INF551: Simply typed λ-calculus

Samuel Mimram

École Polytechnique
Part I

Introduction
Putting it all together

What we have done so far.

1. We have seen that types in OCaml could intuitively be interpreted as formulas.
2. We have formally defined what is a formula and a proof.
3. We have formally defined the core of a functional language ($\lambda$-calculus).

and now we put it all together:

4. We define a typing system for $\lambda$-calculus and show that it corresponds precisely to building proofs.
In other words,

PROGRAM = PROOF

Or, more precisely, there is a bijection between

- types and formulas,
- programs of type $A$ and proofs of $A$, 
Putting it all together

In other words,

\[ \text{PROGRAM} = \text{PROOF} \]

Or, more precisely, there is a bijection between

- types and formulas,
- programs of type \( A \) and proofs of \( A \),
- reductions of programs and cut elimination.
In order to have things as simple as possible, we will first focus on functions.

But, we will see that it extends to more realistic programming languages.
Part II

Simply typed $\lambda$-calculus
Simple types

The **simple types** are generated by the grammar

\[ A, B ::= X \mid A \rightarrow B \]

where \( X \) is a variable.

For instance, we have a type

\[(X \rightarrow Y) \rightarrow X\]

which roughly corresponds to OCaml's

\[(’a -> ’b) -> ’a\]
The **simple types** are generated by the grammar

\[ A, B ::= X \mid A \rightarrow B \]

where \( X \) is a variable.

For instance, we have a type

\[ (X \rightarrow Y) \rightarrow X \]

which roughly corresponds to OCaml’s

\[ ('a \rightarrow 'b) \rightarrow 'a \]

By convention, arrows are associated on the **right**:

\[ X \rightarrow Y \rightarrow Z = X \rightarrow (Y \rightarrow Z) \]
Terms

The programs we consider are \( \lambda \text{-terms} \) generated by the grammar

\[
t, u ::= x \mid t u \mid \lambda x^A.t
\]

where \( x \) is a variable and \( A \) is a type.

All the abstractions carry the type of the abstracted variable:

\[
\lambda x^{\text{int}}.x
\]

corresponds to OCaml's

\[
\text{fun (x : int) -> x}
\]
The programs we consider are $\lambda$-terms generated by the grammar

$$t, u ::= x \mid t \ u \mid \lambda x^A. t$$

where $x$ is a variable and $A$ is a type.

All the abstractions carry the type of the abstracted variable:

$$\lambda x^{\text{int}}. x$$

corresponds to OCaml’s

$$\text{fun (x : int) -> x}$$

This is called Church style.
The programs we consider are  $\lambda$-terms generated by the grammar

$$t, u ::= x \mid tu \mid \lambda x \; . t$$

where $x$ is a variable and $A$ is a type.

All the abstractions carry the type of the abstracted variable:

$$\lambda x \; . x$$

corresponds to OCaml's

$$\text{fun } (x \; \text{)} \to x$$

This is called **Church style** (the other one being **Curry style**).
We are now going assign types to terms. For instance, the type of 

\[ \lambda x^A.x \rightarrow A \rightarrow A \rightarrow A \]

As usual, we will formulate this by using inference rules on sequents.
We are now going assign types to terms. For instance, the type of

<table>
<thead>
<tr>
<th>term</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x^A.x$</td>
<td>$A \rightarrow A$</td>
</tr>
</tbody>
</table>

As usual, we will formulate this by using inference rules on sequents.
We are now going assign types to terms. For instance, the type of

<table>
<thead>
<tr>
<th>term</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x^A.x )</td>
<td>( A \to A )</td>
</tr>
<tr>
<td>( \lambda f^{A \to A} \lambda x^A. f(fx) )</td>
<td></td>
</tr>
</tbody>
</table>

As usual, we will formulate this by using inference rules on sequents.
We are now going assign types to terms. For instance, the type of

<table>
<thead>
<tr>
<th>term</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x^A.x$</td>
<td>$A \rightarrow A$</td>
</tr>
<tr>
<td>$\lambda f^{A \rightarrow A}.\lambda x^A.f(fx)$</td>
<td>$(A \rightarrow A) \rightarrow A \rightarrow A$</td>
</tr>
</tbody>
</table>

As usual, we will formulate this by using inference rules on sequents.
We are now going assign types to terms. For instance, the type of

<table>
<thead>
<tr>
<th>term</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x^A.x$</td>
<td>$A \rightarrow A$</td>
</tr>
<tr>
<td>$\lambda f^{A\rightarrow A}.\lambda x^A.f(x)$</td>
<td>$(A \rightarrow A) \rightarrow A \rightarrow A$</td>
</tr>
<tr>
<td>$\lambda x^A.xx$</td>
<td></td>
</tr>
</tbody>
</table>

As usual, we will formulate this by using inference rules on sequents.
We are now going assign types to terms. For instance, the type of

<table>
<thead>
<tr>
<th>term</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x^A.x )</td>
<td>( A \rightarrow A )</td>
</tr>
<tr>
<td>( \lambda f^{A \rightarrow A}.\lambda x^A. f(x) )</td>
<td>( (A \rightarrow A) \rightarrow A \rightarrow A )</td>
</tr>
<tr>
<td>( \lambda x^A.xx )</td>
<td>not well-typed!</td>
</tr>
</tbody>
</table>

As usual, we will formulate this by using inference rules on sequents.
A context $\Gamma$ is a list

\[ x_1 : A_1, \ldots, x_n : A_n \]

of pairs consisting of a variable $x_i$ and a type $A_i$.

It can be read as “I assume that the variable $x_i$ has type $A_i$ for every index $i$”.
A context $\Gamma$ is a list

\[ x_1 : A_1, \ldots, x_n : A_n \]

of pairs consisting of a variable $x_i$ and a type $A_i$.

It can be read as “I assume that the variable $x_i$ has type $A_i$ for every index $i$”.

The domain of the context $\Gamma$ is $\text{dom}(\Gamma) = \{x_1, \ldots, x_n\}$.
A context $\Gamma$ is a list

$$x_1 : A_1, \ldots, x_n : A_n$$

of pairs consisting of a variable $x_i$ and a type $A_i$.

It can be read as “I assume that the variable $x_i$ has type $A_i$ for every index $i$”.

The domain of the context $\Gamma$ is $\text{dom}(\Gamma) = \{x_1, \ldots, x_n\}$.

Given $x \in \text{dom}(\Gamma)$ we write $\Gamma(x)$ for the type of $x$:

$$(\Gamma, x : A)(x) = A \quad (\Gamma, y : A)(x) = \Gamma(x)$$

(note that a variable might occur multiple times).
A **sequent** is a triple noted

\[ \Gamma \vdash t : A \]

where

- \( \Gamma \) is a context,
- \( t \) is a term,
- \( A \) is a type.

Read as “under the typing assumptions for the variables in \( \Gamma \), the term \( t \) has type \( A \)”. 

**Sequents**
Typing rules

The **typing rules** are

\[ \Gamma, x : A \vdash t : B \quad \rightarrow_1 \quad \Gamma \vdash \lambda x^A.t : A \rightarrow B \]

\[ \Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A \quad \rightarrow_E \quad \Gamma \vdash tu : B \]

with \( x \in \text{dom}(\Gamma) \) for \((ax)\).
The **typing rules** are

\[
\Gamma \vdash x : \Gamma(x) \quad (\text{ax})
\]

\[
\Gamma, x : A \vdash t : B \\
\Gamma \vdash \lambda x^A.t : A \to B \quad (\to_I)
\]

\[
\Gamma \vdash t : A \to B \\
\Gamma \vdash u : A \\
\Gamma \vdash t u : B \quad (\to_E)
\]

with \( x \in \text{dom}(\Gamma) \) for (ax).

Note that depending on the term only one rule applies.
We say that a term $t$ has type $A$ in context $\Gamma$ when $\Gamma \vdash t : A$ is derivable.
We say that a term $t$ has type $A$ in context $\Gamma$ when $\Gamma \vdash t : A$ is derivable.

We say that a term $t$ has type $A$ when $\vdash t : A$ is derivable.
We say that *a term* \( t \) *has type* \( A \) *in context* \( \Gamma \) when \( \Gamma \vdash t : A \) is derivable.

We say that *a term* \( t \) *has type* \( A \) when \( \vdash t : A \) is derivable.

NB: even in this last case, we still need the context to inductively check the type, because of the rule \((\to I)\).
An example of typing derivation

For instance we claim that

$$\lambda f^A \cdot \lambda x^A . f (fx)$$

has type

$$(A \rightarrow A) \rightarrow A \rightarrow A$$

which means that

$$\vdash \lambda f^A \cdot \lambda x^A . f (fx) : (A \rightarrow A) \rightarrow A \rightarrow A$$

is derivable.
An example of typing derivation

\[ \vdash \lambda f^A \rightarrow A. \lambda x^A. f(x) : (A \rightarrow A) \rightarrow A \rightarrow A \]
An example of typing derivation

\[ f : A \to A \vdash \lambda x^A.f(x) : A \to A \]

\[ \vdash \lambda f^{A\to A}.\lambda x^A.f(x) : (A \to A) \to A \to A \]  

(\to_1)
An example of typing derivation

\[
\begin{align*}
f &: A \to A, x &: A \vdash f(fx) &: A \\
\hline
f &: A \to A \vdash \lambda x^A.f(x) &: A \to A \\
\hline
\vdash \lambda f^{A\to A}.\lambda x^A.f(x) &: (A \to A) \to A \to A
\end{align*}
\]
An example of typing derivation

\[\Gamma \vdash \lambda f : A \to A.\lambda x : A.f(x) : (A \to A) \to A \to A\]
An example of typing derivation

\[ \begin{align*}
\Gamma & \vdash f : A \rightarrow A \quad \text{(ax)} \\
\Gamma & \vdash f x : A \\
\Gamma & \vdash f : A \rightarrow A, x : A \vdash f(f x) : A \\
\vdash f : A \rightarrow A \vdash \lambda x.A.f(x) : A \rightarrow A \\
\vdash \lambda f.A.\lambda x.A.f(x) : (A \rightarrow A) \rightarrow A \rightarrow A
\end{align*} \]
An example of typing derivation

\[
\begin{array}{c}
\Gamma \vdash f : A \rightarrow A \\
\hline
(\text{ax})
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash f : A \rightarrow A \\
\hline
\text{(}\rightarrow E\text{)}
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash x : A \\
\hline
\text{(}\rightarrow E\text{)}
\end{array}

\begin{array}{c}
\hline
\Gamma \vdash fx : A
\end{array}
\quad
\begin{array}{c}
\hline
\text{(}\rightarrow E\text{)}
\end{array}

\begin{array}{c}
\quad
\hline
f : A \rightarrow A, x : A \vdash f(fx) : A
\end{array}
\quad
\begin{array}{c}
\hline
\text{(}\rightarrow I\text{)}
\end{array}

\begin{array}{c}
\Gamma \vdash \lambda x^A.f(fx) : A \rightarrow A \\
\hline
\text{(}\rightarrow I\text{)}
\end{array}

\begin{array}{c}
\vdash \lambda f^A \rightarrow A.\lambda x^A.f(fx) : (A \rightarrow A) \rightarrow A \rightarrow A
\end{array}
\]
An example of typing derivation

\[ \Gamma \vdash f : A \rightarrow A \]  \hspace{1cm} \Gamma \vdash x : A

\[ \Gamma \vdash fx : A \]  \hspace{1cm} (\rightarrow_E)

\[ f : A \rightarrow A, x : A \vdash f(fx) : A \]  \hspace{1cm} (\rightarrow_E)

\[ f : A \rightarrow A \vdash \lambda x^A.f(fx) : A \rightarrow A \]  \hspace{1cm} (\rightarrow_I)

\[ \vdash \lambda f^{A \rightarrow A}.\lambda x^A.f(fx) : (A \rightarrow A) \rightarrow A \rightarrow A \]  \hspace{1cm} (\rightarrow_I)
An example of typing derivation
The typing system satisfies the usual structural properties.

