abs*) $\Gamma \vdash \lambda x : \sigma.M : \tau \rightarrow \exists \rho.\ (\Gamma, x : \sigma \vdash M : \rho \wedge \tau = \sigma \twoheadrightarrow \rho).$

Lemmas 2

Lemma (Uniqueness)

$\Gamma \vdash M : \sigma \to \Gamma \vdash M : \tau \to \sigma = \tau.$

Lemma (Weakening)

$x \notin dom(\Gamma) \to \Gamma \vdash M : \sigma \to \Gamma, x : \tau \vdash M : \sigma.$

Lemma (Substitution)

$\Gamma, x : \sigma \vdash M : \tau \to \Gamma \vdash N : \sigma \to \Gamma \vdash [x := N]M : \tau.$

Lemma (Subject Reduction)

$\Gamma \vdash M : \sigma \to (M \twoheadrightarrow_\beta N) \to \Gamma \vdash N : \sigma.$

Some coding experience report on extrinsic l.c. in Coq

Recall: $$update(\Gamma, x, \sigma)(y) := \begin{cases} \texttt{Some}\ \sigma & \text{if } x = y. \\ \Gamma(y) & \text{if } x \neq y. \end{cases}$$

- Because there are several case distinctions like *update*, `case_eq (eqb_string x y)` was used a lot.
- Most lemmas were proven by induction on something. Sometimes, if we started a proof by `intros`, we lost some information in IH. So, we learned this technique using `revert` after `intros` so that IH has a proper quantification.
- `admit` is useful, when there are many induction steps.

Comparison

Intrinsic lambda calculus:

- Benefit: compact code, abstract implementation, some results are for free,
- Downside: conceptually hard.

Extrinsic lambda calculus:

- Benefit: intuitively clear implementation,
- Downside: more lines of code.

References

Intrinsic system:

`https://plfa.inf.ed.ac.uk/DeBruijn/`

Extrinsic system:

`https://softwarefoundations.cis.upenn.edu/plf-current/Stlc.html`