For instance, the weakening rule is admissible:

**Proposition**

If $\Gamma \vdash t : A$ is derivable then $\Gamma, \Delta \vdash t : A$ is also derivable, provided that $\text{dom}(\Delta) \cap \text{dom}(\Gamma) = \emptyset$. 
A term admits at most one type:

**Theorem**

If \( \Gamma \vdash t : A \) and \( \Gamma \vdash t : A' \) are derivable then \( A = A' \) (and the two proofs are the same!).
A term admits at most one type:

**Theorem**

*If* $\Gamma \vdash t : A$ *and* $\Gamma \vdash t : A'$ *are derivable then* $A = A'$ *(and the two proofs are the same!).*

**Proof.**

By induction on the term $t$. 

---

16
A term admits at most one type:

**Theorem**

If $\Gamma \vdash t : A$ and $\Gamma \vdash t : A'$ are derivable then $A = A'$ (and the two proofs are the same!).

**Proof.**

By induction on the term $t$.

We recall that the typing rules are:

$\Gamma \vdash x : \Gamma(x)$ (ax)  

$\frac{}{\Gamma, x : A \vdash t : B}$ ($\rightarrow_1$)  

$\frac{\Gamma \vdash \lambda x^A.t : A \rightarrow B}{\Gamma \vdash \text{lam}^x^A.t : \text{A} \rightarrow \text{B}}$ ($\rightarrow_E$)
Uniqueness of typing

A term admits at most one type:

**Theorem**
If $\Gamma \vdash t : A$ and $\Gamma \vdash t : A'$ are derivable then $A = A'$ (and the two proofs are the same!).

**Proof.**
By induction on the term $t$.

- If $t = x$ then the two derivations are necessarily

  \[
  \Gamma \vdash x : \Gamma(x) \quad \text{(ax)}
  \]

  and $A = A' = \Gamma(x)$
Uniqueness of typing

A term admits at most one type:

**Theorem**

If $\Gamma \vdash t : A$ and $\Gamma \vdash t : A'$ are derivable then $A = A'$ (and the two proofs are the same!).

**Proof.**

By induction on the term $t$.

- If $t = \lambda x^B.u$ then the two derivations are necessarily of the form

  \[
  \frac{\Gamma, x : B \vdash t : C}{\Gamma \vdash \lambda x^B.t : B \rightarrow C} \quad (\rightarrow_1)
  \]

  \[
  \frac{\Gamma, x : B \vdash t : C'}{\Gamma \vdash \lambda x^B.t : B \rightarrow C'} \quad (\rightarrow_1)
  \]

  by induction hypothesis we have $C = C'$

  and thus $A = B \rightarrow C = B \rightarrow C' = A'$. 

16
Uniqueness of typing

A term admits at most one type:

**Theorem**

If $\Gamma \vdash t : A$ and $\Gamma \vdash t : A'$ are derivable then $A = A'$ (and the two proofs are the same!).

**Proof.**

By induction on the term $t$.

- If $t = u \, v$ then the two derivations are necessarily of the form

\[
\frac{\Gamma \vdash t : B \rightarrow A \quad \Gamma \vdash u : B}{\Gamma \vdash t \, u : A} \quad (\rightarrow_E) \\
\frac{\Gamma \vdash t : B' \rightarrow A' \quad \Gamma \vdash u : B'}{\Gamma \vdash t \, u : A'} \quad (\rightarrow_E)
\]

by induction hypothesis we have $B \rightarrow A = B' \rightarrow A'$ and thus $A = A'$.

\[\square\]
The fact that we used Church style is important here!
Uniqueness of typing

The fact that we used Church style is important here!

In Curry style, this is a small variant:

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) \quad & (\text{ax}) \\
\Gamma, x : A \vdash t : B \quad & (\rightarrow_1) \\
\Gamma \vdash \lambda x. t : A \rightarrow B \\
\Gamma \vdash t \ u : B \\
\Gamma \vdash u : A \\
\Gamma \vdash t : A \rightarrow B \\
\Gamma \vdash t \ u : B \\
\end{align*}
\]
Uniqueness of typing

The fact that we used Church style is important here!

In Curry style, this is a small variant:

\[
\begin{array}{c}
\Gamma \vdash x : \Gamma(x) \quad (ax) \\
\Gamma, x : A \vdash t : B \quad (\to_i) \\
\Gamma \vdash \lambda x.t : A \to B \\
\Gamma \vdash t : A \to B \\
\Gamma \vdash u : A \\
\Gamma \vdash t u : B \quad (\to_E)
\end{array}
\]

but types are not unique anymore, e.g. \( \lambda x.x \) has types

\[
A \to A \\
(A \to B) \to (A \to B)
\]

etc.
In fact, we have more.

We observed earlier that one rule applies on a given term:

\[ \frac{\Gamma \vdash x : \Gamma(x)}{\Gamma \vdash x : \Gamma(x)} \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A. t : A \rightarrow B} \quad \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} \]

**Theorem**

*Given a derivable judgment \( \Gamma \vdash t : A \), there is exactly one way to derive it.*
Typing problems

If a term admits a type, there is only one and we can compute it!

As a consequence the three following problems are decidable: given $\Gamma$ and $t$,

- **type checking**: determine whether $\Gamma \vdash t : A$ is derivable,
- **typability**: determine whether there exists an $A$ such that $\Gamma \vdash t : A$ is derivable,
- **type inference**: construct an $A$ such that $\Gamma \vdash t : A$ is derivable.
Type inference and checking

(** Types. *)

type ty =
  | TVar of string
  | Arr of ty * ty

(** Terms. *)

type term =
  | Var of string
  | App of term * term
  | Abs of string * ty * term

(** Environments. *)

type context = (string * ty) list
exception Type_error

(** Type inference. *)

let rec infer env = function
    | Var x -> (try List.assoc x env with Not_found -> raise Type_error)
    | Abs (x, a, t) -> Arr (a, infer ((x,a)::env) t)
    | App (t, u) ->
        match infer env t with
        | Arr (a, b) -> if infer env u <> a then raise Type_error; b
        | _ -> raise Type_error

\[\Gamma \vdash x : \Gamma(x) \quad (ax)\]
\[\Gamma, x : A \vdash t : B \quad (\to_1)\]
\[\Gamma \vdash \lambda x^A.t : A \to B\]
\[\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A \quad (\to_E)\]
\[\Gamma \vdash tu : B\]
(** Type checking. *)
let check env t a =
    if infer env t <> a then raise Type_error

(** Typability. *)
let typable env t =
    try let _ = infer env t in true
    with Type_error -> false
Part III

The Curry-Howard correspondence
The Curry-Howard correspondence is the observation that

- a type is the same as a formula in the implicative fragment of logic:
  
  \[(A \rightarrow B) \rightarrow A \rightarrow B\]
  
  corresponds to
  
  \[(A \Rightarrow B) \Rightarrow A \Rightarrow B\]

- a typing derivation for simply typed \(\lambda\)-calculus is the same as a proof in NJ (implicative fragment).
The Curry-Howard correspondence

typing

\[
\Gamma, x : A, \Gamma' \vdash x : A \quad \text{(ax)}
\]

\[
\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \to B} \quad \text{($\to_I$)}
\]

\[
\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} \quad \text{($\to_E$)}
\]

logic

\[
\Gamma, A, \Gamma' \vdash A \quad \text{(ax)}
\]

\[
\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \text{($\Rightarrow_I$)}
\]

\[
\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \quad \text{($\Rightarrow_E$)}
\]
The Curry-Howard correspondence

The "term-erasing procedure" consists, starting from a typing derivation, in removing all the variables and terms (and replacing $\rightarrow$ by $\Rightarrow$):

\[
\frac{f : A \rightarrow B, x : A \vdash f : A \rightarrow B}{f : A \rightarrow B, x : A \vdash f : A \rightarrow B} \quad \text{(ax)}
\]

\[
\frac{f : A \rightarrow B, x : A \vdash x : A}{f : A \rightarrow B, x : A \vdash x : A} \quad \text{(ax)}
\]

\[
\frac{f : A \rightarrow B, x : A \vdash f x : B}{f : A \rightarrow B, x : A \vdash f x : B} \quad \text{($\rightarrow$E)}
\]

\[
\frac{f : A \rightarrow B, x : A \vdash \lambda x^A. f x : A \rightarrow B}{f : A \rightarrow B, x : A \vdash \lambda x^A. f x : A \rightarrow B} \quad \text{($\rightarrow$I)}
\]

\[
\vdash \lambda f^{A \rightarrow B}. \lambda x^A. f x : (A \rightarrow B) \rightarrow A \rightarrow B \quad \text{($\rightarrow$I)}
\]
The Curry-Howard correspondence

The “term-erasing procedure” consists, starting from a typing derivation, in removing all the variables and terms (and replacing $\rightarrow$ by $\Rightarrow$):

\[
\begin{align*}
A \Rightarrow B, & \quad A \vdash A \Rightarrow B & (\text{ax}) \\
A \Rightarrow B, & \quad A \vdash A & (\Rightarrow E) \\
A \Rightarrow B, & \quad A \vdash B & (\Rightarrow I) \\
A \Rightarrow B, & \quad A \vdash A \Rightarrow B & (\Rightarrow I) \\
\vdash (A \Rightarrow B) \Rightarrow A \Rightarrow B & (\Rightarrow I)
\end{align*}
\]
The Curry-Howard correspondence

The “term-erasing procedure” consists, starting from a typing derivation, in removing all the variables and terms (and replacing → by ⇒):

\[
\frac{A \Rightarrow B, \ A \vdash A \Rightarrow B}{A \Rightarrow B, \ A \vdash B} \quad (\text{ax})
\]

\[
\frac{A \Rightarrow B, \ A \vdash B}{\quad \vdash (A \Rightarrow B) \Rightarrow A \Rightarrow B} \quad (\Rightarrow \text{I})
\]

Lemma

Given a typing derivation, its term-erasure is a valid proof in NJ.

Proof.

Immediate induction.
Lemma
Conversely, given a proof $\pi$ of $A_1, \ldots, A_n \vdash A$ in NJ, we can construct a typing derivation of $x_1 : A_1, \ldots, x_n : A_n \vdash t : A$, for some term $t$, whose term-erasure is $\pi$.

Proof.
By induction on $\pi$. 
Lemma

Conversely, given a proof $\pi$ of $A_1, \ldots, A_n \vdash A$ in NJ, we can construct a typing derivation of $x_1 : A_1, \ldots, x_n : A_n \vdash t : A$, for some term $t$, whose term-erasure is $\pi$.

Proof.

It the last rule is

$$\Gamma, A, \Gamma' \vdash A \quad (ax)$$

then we construct

$$\Gamma, x : A, \Gamma' \vdash x : A \quad (ax)$$
The Curry-Howard correspondence

Lemma
Conversely, given a proof \( \pi \) of \( A_1, \ldots, A_n \vdash A \) in NJ, we can construct a typing derivation of \( x_1 : A_1, \ldots, x_n : A_n \vdash t : A \), for some term \( t \), whose term-erasure is \( \pi \).

Proof.

If the last rule is \( \pi \) \( \Gamma \vdash A \Rightarrow B \) \( \pi' \) \( \Gamma \vdash A \) \( (\Rightarrow_E) \)

then, by induction hypothesis, we have \( \vdots \) \( \Gamma \vdash t : A \rightarrow B \) \( \Gamma \vdash u : A \) \( \vdots \) \( \vdots \)

and we construct \( \vdots \) \( \Gamma \vdash t : A \rightarrow B \) \( \vdots \) \( \Gamma \vdash u : A \) \( \Gamma \vdash t u : B \) \( (\rightarrow_I) \).
Lemma
Conversely, given a proof \( \pi \) of \( A_1, \ldots, A_n \vdash A \) in NJ, we can construct a typing derivation of \( x_1 : A_1, \ldots, x_n : A_n \vdash t : A \), for some term \( t \), whose term-erasure is \( \pi \).

Proof.

\[
\frac{\pi}{\Gamma, A \vdash B} \quad (\Rightarrow_I)
\]

If the last rule is \( \Gamma \vdash A \Rightarrow B \), then by induction hypothesis we have

\[
\vdots
\]

\[
\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : A \vdash \lambda x^A.t : A \rightarrow B} \quad (\rightarrow_I).
\]

\( \square \)
The Curry-Howard correspondence

In the previous proof we did not have any choice for the terms (up to $\alpha$-conversion):

\[
\begin{align*}
A \Rightarrow B, & \quad A \vdash A \Rightarrow B & \text{(ax)} \quad & A \Rightarrow B, & \quad A \vdash A & \text{(ax)} \quad & (\Rightarrow E) \\
\hline
A \Rightarrow B, & \quad A \vdash B & \text{(}\Rightarrow I\text{)} \\
\hline
A \Rightarrow B \vdash & \quad A \Rightarrow B & \text{(}\Rightarrow I\text{)} \\
\hline
\vdash & (A \Rightarrow B) \Rightarrow A \Rightarrow B & \text{(}\Rightarrow I\text{)}
\end{align*}
\]
The Curry-Howard correspondence

In the previous proof we did not have any choice for the terms (up to $\alpha$-conversion):

\[
\begin{array}{c}
A \to B, \quad A \vdash A \to B \\
\hline
A \to B, \quad A \vdash A
\end{array}
\]

(ax)

\[
\begin{array}{c}
A \to B, \quad A \vdash B \\
\hline
A \to B \vdash A \to B
\end{array}
\]

(\to\text{E})

\[
\begin{array}{c}
A \vdash (A \to B) \to A \to B
\end{array}
\]

(\to\text{I})
In the previous proof we did not have any choice for the terms (up to $\alpha$-conversion):

\[
\begin{align*}
&f : A \to B, \quad A \vdash A \to B \quad (\text{ax}) \quad f : A \to B, \quad A \vdash A \quad (\text{ax}) \\
&\quad \quad \quad \quad f : A \to B, \quad A \vdash B \quad (\toE) \\
&\quad \quad \quad \quad f : A \to B \vdash A \to B \quad (\toI) \\
&\quad \vdash (A \to B) \to A \to B \quad (\toI)
\end{align*}
\]
In the previous proof we did not have any choice for the terms (up to $\alpha$-conversion):

\[
\begin{align*}
&f : A \to B, x : A \vdash A \to B \\
\frac{f : A \to B, x : A \vdash A}{f : A \to B, x : A \vdash B} &\text{ (ax)} \\
\frac{f : A \to B, x : A \vdash B}{f : A \to B \vdash A \to B} &\text{ (→E)} \\
\frac{f : A \to B \vdash A \to B}{\vdash (A \to B) \to A \to B} &\text{ (→I)}
\end{align*}
\]
The Curry-Howard correspondence

In the previous proof we did not have any choice for the terms (up to $\alpha$-conversion):

\[
\begin{align*}
\frac{f : A \to B, x : A \vdash f : A \to B}{f : A \to B, x : A \vdash f \cdot x : A \to B} \quad \text{(ax)} \\
\frac{f : A \to B, x : A \vdash A}{f : A \to B, x : A \vdash \lambda x \cdot A. f \cdot x : A \to B} \quad \text{(\to_E)} \\
\frac{f : A \to B, x : A \vdash B}{f : A \to B, x : A \vdash \lambda x \cdot A. f \cdot x : A \to B} \quad \text{(\to_I)} \\
\frac{f : A \to B, x : A \vdash A \to B}{\vdash (\lambda x \cdot A. f \cdot x : A \to B) : (A \to B) \to A \to B} \quad \text{(\to_I)}
\end{align*}
\]
The Curry-Howard correspondence

In the previous proof we did not have any choice for the terms (up to $\alpha$-conversion):

\[
\begin{align*}
&f : A \to B, x : A \vdash f : A \to B \\
&f : A \to B, x : A \vdash x : A \\
&f : A \to B, x : A \vdash B \\
&f : A \to B \vdash A \to B \\
&\vdash (A \to B) \to A \to B
\end{align*}
\]
The Curry-Howard correspondence

In the previous proof we did not have any choice for the terms (up to $\alpha$-conversion):

\[
\frac{f : A \rightarrow B, x : A \vdash f : A \rightarrow B}{f : A \rightarrow B, x : A \vdash fx : B} \quad (\rightarrow_\text{E})
\]

\[
\frac{f : A \rightarrow B, x : A \vdash f : A \rightarrow B}{f : A \rightarrow B, x : A \vdash x : A} \quad (\rightarrow_\text{I})
\]

\[
\frac{f : A \rightarrow B \vdash A \rightarrow B}{(A \rightarrow B) \rightarrow A \rightarrow B} \quad (\rightarrow_\text{I})
\]
The Curry-Howard correspondence

In the previous proof we did not have any choice for the terms (up to $\alpha$-conversion):

\[
\begin{array}{c}
\frac{f : A \to B, x : A \vdash f : A \to B}{f : A \to B, x : A \vdash f x : B} \quad (\to_E) \\
\frac{f : A \to B, x : A \vdash f x : B}{f : A \to B \vdash \lambda x^A. f x : A \to B} \quad (\to_I)
\end{array}
\]

\[
\frac{f : A \to B \vdash \lambda x^A. f x : A \to B}{\vdash (A \to B) \to A \to B} \quad (\to_I)
\]
In the previous proof we did not have any choice for the terms (up to $\alpha$-conversion):

\[
\begin{align*}
\frac{f : A \to B, x : A \vdash f : A \to B}{f : A \to B, x : A \vdash x : A} & \quad \text{(ax)} \\
\frac{f : A \to B, x : A \vdash f x : B}{f : A \to B \vdash \lambda x : A. f x : A \to B} & \quad \text{(\rightarrow E)} \\
\frac{f : A \to B \vdash \lambda x : A. f x : A \to B}{\vdash \lambda f : A \to B. \lambda x : A. f x : (A \to B) \to A \to B} & \quad \text{(\rightarrow I)}
\end{align*}
\]
The Curry-Howard correspondence

**Theorem**

*There is a bijection between*

- typable $\lambda$-terms (up to $\alpha$-conversion),
- typing derivations of $\lambda$-terms,
- proofs in the implicative fragment of NJ.
The Curry-Howard correspondence

**Theorem**

*There is a bijection between*

- typable $\lambda$-terms (up to $\alpha$-conversion),
- typing derivations of $\lambda$-terms,
- proofs in the implicative fragment of NJ.

*In other words,*

\[
\text{PROGRAM} = \text{PROOF}
\]
In particular, λ-terms can be considered as *proof witnesses*:

— *you*: Hey, the formula $A$ is true!
In particular, $\lambda$-terms can be considered as proof witnesses:

— you: Hey, the formula $A$ is true!
— me: Why should I believe you?
The Curry-Howard correspondence

In particular, λ-terms can be considered as *proof witnesses*:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>you</strong>: Hey, the formula $A$ is true!</td>
<td></td>
</tr>
<tr>
<td><strong>me</strong>: Why should I believe you?</td>
<td></td>
</tr>
<tr>
<td><strong>you</strong>: Here is a term $t$ witnessing for that.</td>
<td></td>
</tr>
</tbody>
</table>
The Curry-Howard correspondence

In particular, $\lambda$-terms can be considered as proof witnesses:

— *you*: Hey, the formula $A$ is true!
— *me*: Why should I believe you?
— *you*: Here is a term $t$ witnessing for that.
— *me*: Let me typecheck that...
In particular, λ-terms can be considered as proof witnesses:

— you: Hey, the formula $A$ is true!
— me: Why should I believe you?
— you: Here is a term $t$ witnessing for that.
— me: Let me typecheck that...
— me: Ok, now I believe you!
Part IV

Other connectives
Other connectives

The correspondence extends to other logical connectives too!

The general idea is that for each connective we can introduce in λ-calculus

- constructions to create a value of this type (= introduction rules)
- constructions to use a value of this type (= elimination rules)

with appropriate reduction rules (= cut elimination).
We extend the syntax of $\lambda$-terms with
We extend the syntax of $\lambda$-terms with

$$t, u ::= \ldots \mid \langle t, u \rangle \mid \pi_1(t) \mid \pi_r(t)$$

together with the reduction rules
We extend the syntax of $\lambda$-terms with

$$t, u ::= \ldots | \langle t, u \rangle | \pi_l(t) | \pi_r(t)$$

together with the reduction rules

$$\pi_l(\langle t, u \rangle) \longrightarrow t \quad \quad \quad \pi_r(\langle t, u \rangle) \longrightarrow u$$
(** Terms. *)

type term =
  | Var  of string
  | App  of term * term
  | Abs  of string * ty * term
(** Terms. *)

```haskell
type term =
  | Var of string
  | App of term * term
  | Abs of string * ty * term
  | Pair of term * term
  | PrjL of term
  | PrjR of term
```
Products

We extends the syntax of types with

\[ A, B ::= \ldots | A \times B \]

and add the typing rules
We extend the syntax of types with

\[ A, B ::= \ldots | A \times B \]

and add the typing rules

\[
\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B} \quad (\times_1)
\]

\[
\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_l(t) : A} \quad (\times^l_E) \quad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_r(t) : B} \quad (\times^r_E)
\]
We extend the syntax of types with

\[ A, B ::= \ldots \mid A \times B \]

and add the typing rules

\[
\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \land B} \quad (\land_I)
\]

\[
\frac{\Gamma \vdash A \land B}{\Gamma \vdash A} \quad (\land_E^I)
\]

\[
\frac{\Gamma \vdash A \land B}{\Gamma \vdash B} \quad (\land_E^E)
\]
We extend the syntax of \( \lambda \)-terms with

\[
t ::= \ldots \mid \langle \rangle
\]

(no reduction rule), extend the syntax of types with
We extend the syntax of $\lambda$-terms with

$$t ::= \ldots \mid \langle \rangle$$

(no reduction rule), extend the syntax of types with

$$A ::= \ldots \mid 1$$

and add the typing rule
Unit

We extend the syntax of $\lambda$-terms with

$$ t ::= \ldots \mid \langle \rangle $$

(no reduction rule), extend the syntax of types with

$$ A ::= \ldots \mid 1 $$

and add the typing rule

$$ \Gamma \vdash \langle \rangle : 1 _i $$
We extend the syntax of $\lambda$-terms with

$$t ::= \ldots \mid \langle \rangle$$

(no reduction rule), extend the syntax of types with

$$A ::= \ldots \mid 1$$

and add the typing rule

$$\Gamma \vdash \top$$
We now want to add coproducts types $A + B$, which corresponds to the formula $A \lor B$.

Recall that in OCaml the corresponding type is implemented with
We now want to add coproducts types $A + B$, which corresponds to the formula $A \lor B$.

Recall that in OCaml the corresponding type is implemented with

```ocaml
type ('a,'b) coprod =
  | Left of 'a
  | Right of 'b
```

and a typical program using those is of the form
Coproducts

We now want to add coproducts types $A + B$, which corresponds to the formula $A \lor B$.

Recall that in OCaml the corresponding type is implemented with

```
type ('a,'b) coprod =
  | Left  of 'a
  | Right of 'b
```

and a typical program using those is of the form

```
match t with
  | Left  x -> u
  | Right y -> v
```
If we do not want to use matching, we can program once for all the function

```ml
let case t u v =
    match t with
    | Left  x -> u x
    | Right y -> v y
```
We extend the syntax of $\lambda$-terms with
We extend the syntax of $\lambda$-terms with

$$t ::= \ldots \mid \iota_l(t) \mid \iota_r(u) \mid \text{case}(t, x \mapsto u, y \mapsto v)$$

together with the reduction rules
We extend the syntax of $\lambda$-terms with

$$t ::= \ldots \mid \iota_l(t) \mid \iota_r(u) \mid \text{case}(t, x \mapsto u, y \mapsto v)$$

together with the reduction rules

$$\text{case}(\iota_l(t), x \mapsto u, y \mapsto v) \rightarrow u[t/x]$$

$$\text{case}(\iota_r(t), x \mapsto u, y \mapsto v) \rightarrow v[t/y]$$
The reduction rules thus say that

```
match Left t with
| Left  x -> u
| Right y -> v
```

reduces to

```
u[t/x]
```

and similarly for `Right`.  

**Coproducts**
Coproducts

We extend the syntax of types with

$$A, B ::= \ldots \mid A + B$$

and add the typing rules
Coproducts

We extend the syntax of types with

\[ A, B ::= \ldots | A + B \]

and add the typing rules

\[ \Gamma \vdash t : A + B \quad \Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C \]

\[ \Gamma \vdash \text{case}(t, x \mapsto u, y \mapsto v) : C \] \hspace{1cm} (+E)

\[ \Gamma \vdash t : A \]

\[ \Gamma \vdash \iota_l(t) : A + B \] \hspace{1cm} (+I)

\[ \Gamma \vdash B \]

\[ \Gamma \vdash \iota_r(t) : A + B \] \hspace{1cm} (+I)
Coprodutcs

We extend the syntax of types with

\[ A, B ::= \ldots \mid A + B \]

and add the typing rules

\[
\begin{align*}
\Gamma \vdash A \lor B & \quad \Gamma, A \vdash C & \quad \Gamma, B \vdash C \\
\hline
\Gamma \vdash \Gamma \vdash C & \quad \Gamma \vdash C \quad \text{(}\lor_E\text{)}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash A & \quad \text{(}\lor_i\text{)} \\
\Gamma \vdash \Gamma \vdash A \lor B \\
\hline
\Gamma \vdash B & \quad \text{(}\lor_i\text{)} \\
\Gamma \vdash \Gamma \vdash A \lor B
\end{align*}
\]
Coproducts

Note that in OCaml, the type of our function

```ocaml
let case t u v =
  match t with
  | Left x -> u x
  | Right y -> v y

is
```

which can be read logically as

\[(A \lor B) \implies (A \implies C) \implies (B \implies C) \implies C\]
Note that in OCaml, the type of our function

```ocaml
let case t u v =
    match t with
    | Left x -> u x
    | Right y -> v y

is

('a, 'b) coprod -> ('a -> 'c) -> ('b -> 'c) -> 'c

which can be read logically as

\((A \lor B) \Rightarrow (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C\)
Coproducts

There is a slight problem: what’s wrong if we try to perform type inference?

\[
\frac{\Gamma \vdash t : A}{\Gamma \vdash \iota_l(t) : A + B} \quad (\lor_i)
\]

\[
\frac{\Gamma \vdash t : B}{\Gamma \vdash \iota_r(t) : A + B} \quad (\lor_i)
\]

For instance, what is the type of \(\iota_l(\lambda x A. x)\)?

It is like Church vs Curry, we need more typing information:

\[
\frac{\Gamma \vdash t : A}{\Gamma \vdash \iota_l(t) : A + B} \quad (\lor_i)
\]

\[
\frac{\Gamma \vdash t : B}{\Gamma \vdash \iota_r(t) : A + B} \quad (\lor_i)
\]
There is a slight problem: what’s wrong if we try to perform type inference?

\[
\frac{\Gamma \vdash t : A}{\Gamma \vdash \iota_1(t) : A + B} \quad (+_l) \quad \frac{\Gamma \vdash t : B}{\Gamma \vdash \iota_r(t) : A + B} \quad (+_r)
\]

For instance, what is the type of \( \iota_1(\lambda x^A.x) \)?
Coproducts

There is a slight problem: what’s wrong if we try to perform type inference?

\[
\frac{\Gamma \vdash t : A}{\Gamma \vdash \iota_l(t) : A + B} \quad \text{(}+_{l}\text{)} \quad \frac{\Gamma \vdash t : B}{\Gamma \vdash \iota_r(t) : A + B} \quad \text{(}+_{r}\text{)}
\]

For instance, what is the type of \(\iota_l(\lambda x^A.x)\)?

It is like Church vs Curry, we need more typing information:

\[
\frac{\Gamma \vdash t : A}{\Gamma \vdash \iota_l^B(t) : A + B} \quad \text{(}+_{l}\text{)} \quad \frac{\Gamma \vdash t : B}{\Gamma \vdash \iota_r^A(t) : A + B} \quad \text{(}+_{r}\text{)}
\]
Note that a term

\[
\text{case}(t, x \mapsto u, y \mapsto v)
\]

should be considered up to \(\alpha\)-conversion (we can rename \(x\) and \(y\)), which means extra care when implementing substitution.
Note that a term

\[
\text{\texttt{case}}(t, x \mapsto u, y \mapsto v)
\]

should be considered up to \(\alpha\)-conversion (we can rename \(x\) and \(y\), which means extra care when implementing substitution.

Instead, it is often easier to implement the variant with actual functions

\[
\text{\texttt{case}}(t, u, v)
\]
Coproducts

Note that a term
\[ \text{case}(t, x \mapsto u, y \mapsto v) \]
should be considered up to $\alpha$-conversion (we can rename $x$ and $y$), which means extra care when implementing substitution.

Instead, it is often easier to implement the variant with actual functions
\[ \text{case}(t, u, v) \]
which is typed as
\[
\Gamma \vdash t : A + B \quad \Gamma \vdash u : A \rightarrow C \quad \Gamma \vdash v : B \rightarrow C \\
\Gamma \vdash \text{case}(t, u, v) : C
\] (+$E$)
Coproducts

Note that a term

\[ \text{case}(t, x \mapsto u, y \mapsto v) \]

should be considered up to \(\alpha\)-conversion (we can rename \(x\) and \(y\)), which means extra care when implementing substitution.

Instead, it is often easier to implement the variant with actual functions

\[ \text{case}(t, u, v) \]

which is typed as

\[ \Gamma \vdash t : A + B \quad \Gamma \vdash u : A \rightarrow C \quad \Gamma \vdash v : B \rightarrow C \]

\[ \Gamma \vdash \text{case}(t, u, v) : C \]

instead of

\[ \Gamma \vdash t : A + B \quad \Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C \]

\[ \Gamma \vdash \text{case}(t, x \mapsto u, y \mapsto v) : C \]
We extend the syntax of $\lambda$-terms with

$$t ::= \ldots \mid \text{case}(t)$$

(no reduction rule), extend the syntax of types with

$$A ::= \ldots \mid 0$$

and add the typing rule

$$\Gamma \vdash t : 0 \quad \Gamma \vdash \text{case}(t) : A$$
Empty type

We extend the syntax of \( \lambda \)-terms with

\[
t ::= \ldots \mid \text{case}(t)
\]

(no reduction rule), extend the syntax of types with

\[
A ::= \ldots \mid 0
\]

and add the typing rule

\[
\frac{\Gamma \vdash t : 0}{\Gamma \vdash \text{case}(t) : A} \quad (0_\text{E})
\]
We extend the syntax of $\lambda$-terms with

\[ t ::= \ldots \mid \text{case}(t) \]

(no reduction rule), extend the syntax of types with

\[ A ::= \ldots \mid 0 \]

and add the typing rule

\[ \frac{\Gamma \vdash \bot}{\Gamma \vdash A} \quad (\bot_E) \]
If we add them all together, we want more reduction rules:

\[
\begin{align*}
\text{case}^{A\rightarrow B}(t) \ u & \longrightarrow \text{case}^B(t) \\
\pi_l(\text{case}^{A \times B}(t)) & \longrightarrow \text{case}^A(t) \\
\pi_r(\text{case}^{A \times B}(t)) & \longrightarrow \text{case}^B(t) \\
\text{case}(\text{case}^{A + B}(t), x \mapsto u, y \mapsto v) & \longrightarrow \text{case}^C(t) \\
\text{case}^A(\text{case}^0(t)) & \longrightarrow \text{case}^A(t) \\
\text{case}(t, x \mapsto u, y \mapsto v) \ w & \longrightarrow \text{case}(t, x \mapsto uw, y \mapsto vw) \\
\pi_l(\text{case}(t, x \mapsto u, y \mapsto v)) & \longrightarrow \text{case}(t, x \mapsto \pi_l(u), y \mapsto \pi_l(v)) \\
\pi_r(\text{case}(t, x \mapsto u, y \mapsto v)) & \longrightarrow \text{case}(t, x \mapsto \pi_r(u), y \mapsto \pi_r(v)) \\
\text{case}^C(\text{case}(t, x \mapsto u, y \mapsto v)) & \longrightarrow \text{case}(t, x \mapsto \text{case}^C(u), y \mapsto \text{case}^C(v)) \\
\text{case}(\text{case}(t, x \mapsto u, y \mapsto v), x' \mapsto u', y' \mapsto v') & \longrightarrow \text{case}(t, x \mapsto \text{case}(u, x' \mapsto u', y' \mapsto v'), y \mapsto \text{case}(v, x' \mapsto v', y' \mapsto v'))
\end{align*}
\]
In OCaml, natural numbers can be defined as

type nat =
  | Zero
  | Succ of nat

so that factorial can be implemented with
In OCaml, natural numbers can be defined as

```ocaml
type nat =
  | Zero
  | Succ of nat
```

so that factorial can be implemented with

```ocaml
let rec fact n =
  match n with
  | Zero    -> Succ Zero
  | Succ n  -> mult (Succ n) (fact n)
```
The “recurrence principle” / eliminator can then be defined as

```ml
let rec recursor n z s =
  match n with
  | Zero    -> z
  | Succ n  -> s n (recursor n z s)
```
of type

```ml
val recursor : nat -> 'a -> (nat -> 'a -> 'a) -> 'a = <fun>
```
The “recurrence principle” / eliminator can then be defined as

```ocaml
let rec recursor n z s =
  match n with
  | Zero       -> z
  | Succ n     -> s n (recursor n z s)
```
of type

```ocaml
val recursor : nat -> 'a -> (nat -> 'a -> 'a) -> 'a = <fun>
```

so that

```ocaml
let fact n =
  recursor n (Succ Zero) (fun n r -> mult (Succ n) r)
```
Natural numbers

We extend the syntax of $\lambda$-terms with

$$t ::= \ldots | Z | S(t) | \text{rec}(t, u, xy \mapsto v)$$

and add the reduction rules

$$\text{rec}(Z, z, xy \mapsto s) \rightarrow z$$
$$\text{rec}(S(n), z, xy \mapsto s) \rightarrow s[n/x, \text{rec}(n, z, xy \mapsto s)/y]$$
We extend the syntax of types with

\[ A, B ::= \ldots \mid \text{Nat} \]

and add the typing rules

\[
\begin{align*}
\Gamma \vdash Z & : \text{Nat} \\
\Gamma \vdash S(t) & : \text{Nat} \\
\Gamma \vdash n & : \text{Nat} \\
\Gamma \vdash z & : A \\
\Gamma, x & : \text{Nat}, y : A \vdash s : A \\
\Gamma \vdash \text{rec}(n, z, xy \mapsto s) & : A
\end{align*}
\]
Natural numbers

We extend the syntax of types with

\[ A, B ::= \ldots \mid \text{Nat} \]

and add the typing rules

\[
\begin{align*}
\Gamma &\vdash \text{Nat} \\
\Gamma &\vdash \text{Nat} \\
\Gamma &\vdash A \\
\Gamma, \text{Nat}, \quad A &\vdash A \\
\Gamma &\vdash A
\end{align*}
\]
Part V

Dynamics of Curry-Howard
Remember that a **cut** in a proof is an elimination rule whose principal premise is an introduction rule.

\[
\begin{align*}
\pi & \\
\Gamma \vdash A & \quad \pi' \quad \Gamma \vdash B \\
\hline
\Gamma \vdash A \land B & \quad (\land_I) \\
\hline
\Gamma \vdash A & \quad (\land_E)
\end{align*}
\]

\[
\begin{align*}
\pi & \\
\Gamma, A \vdash B & \quad \pi' \quad \Gamma \vdash A \\
\hline
\Gamma \vdash A \Rightarrow B & \quad (\Rightarrow_I) \\
\hline
\Gamma \vdash B & \quad (\Rightarrow_E)
\end{align*}
\]

Such a proof is intuitively doing “useless work” and we have seen that we could gradually remove all the cuts from a proof.
Cut elimination: conjunction

For instance, the cuts related to conjunction can be eliminated with

\[
\begin{align*}
\pi & \quad \pi' \\
\Gamma \vdash A & \quad \Gamma \vdash B \\
\hline
\Gamma \vdash A \land B & \quad (\land_I) \\
\hline
\Gamma \vdash A & \quad (\land_E) \\
\end{align*}
\]

and

\[
\begin{align*}
\pi & \quad \pi' \\
\Gamma \vdash A & \quad \Gamma \vdash B \\
\hline
\Gamma \vdash A \land B & \quad (\land_I) \\
\hline
\Gamma \vdash B & \quad (\land_E) \\
\end{align*}
\]

What is the computational content of this transformation?
For instance, the cuts related to conjunction can be eliminated with

\[
\begin{align*}
\pi & \quad \pi' \\
\Gamma \vdash A & \quad \Gamma \vdash B \\
\hline \\
\Gamma \vdash A \land B & \\
\hline \\
\Gamma \vdash A &
\end{align*}
\]

\((\land_I)\) \implies \pi

\[
\begin{align*}
\pi & \quad \pi' \\
\Gamma \vdash A & \quad \Gamma \vdash B \\
\hline \\
\Gamma \vdash A \land B & \\
\hline \\
\Gamma \vdash B &
\end{align*}
\]

\((\land_E)\) \implies \pi'

For instance, the cuts related to conjunction can be eliminated with

\[
\begin{align*}
\pi & \quad \pi' \\
\Gamma \vdash A & \quad \Gamma \vdash B \\
\hline \\
\Gamma \vdash A \land B & \quad (\land_I) \\
\hline \\
\Gamma \vdash A & \quad (\land_I) \\
\end{align*}
\]

\[\sim\]

\[
\begin{align*}
\pi & \quad \pi' \\
\Gamma \vdash A & \quad \Gamma \vdash B \\
\hline \\
\Gamma \vdash A \land B & \quad (\land_I) \\
\hline \\
\Gamma \vdash B & \quad (\land_E) \\
\end{align*}
\]

What is the computational contents of this transformation?
Cut elimination: conjunction

One of the cut-elimination rules is

\[
\begin{align*}
\pi & \quad \pi' \\
\Gamma \vdash A & \quad \Gamma \vdash B \\
\Gamma \vdash A \land B & \quad (\land_I) \\
\Gamma \vdash A & \quad (\land_E)
\end{align*}
\]

\[\sim\]

\[
\Gamma \vdash A
\]
One of the cut-elimination rules is

\[
\begin{array}{c}
\pi \\
\hline
\Gamma \vdash t : A
\end{array}
\quad
\begin{array}{c}
\pi'
\hline
\Gamma \vdash B
\end{array}
\quad
\begin{array}{c}
(\land_I)
\hline
\Gamma \vdash A \land B
\end{array}
\quad
\begin{array}{c}
(\land_E)
\hline
\Gamma \vdash A
\end{array}
\quad
\begin{array}{c}
\pi
\hline
\Gamma \vdash A
\end{array}
\]
One of the cut-elimination rules is

\[
\begin{array}{c}
\pi \\
\hline
\Gamma \vdash t : A \\
\Gamma \vdash u : B \\
\hline
\Gamma \vdash A \land B \\
\end{array}
\quad
\begin{array}{c}
\pi' \\
\hline
\Gamma \vdash (A \land B) \\
\end{array}
\quad
\begin{array}{c}
\land_I \\
\hline
\Gamma \vdash A \\
\end{array}
\quad
\begin{array}{c}
\land_E \\
\hline
\Gamma \vdash A \\
\end{array}
\]

In other words, it transforms a subterm \( \pi \Gamma \vdash \langle t, u \rangle \rightarrow_{\beta} t \) which is the reduction rule!
Cut elimination: conjunction

One of the cut-elimination rules is

\[
\begin{array}{c}
\pi \\
\hline
\Gamma \vdash t : A \\
\pi' \\
\Gamma \vdash u : B \\
\hline
\Gamma \vdash \langle t, u \rangle : A \land B \\
(\land_I) \\
\hline
\Gamma \vdash t : A \\
(\land_E) \\
\hline
\Gamma \vdash A
\end{array}
\]

In other words, it transforms a subterm \( \pi \Gamma \langle t, u \rangle \) which is the reduction rule!
One of the cut-elimination rules is

\[
\begin{array}{c}
\pi \\
\Gamma \vdash t : A \\
\pi' \\
\Gamma \vdash u : B \\
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash \langle t, u \rangle : A \land B \\
\langle \land_I \rangle \\
\Gamma \vdash \pi_1(\langle t, u \rangle) : A \\
\langle \land_E \rangle \\
\sim \sim \\
\Gamma \vdash A
\end{array}
\]

In other words, it transforms a subterm \( \pi_1(\langle t, u \rangle) \) which is the reduction rule!
One of the cut-elimination rules is

\[
\begin{align*}
\pi & \quad \pi' \\
\Gamma \vdash t : A & \quad \Gamma \vdash u : B \\
\Gamma \vdash \langle t, u \rangle : A \land B & \quad (\land_I) \\
\Gamma \vdash \pi_l(\langle t, u \rangle) : A & \quad (\land_E) \\
\Gamma \vdash t : A & \quad \rightsquigarrow \\
\end{align*}
\]

In other words, it transforms a subterm $\pi_l(\langle t, u \rangle) \rightarrow_{\beta} t$ which is the reduction rule!
Cut elimination: conjunction

One of the cut-elimination rules is

\[
\frac{\pi}{\Gamma \vdash t : A} \quad \frac{\pi'}{\Gamma \vdash u : B}
\]

\[
\Gamma \vdash \langle t, u \rangle : A \land B
\]

\[
\Gamma \vdash \pi_1(\langle t, u \rangle) : A
\]

\[
(\land_I)
\]

\[
(\land^l_E)
\]

\[
\pi
\]

\[
\Longrightarrow
\]

\[
\Gamma \vdash t : A
\]

In other words, it transforms a subterm

\[
\pi_1(\langle t, u \rangle) \rightarrow^\beta t
\]

which is the reduction rule!
The cut elimination rule for $\Rightarrow$ is

\[
\frac{
\frac{
\frac{
\pi
}{\Gamma, A \vdash B}
}{\Gamma \vdash A \Rightarrow B}
}{\Gamma \vdash B}
\quad
\frac{
\frac{
\pi'
}{\Gamma \vdash A}
}{\Gamma \vdash A \Rightarrow B}
\quad
\frac{
\pi'
}{\Gamma \vdash B}
\]

($\Rightarrow_1$) ($\Rightarrow_E$)
Cut elimination: implication

The cut elimination rule for $\Rightarrow$ is

$$
\begin{align*}
\frac{\pi}{\Gamma, A \vdash B} & \quad \frac{\pi'}{\Gamma \vdash A} \\
\Gamma \vdash A \Rightarrow B & \quad \Rightarrow \mathrm{I} & \quad \Rightarrow \mathrm{E} & \quad \Rightarrow \mathrm{E}
\end{align*}
$$

$\Rightarrow \mathrm{I}$

$$
\frac{\pi \pi'}{\Gamma \vdash B} \quad \frac{\pi'}{\Gamma \vdash A} \quad \Rightarrow \mathrm{E} \quad \Rightarrow \mathrm{E}
$$

where $\pi[\pi'/A]$ is $\pi$ where we have replaced all axioms on $A$

$$
\begin{align*}
\Gamma, A, \Gamma' \vdash A & \quad \text{by} \quad \frac{w(\pi')}{\Gamma, A, \Gamma' \vdash A} \\
\text{ax} & \quad w(\pi')
\end{align*}
$$

where $w(\pi')$ is an appropriate weakening of $\pi$. 

55
For instance, we can eliminate the cut

\[
\begin{array}{c}
\Gamma, A \vdash A \\
\hline
\Gamma \vdash A \Rightarrow A \\
\pi
\end{array}
\Rightarrow (\Rightarrow I)
\]

\[
\begin{array}{c}
\Gamma \vdash A \\
\hline
\Gamma \vdash A
\end{array}
\Rightarrow (\Rightarrow E) \quad \Rightarrow
\]

In other words,

\[(\lambda x \ A. x) t \rightarrow_{\beta} t\]
For instance, we can eliminate the cut

\[
\frac{
\frac{
\Delta, A \vdash A
}{\Delta \vdash A \Rightarrow A}
\quad
\pi
}{\pi
}\Rightarrow E}
\quad
\Rightarrow E
\end{array}\\Rightarrow E}
\quad
\Rightarrow E
\end{array}
\frac{
\frac{
\Delta, A \vdash A
}{\Delta \vdash A \Rightarrow A}
\quad
\pi
}{\pi
}\Rightarrow E}
\quad
\Rightarrow E
\end{array}\]
For instance, we can eliminate the cut

\[
\begin{array}{c}
\Gamma, x : A \vdash A \\
\hline
\Gamma, x : A \vdash A \\
\end{array}
\]

(ax)

\[
\begin{array}{c}
\Gamma \\
\hline
A \Rightarrow A \\
\end{array}
\]

(⇒₁)

\[
\begin{array}{c}
\Gamma \\
\hline
A \\
\end{array}
\]

(⇒₁)

\[
\begin{array}{c}
\Gamma \\
\hline
A \\
\end{array}
\]

(⇒₁)

\[
\begin{array}{c}
\Gamma \\
\hline
A \\
\end{array}
\]

(⇒₁)

\[
\begin{array}{c}
\Gamma \\
\hline
A \\
\end{array}
\]

(⇒₁)

\[
\begin{array}{c}
\Gamma \\
\hline
A \\
\end{array}
\]

(⇒₁)
For instance, we can eliminate the cut

\[
\frac{\Gamma, x : A \vdash x : A}{\Gamma \vdash \Lambda x : A (ax)} \quad \frac{\pi}{\Gamma \vdash A \Rightarrow A} \quad (\Rightarrow_1) \quad \frac{\Gamma \vdash A}{\Gamma \vdash \Lambda x : A. x : A} \quad (\Rightarrow_E) \quad \leadsto \quad \frac{\pi}{\Gamma \vdash A}
\]
For instance, we can eliminate the cut

\[
\begin{array}{c}
\frac{
\Gamma, x : A \vdash x : A
}{
\Gamma \vdash \lambda x^A. x : A \Rightarrow A
}(\Rightarrow_1)
\end{array}
\]

\[
\begin{array}{c}
\frac{
\pi
}{
\Gamma \vdash A
}(\Rightarrow_E)
\end{array}
\]

\[
\bowtie
\begin{array}{c}
\frac{
\pi
}{
\Gamma \vdash A
}
\end{array}
\]
For instance, we can eliminate the cut

\[
\begin{align*}
\Gamma, x : A & \vdash x : A \quad \text{(ax)} \\
\Gamma & \vdash \lambda x^A . x : A \Rightarrow A \quad \text{(⇒₁)} \\
\Gamma & \vdash t : A \\
\Gamma & \vdash A
\end{align*}
\]
For instance, we can eliminate the cut

\[ \frac{\Gamma, x : A \vdash x : A}{\Gamma \vdash \lambda x:A.x : A \Rightarrow A} \quad \text{(⇒1)} \]

\[ \frac{\pi}{\Gamma \vdash t : A} \]

\[ \frac{\Gamma \vdash (\lambda x:A.x)t : A}{\Gamma \vdash (\lambda x:A.x)t : A} \quad \text{(⇒E)} \]

\[ \sim \]

\[ \frac{\pi}{\Gamma \vdash A} \]
For instance, we can eliminate the cut

\[
\begin{align*}
\Gamma, x : A &\vdash x : A & (ax) \\
\Gamma &\vdash \lambda x^A.x : A \Rightarrow A & (\Rightarrow_1) \\
\Gamma &\vdash t : A & (\Rightarrow_\Pi) \\
\Gamma &\vdash (\lambda x^A.x)t : A & (\Rightarrow_\Pi) \\
\Gamma &\vdash t : A & (\Rightarrow_\Pi) \\
\end{align*}
\]

In other words, \((\lambda x^A.x)t \rightarrow_{\beta} t\).
For instance, we can eliminate the cut

\[
\frac{\Gamma, x : A \vdash x : A}{\Gamma \vdash \lambda x^A. x : A \Rightarrow A} \quad \text{(ax)} \quad \frac{\Gamma \vdash t : A}{\pi} \quad \frac{\pi}{\Gamma \vdash (\lambda x^A. x)t : A} \quad \text{(⇒E)} \quad \leadsto \quad \frac{\pi}{\Gamma \vdash t : A}
\]

In other words,

\[
(\lambda x^A. x)t \rightarrow_\beta t
\]
More generally, we have

\[
\frac{\pi}{\Gamma, x : A \vdash B} \quad \frac{\pi'}{\Gamma \vdash A} \quad (\Rightarrow_1) \quad \frac{\pi'}{\Gamma \vdash A} \quad (\Rightarrow_E) \quad \leadsto \quad \frac{\pi[\pi'/A]}{\Gamma \vdash B}
\]
More generally, we have

\[
\begin{array}{c}
\pi \\
\hline
\Gamma, x : A \vdash t : B
\end{array}
\quad \quad
\begin{array}{c}
\pi' \\
\hline
\Gamma \vdash A \Rightarrow B
\end{array}
\quad \quad
\begin{array}{c}
\hline
\pi'[\pi'/A]
\end{array}
\quad \quad
\begin{array}{c}
\hline
\Gamma \vdash B
\end{array}
\]
More generally, we have

\[
\begin{array}{c}
\pi \\
\Gamma, x : A \vdash t : B \\
\hline
\Gamma \vdash \lambda x^A.t : A \Rightarrow B \\
\end{array}
\]

\(\Rightarrow_i\)

\[
\begin{array}{c}
\pi' \\
\Gamma \vdash x : A \\
\hline
\Gamma \vdash A \\
\end{array}
\]

\(\Rightarrow_E\)

\[
\begin{array}{c}
\Gamma \vdash B \\
\hline
\Gamma \vdash B \\
\end{array}
\]

\(~\sim\) \[
\begin{array}{c}
\pi[\pi'/A] \\
\hline
\Gamma \vdash B \\
\end{array}
\]
More generally, we have

\[
\frac{\pi}{\Gamma, x : A \vdash t : B} \quad \frac{\pi'}{\Gamma \vdash u : A} \quad (\Rightarrow_i) \quad \frac{\Gamma \vdash \lambda x^A. t : A \Rightarrow B}{\Gamma \vdash B} \quad (\Rightarrow_E) \quad \pi[\pi'/A] \quad \frac{\Gamma \vdash B}{B}
\]
More generally, we have

\[
\begin{align*}
\pi \\
\Gamma, x : A \vdash t : B \quad \frac{\pi'}{\Gamma \vdash \lambda x^A t : A \Rightarrow B} \quad \frac{\pi'}{\Gamma \vdash u : A} \quad \frac{\Rightarrow E}{\Gamma \vdash (\lambda x^A t) u : B} \\
\Gamma \vdash (\lambda x^A t) u : B \\
\end{align*}
\]

In other words, \((\lambda x^A t) u \rightarrow^\beta t[u/x]\)
More generally, we have

\[
\begin{array}{c}
\pi' \\
\infer[(\Rightarrow_i)]{\Gamma \vdash \lambda x : A. t : A \Rightarrow B}{\Gamma, x : A \vdash t : B} \\
\pi' \quad \Gamma \vdash u : A & \quad \Gamma \vdash (\lambda x : A. t) u : B \\
\end{array} (\Rightarrow_E) \\
\Rightarrow \quad \Gamma \vdash t[u/x] : B
\]
More generally, we have

\[
\begin{array}{c}
\pi \\
\Gamma, x : A \vdash t : B \\
\pi' \\
\Gamma \vdash \lambda x^A \cdot t : A \Rightarrow B \\
\Gamma \vdash u : A \\
\Gamma \vdash (\lambda x^A \cdot t) u : B \\
\Gamma \vdash t[u/x] : B
\end{array}
\]

In other words,

\[
(\lambda x^A \cdot t) u \quad \rightarrow_\beta \quad t[u/x]
\]
Suppose given a term $t$ of type $A$ in a context $\Gamma$, its typing derivation

$$\frac{\pi}{\Gamma \vdash t : A}$$

can be seen as a proof in NJ. We have shown that
Suppose given a term $t$ of type $A$ in a context $\Gamma$, its typing derivation

$$
\frac{\pi}{\Gamma \vdash t : A}
$$

can be seen as a proof in NJ. We have shown that

**Theorem**

*The cut elimination steps of $\pi$ are in correspondence with the $\beta$-reduction steps of $t$.***

This means that for every $\beta$-reduction $t \rightarrow_{\beta} t'$ there is a derivation $\pi'$ of $\Gamma \vdash t' : A$

$$
\frac{\pi}{\Gamma \vdash t : A} \rightsquigarrow \frac{\pi'}{\Gamma \vdash t' : A}
$$

which is obtained by a cut elimination step from $\pi$, and conversely every cut-elimination step from $\pi$ is of this form.
In particular, we have showed the **subject reduction** property: typing is compatible with $\beta$-reduction.

**Theorem**

If $\Gamma \vdash t : A$ is derivable and $t \rightarrow^\beta t'$ then $\Gamma \vdash t' : A$ is also derivable.
Subject reduction

In particular, we have show the **subject reduction** property: typing is compatible with $\beta$-reduction.

**Theorem**

*If $\Gamma \vdash t : A$ is derivable and $t \xrightarrow{\beta} t'$ then $\Gamma \vdash t' : A$ is also derivable.*

For instance,

$$
\frac{
\frac{
\Gamma, x : A \vdash x : A 
\quad \text{(ax)}

\quad \pi 
}{
\Gamma \vdash \lambda x^A.x : A \rightarrow A 
\quad \text{($\rightarrow I$)}

\quad \Gamma \vdash t : B
}{
\Gamma \vdash (\lambda x^A.x) t : B 
\quad \text{($\rightarrow E$)}
\quad \leadsto

\frac{
\pi 
}{
\Gamma \vdash t : B
} 
}
$$
The Curry-Howard correspondence

We can add a third level to the correspondence:

**Theorem**

*There is a bijection between*

1. *types and formulas,*
2. *λ-terms of type A and proofs of A in NJ,*
The Curry-Howard correspondence

We can add a third level to the correspondence:

**Theorem**
*There is a bijection between*

1. *types and formulas,*
2. *λ-terms of type A and proofs of A in NJ,*
3. *reduction steps and cut elimination steps.*
We can add a third level to the correspondence:

**Theorem**

*There is a bijection between*

1. *types and formulas,*
2. \( \lambda \)-*terms of type \( A \) and proofs of \( A \) in \( NJ \),
3. *reduction steps and cut elimination steps.*

Using a function in programming is the same as using a lemma in mathematics!
A variant of cuts

A cut is an elimination of an introduction of some connective. What if we try to do the converse?

For implication,

$$\pi$$

$$\Gamma \vdash A \Rightarrow B$$

$$\Gamma, A \vdash A \Rightarrow B \quad \text{(wk)} \quad \Gamma, A \vdash A \quad \text{(ax)}$$

$$\Gamma, A \vdash B \quad \text{(⇒E)}$$

$$\Gamma \vdash A \Rightarrow B \quad \text{(⇒I)}$$

$$\Rightarrow$$
A variant of cuts

A cut is an elimination of an introduction of some connective.
What if we try to do the converse?

For implication,

\[
\begin{align*}
    \pi & \quad \frac{\Gamma \vdash A \Rightarrow B}{\Gamma, A \vdash A \Rightarrow B} \quad \text{(wk)} \\
    & \quad \frac{\Gamma, A \vdash A \Rightarrow B}{\Gamma, A \vdash A} \quad \text{(ax)} \\
    & \quad \frac{\Gamma, A \vdash B}{\Gamma, A \vdash A \Rightarrow B} \quad \text{(\Rightarrow E)} \\
    & \quad \frac{\Gamma \vdash A \Rightarrow B}{\Gamma \vdash A \Rightarrow B} \quad \text{(\Rightarrow I)} \\
\end{align*}
\]

\[\Rightarrow \quad \pi \quad \frac{\Gamma \vdash A \Rightarrow B}{\Gamma \vdash A \Rightarrow B}\]

In other words, we recover \(\eta\)-reduction:
\[\lambda x. A \ x \rightarrow_{\eta} t\]
A variant of cuts

A cut is an elimination of an introduction of some connective.
What if we try to do the converse?

For implication,

\[
\begin{align*}
\Gamma \vdash A \Rightarrow B \\
\Gamma, x : A \vdash A \Rightarrow B & \quad \text{(wk)} \\
\Gamma, x : A \vdash A & \quad \text{(ax)} \\
\Gamma, x : A \vdash B & \quad \text{(⇒E)} \\
\Gamma \vdash A \Rightarrow B & \quad \text{(⇒I)} \\
\end{align*}
\]

\[\leadsto\]

\[
\Gamma \vdash A \Rightarrow B
\]

In other words, we recover \( \eta \)-reduction:

\[
\lambda x \ A . tx \rightarrow_{\eta} t
\]
A variant of cuts

A cut is an elimination of an introduction of some connective.
What if we try to do the converse?

For implication,

\[
\begin{align*}
\pi & \quad \frac{t : A \Rightarrow B}{\Gamma \vdash t : A \Rightarrow B} \\
\text{wk} & \quad \frac{\pi}{\frac{A \Rightarrow B}{\Gamma, x : A \vdash A \Rightarrow B}} \\
\text{ax} & \quad \frac{A}{\Gamma, x : A \vdash A} \\
\Rightarrow_e & \quad \frac{B}{\Gamma, x : A \vdash B} \\
\Rightarrow_i & \quad \frac{A \Rightarrow B}{\Gamma \vdash A \Rightarrow B} \\
\sim & \quad \frac{\pi}{\Gamma \vdash A \Rightarrow B}
\end{align*}
\]

In other words, we recover η-reduction:

\[
\lambda x : A. tx \rightarrow_{\eta} \pi \frac{t : A \Rightarrow B}{\Gamma \vdash t : A \Rightarrow B}
\]
A variant of cuts

A cut is an elimination of an introduction of some connective. What if we try to do the converse?

For implication,

\[
\begin{array}{c}
\pi \\
\hline
\Gamma \vdash t : A \Rightarrow B \\
\hline
\Gamma, x : A \vdash t : A \Rightarrow B \quad (\text{wk})
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x : A \vdash A \quad (\text{ax})
\hline
\Gamma, x : A \vdash B \quad (\Rightarrow \text{E})
\hline
\Gamma \vdash A \Rightarrow B \quad (\Rightarrow \text{I})
\end{array}
\]

\[
\Gamma \vdash A \Rightarrow B
\]

In other words, we recover \( \eta \)-reduction:

\[
\Lambda x : A. tx \rightarrow \eta t
\]
A variant of cuts

A cut is an elimination of an introduction of some connective. What if we try to do the converse?

For implication,

\[
\begin{align*}
\pi \\
\Gamma \vdash t : A \Rightarrow B \\
\Gamma, x : A \vdash t : A \Rightarrow B & \quad \text{(wk)} \\
\Gamma, x : A \vdash x : A & \quad \text{(ax)} \\
\Gamma, x : A \vdash B & \quad \text{(\Rightarrow E)} \\
\Gamma \vdash A \Rightarrow B & \quad \text{(\Rightarrow I)} \\
\end{align*}
\]

\[\sim \]

\[
\pi \\
\Gamma \vdash A \Rightarrow B
\]
A variant of cuts

A cut is an elimination of an introduction of some connective. What if we try to do the converse?

For implication,

\[
\begin{align*}
\pi \\
\Gamma \vdash t : A \Rightarrow B \\
\Gamma, x : A \vdash t : A \Rightarrow B \quad \text{(wk)} \\
\Gamma, x : A \vdash x : A \quad \text{(ax)} \\
\Gamma, x : A \vdash tx : B \quad \text{($\Rightarrow_E$)} \\
\Gamma \vdash A \Rightarrow B \quad \text{($\Rightarrow_I$)} \\
\end{align*}
\]

\[
\Gamma \vdash \lambda x : A. tx : A \Rightarrow B \quad \text{($\Rightarrow_I$)}
\]

\[
\Gamma \vdash A \Rightarrow B
\]

\[
\Rightarrow \quad \Gamma \vdash A \Rightarrow B
\]
A variant of cuts

A cut is an elimination of an introduction of some connective. What if we try to do the converse?

For implication,

\[
\begin{align*}
\pi & \quad \Gamma \vdash t : A \Rightarrow B \\
\hline
\Gamma, x : A & \vdash t : A \Rightarrow B \quad \text{(wk)} \\
\hline
\Gamma, x : A & \vdash x : A \quad \text{(ax)} \\
\hline
\Gamma, x : A & \vdash tx : B \quad \text{(\Rightarrow E)} \\
\hline
\Gamma & \vdash \lambda x^A . tx : A \Rightarrow B \quad \text{(\Rightarrow I)}
\end{align*}
\]

\[\sim\]

\[
\Gamma \vdash A \Rightarrow B
\]
A variant of cuts

A cut is an elimination of an introduction of some connective.

What if we try to do the converse?

For implication,

\[
\begin{align*}
\pi & \quad \Gamma \vdash t : A \Rightarrow B \\
\Gamma, x : A \vdash t : A \Rightarrow B & \quad \text{(wk)} \quad \Gamma, x : A \vdash x : A \\
\Gamma, x : A \vdash t x : B & \quad \text{(⇒E)} \\
\Gamma \vdash \lambda x^A . t x : A \Rightarrow B & \quad \text{(⇒I)} \\
\Gamma \vdash t : A \Rightarrow B & \quad \text{⇝} \\
\end{align*}
\]

In other words, we recover η-reduction:

\[
\lambda x^A . t x \rightarrow^\eta \pi t
\]
A variant of cuts

A cut is an elimination of an introduction of some connective. What if we try to do the converse?

For implication,

\[
\frac{\pi}{\Gamma \vdash t : A \Rightarrow B} \quad (\text{wk})
\]

\[
\frac{\Gamma, x : A \vdash t : A \Rightarrow B}{\Gamma, x : A \vdash x : A} \quad (\text{ax})
\]

\[
\frac{\Gamma, x : A \vdash x : A}{\Gamma, x : A \vdash tx : B} \quad (\Rightarrow \text{E})
\]

\[
\frac{\Gamma \vdash \lambda x^A. tx : A \Rightarrow B}{\Gamma \vdash t : A \Rightarrow B} \quad (\Rightarrow \text{I})
\]

In other words, we recover \( \eta \)-\textit{reduction}:

\[
\lambda x^A. tx \quad \longrightarrow_\eta \quad t
\]
A variant of cuts

This also works for other connectives:

\[
\pi \\
\frac{\Gamma \vdash A \land B}{\Gamma \vdash A} \quad \text{(\land^I_E)} \quad \frac{\pi}{\land^E} \\
\frac{\pi}{\Gamma \vdash B} \quad \frac{\Gamma \vdash A \land B}{\land^I} \quad \rightsquigarrow \\
\frac{\pi}{\Gamma \vdash A \land B}
\]

In other words, the \(\eta\)-reduction rule for products is

\[
\langle \pi_l(t), \pi_r(t) \rangle \rightarrow_{\eta} t
\]
A variant of cuts

This also works for other connectives:

\[
\begin{array}{c}
\frac{\pi}{\Gamma \vdash t : A \land B} \quad \frac{\pi}{\Gamma \vdash t : A \land B} \\
\hline
\frac{\Gamma \vdash t : A \land B}{\Gamma \vdash A} \quad \frac{\Gamma \vdash t : A \land B}{\Gamma \vdash B}
\end{array}
\]

\(\land_1\) \hspace{2cm} \(\land_E\)

\[
\begin{array}{c}
\frac{}{\Gamma \vdash A \land B}
\end{array}
\]

\(\land_1\) \hspace{2cm} \(\land_E\)

\[
\begin{array}{c}
\frac{}{\Gamma \vdash A \land B}
\end{array}
\]

\[\rightsquigarrow\]

\[
\begin{array}{c}
\frac{}{\Gamma \vdash A \land B}
\end{array}
\]
A variant of cuts

This also works for other connectives:

\[
\begin{align*}
\frac{\pi}{\Gamma \vdash t : A \land B} & \quad (\land_1^E) \\
& \quad \\
\frac{\pi}{\Gamma \vdash \pi_l(t) : A} & \quad (\land_1^E) \\
& \quad \\
\frac{\pi}{\Gamma \vdash \pi_r(t) : B} & \quad (\land_1) \\
& \quad \\
\frac{\pi}{\Gamma \vdash A \land B} & \quad (\land_1) \\
& \quad \\
\end{align*}
\]

In other words, the \(\eta\)-reduction rule for products is

\[
\langle \pi_l(t), \pi_r(t) \rangle \rightarrow_{\eta} t
\]

\(\rightarrow_\eta\)
A variant of cuts

This also works for other connectives:

\[
\begin{align*}
\pi \\
\Gamma \vdash t : A \land B \quad & \quad (\land_1^E) \\
\Gamma \vdash \pi_l(t) : A \\
\hline \\
\Gamma \vdash \pi_l(t) : A \\
\end{align*}
\]

\[
\begin{align*}
\pi \\
\Gamma \vdash t : A \land B \\
\hline \\
\Gamma \vdash \pi_r(t) : B \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t : A \land B \quad & \quad (\land_1^E) \\
\Gamma \vdash \pi_r(t) : B \\
\hline \\
\Gamma \vdash \pi_r(t) : B \\
\end{align*}
\]

\[
\begin{align*}
\hline \\
\Gamma \vdash A \land B \\
\end{align*}
\]

\[
\begin{align*}
\hline \\
\pi \quad & \quad \sim \quad & \quad \pi \\
\Gamma \vdash A \land B \\
\end{align*}
\]
A variant of cuts

This also works for other connectives:

\[
\begin{align*}
&\frac{\pi}{\Gamma \vdash t : A \land B} \quad (\land_1^\Gamma) \\
&\frac{\Gamma \vdash \pi_l(t) : A}{\Gamma \vdash \pi_l(t) : A} \quad (\land_\pi^1) \\
&\frac{\Gamma \vdash \langle \pi_l(t), \pi_r(t) \rangle : A \land B}{\Gamma \vdash \langle \pi_l(t), \pi_r(t) \rangle : A \land B} \quad (\land_I) \\
&\Gamma \vdash \pi \\
\end{align*}
\]
A variant of cuts

This also works for other connectives:

\[
\frac{\pi}{\Gamma \vdash t : A \land B} \quad (\land_E) \quad \frac{\pi}{\Gamma \vdash t : A \land B} \quad (\land_E)
\]

\[
\frac{\Gamma \vdash \pi_l(t) : A}{\Gamma \vdash \langle \pi_l(t), \pi_r(t) \rangle : A \land B} \quad (\land_I) \quad \frac{\pi}{\Gamma \vdash t : A \land B}
\]

In other words, the \(\eta\)-reduction rule for products is

\[
\langle \pi_l(t), \pi_r(t) \rangle \xrightarrow{\eta} t
\]
A variant of cuts

This also works for other connectives:

\[
\begin{array}{c}
\pi \\
\hline
\Gamma \vdash t : A \land B \\
\hline
\Gamma \vdash \pi_l(t) : A \\
\hline
\end{array}
\quad
\begin{array}{c}
\pi \\
\hline
\Gamma \vdash t : A \land B \\
\hline
\Gamma \vdash \pi_r(t) : B \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\bigwedge_1
\end{array}
\quad
\begin{array}{c}
\bigwedge_1
\end{array}
\]

\[
\Gamma \vdash \langle \pi_l(t), \pi_r(t) \rangle : A \land B \\
\hline
\bigwedge_1
\]

In other words, the \(\eta\)-reduction rule for products is

\[
\langle \pi_l(t), \pi_r(t) \rangle \quad \xrightarrow{\eta} \quad t
\]
Part VI

Classical logic
For a long time, people thought that this correspondence could not be extended to classical logic.

It turns out that it actually does (Parigot’s $\lambda\mu$-calculus): classical logic is constructive!

This gives rise to strange languages, relying heavily on a variant of exceptions.
We have seen that classical logic could be obtained from intuitionistic one by adding the principle of elimination of double negation:

$$\neg\neg A \Rightarrow A$$
We have seen that classical logic could be obtained from intuitionistic one by adding the principle of elimination of double negation:

\[ \neg\neg A \Rightarrow A \]

This can be equivalently implemented by adding the rule

\[
\frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A} \quad (\neg\neg E)
\]
We have seen that classical logic could be obtained from intuitionistic one by adding the principle of elimination of double negation:

\[ \neg\neg A \Rightarrow A \]

This can be equivalently implemented by adding the rule:

\[
\frac{\Gamma \vdash t : \neg\neg A}{\Gamma \vdash C(t) : A} (\neg\neg E)
\]

This suggests that we should add a new construction \( C \).
Classical logic

Remembering that

\[ \neg A = A \Rightarrow \bot \]

we already have an introduction rule for double negation:

\[
\pi \\
\frac{}{\Gamma \vdash A} \quad \text{(ax)}
\]

\[
\frac{}{\Gamma \vdash A \Rightarrow \bot} \quad \frac{}{A} \quad \text{(wk)}
\]

\[
\frac{}{\Gamma \vdash \bot} \quad \text{(}\Rightarrow \text{E)}
\]

\[
\frac{}{\Gamma \vdash \bot \Rightarrow \bot} \quad \text{(}\Rightarrow \text{I)}
\]

\[
\frac{}{\Gamma \vdash (A \Rightarrow \bot) \Rightarrow \bot}
\]
Remembering that

\[ \neg A = A \Rightarrow \bot \]

we already have an introduction rule for double negation:

\[ \frac{\pi}{\Gamma \vdash t : A} \quad \frac{\Gamma \vDash t : A}{\Gamma, k : A \Rightarrow \bot \vdash A} \quad \text{(wk)} \]

\[ \frac{\Gamma, k : A \Rightarrow \bot \vdash A}{\Gamma, k : A \Rightarrow \bot \vdash \bot} \quad \text{(⇒E)} \]

\[ \frac{\Gamma, k : A \Rightarrow \bot \vdash \bot}{\Gamma \vdash (A \Rightarrow \bot) \Rightarrow \bot} \quad \text{(⇒I)} \]
Remembering that
\[ \neg A = A \Rightarrow \bot \]

we already have an introduction rule for double negation:
Remembering that

\[ \neg A = A \Rightarrow \bot \]

we already have an introduction rule for double negation:

\[ \frac{\Gamma, k : A \Rightarrow \bot \vdash k : A \Rightarrow \bot}{\Gamma \vdash t : A \Rightarrow \bot} \quad \text{(wk)} \]

\[ \frac{\Gamma, k : A \Rightarrow \bot \vdash t : A}{\Gamma, k : A \Rightarrow \bot \vdash \bot} \quad \text{(⇒E)} \]

\[ \frac{\Gamma, k : A \Rightarrow \bot \vdash \bot}{\Gamma \vdash (A \Rightarrow \bot) \Rightarrow \bot} \quad \text{(⇒I)} \]
Classical logic

Remembering that
\[ \neg A = A \Rightarrow \bot \]
we already have an introduction rule for double negation:

\[
\begin{align*}
\Gamma, k : A \Rightarrow \bot & \vdash k : A \Rightarrow \bot & \text{(ax)} \\
\Gamma, k : A \Rightarrow \bot & \vdash t : A & \text{(wk)} \\
\Gamma, k : A \Rightarrow \bot & \vdash k \ t : \bot & \text{($\Rightarrow E$)} \\
\Gamma & \vdash (A \Rightarrow \bot) \Rightarrow \bot & \text{($\Rightarrow I$)}
\end{align*}
\]
Remembering that

$$\neg A = A \Rightarrow \bot$$

we already have an introduction rule for double negation:

$$\Gamma, k : A \Rightarrow \bot \vdash \pi t : A$$ (wk)

$$\Gamma, k : A \Rightarrow \bot \vdash \Gamma, k : A \Rightarrow \bot \vdash k t : \bot$$ (⇒E)

$$\Gamma \vdash \lambda k^{A \Rightarrow \bot}. k t : (A \Rightarrow \bot) \Rightarrow \bot$$ (⇒I)
The cut elimination procedure should give

\[ \pi \]
\[ \Gamma \vdash A \]
\[ \Gamma \vdash \neg \neg A \text{ (\neg \neg I)} \]
\[ \Gamma \vdash A \text{ (\neg \neg E)} \]
\[ \Gamma \vdash A \]

\[ \Rightarrow \]

\[ \Gamma \vdash A \]
The cut elimination procedure should give

\[
\begin{align*}
\pi & \\
\Gamma \vdash t : A & \\
\Gamma \vdash \neg\neg A & (\neg\neg I) \\
\Gamma \vdash A & (\neg\neg E) \\
\Rightarrow & \\
\Gamma \vdash t : A & \\
\end{align*}
\]
The cut elimination procedure should give

\[\frac{\pi}{\Gamma \vdash t : A} \quad \frac{\Gamma \vdash \lambda k \neg A. k t : \neg \neg A}{\Gamma \vdash A} \quad (\neg \neg E) \quad \Rightarrow \quad \frac{\pi}{\Gamma \vdash t : A}\]
The cut elimination procedure should give

\[
\pi \\
\frac{\Gamma \vdash t : A}{\Gamma \vdash \lambda k \neg A. k t : \neg \neg A} \quad (\neg \neg I) \\
\frac{\Gamma \vdash \lambda k \neg A. k t : \neg \neg A}{\Gamma \vdash C(\lambda k \neg A. k t) : A} \quad (\neg \neg E) \\
\leadsto \\
\frac{\pi}{\Gamma \vdash t : A}
\]
The cut elimination procedure should give

\[
\begin{align*}
\pi \\
\frac{\Gamma \vdash t : A}{\Gamma \vdash \lambda \neg A. k t : \neg \neg A} \quad (\neg \neg I) \\
\frac{\Gamma \vdash \lambda \neg A. k t : \neg \neg A}{\Gamma \vdash C(\lambda \neg A. k t) : A} \quad (\neg \neg E) \\
\end{align*}
\]

\[\leadsto \quad \pi \quad \frac{\Gamma \vdash C(\lambda \neg A. k t) : A}{\Gamma \vdash t : A} \]

In other words,

\[C(\lambda \neg A. k t) \rightarrow_{\beta} t\]
The reduction rule is

\[ C(\lambda k^{\neg A}.k\, t) \rightarrow_{\beta} t \]
The reduction rule is

$$C(\lambda k \neg A . k \ t) \rightarrow^\beta t$$

When we apply this function $k$ to some argument $t$ the function $C$ will discard the computation and return the argument $t$. 
The reduction rule is

\[ C(\lambda k\neg A \cdot k \; t) \rightarrow^\beta t \]

When we apply this function \( k \) to some argument \( t \) the function \( C \) will discard the computation and return the argument \( t \). It only makes sense when \( k \notin \text{FV}(t) \).
Classical logic: reduction

The reduction rule is

\[ C(\lambda k \neg A . k \ t) \rightarrow_{\beta} t \]

When we apply this function \( k \) to some argument \( t \) the function \( C \) will discard the computation and return the argument \( t \). It only makes sense when \( k \notin \text{FV}(t) \).

Generally, reduction looks like this:

\[ C(\lambda k \neg A . u) \rightarrow_{\beta} C(\lambda k \neg A . u_1) \rightarrow_{\beta} C(\lambda k \neg A . u_2) \rightarrow_{\beta} \ldots \rightarrow_{\beta} C(\lambda k \neg A . k \ t) \rightarrow_{\beta} t \]
The reduction rule is

\[ C(\lambda k \neg A. k \ t) \rightarrow_\beta t \]

When we apply this function \( k \) to some argument \( t \) the function \( C \) will discard the computation and return the argument \( t \). It only makes sense when \( k \notin \text{FV}(t) \).

Generally, reduction looks like this:

\[ C(\lambda k \neg A. u) \rightarrow_\beta C(\lambda k \neg A. u_1) \rightarrow_\beta C(\lambda k \neg A. u_2) \rightarrow_\beta \ldots \rightarrow_\beta C(\lambda k \neg A. k \ t) \rightarrow_\beta t \]

We can read

\[ C(\ldots) = \text{try } \ldots \text{ catch } x \rightarrow x \quad k = \text{raise} \]
The reduction rule is
\[ C(\lambda k \neg A. k \, t) \rightarrow_\beta t \]

When we apply this function \( k \) to some argument \( t \) the function \( C \) will discard the computation and return the argument \( t \). It only makes sense when \( k \not\in \text{FV}(t) \).

Generally, reduction looks like this:
\[
C(\lambda k \neg A. u) \rightarrow_\beta C(\lambda k \neg A. u_1) \rightarrow_\beta C(\lambda k \neg A. u_2) \rightarrow_\beta \ldots \rightarrow_\beta C(\lambda k \neg A. k \, t) \rightarrow_\beta t
\]

We can read
\[
C(\ldots ) = \text{try} \ldots \text{catch } x \rightarrow x \quad k = \text{raise}
\]

Each time we catch a different raise function is created.
In order for things to work properly, three rules are actually needed:

- the previous catch / raise reduction: for \( k \notin \text{FV}(t) \),
  \[
  C(\lambda k \neg A. k t) \rightarrow_\beta t
  \]
In order for things to work properly, three rules are actually needed:

- the previous catch / raise reduction: for $k \notin \text{FV}(t)$,
  \[ C(\lambda k \neg A \cdot k t) \rightarrow_{\beta} t \]

- application goes through catch:
  \[ C(\lambda k \neg (A \rightarrow B) \cdot t) u \rightarrow_{\beta} C(\lambda k' \neg B \cdot t[\lambda f \rightarrow_{A \rightarrow B} . k(f u)/k]) \]
In order for things to work properly, three rules are actually needed:

- **the previous catch / raise reduction**: for \( k \notin \text{FV}(t) \),
  
  \[
  C(\lambda k \neg A. k \ t) \rightarrow_\beta t
  \]

- **application goes through catch**:
  
  \[
  C(\lambda k \neg (A \rightarrow B). t) \ u \rightarrow_\beta C(\lambda k' \neg B. t[\lambda f \rightarrow B. k (f \ u)/k])
  \]

- **re-raising is the same as raising**:
  
  \[
  C(\lambda k \neg A. k C(\lambda k' \neg A. t)) \rightarrow_\beta C(\lambda k'' \neg A. t[k''/k, k''/k'])
  \]
The operator $C$ is due to Felleisen.

A well-known variant is **call-cc** $cc$ (for *call with current continuation*) which is typed as

\[ cc : (\neg A \rightarrow A) \rightarrow A \]
A proof for excluded middle is

\[ \vdash \neg A \lor A \]
A proof for excluded middle is

\[ \vdash \neg \neg (\neg A \lor A) \]

\[ \vdash \neg A \lor A \]

\[ \neg \neg E \]
A proof for excluded middle is

\[ \neg(\neg A \lor A) \vdash \bot \]

(\neg E)

\[ \vdash \neg \neg(\neg A \lor A) \]

(\neg I)

\[ \vdash \neg A \lor A \]

(\neg \neg E)
A proof for excluded middle is

\[
\neg(\neg A \lor A) \vdash \neg A \lor A \\
\neg(\neg A \lor A) \vdash \bot \\
\vdash \neg(\neg A \lor A) \\
\vdash \neg A \lor A
\]

(\neg E)

(\neg I)

(\neg \neg E)
A proof for excluded middle is

\[ \neg (\neg A \lor A) \vdash \neg A \]

(\text{\textsc{\neg I}})

\[ \neg (\neg A \lor A) \vdash \neg A \lor A \]

(\text{\textsc{\lor L}})

\[ \neg (\neg A \lor A) \vdash \bot \]

(\text{\textsc{\neg E}})

\[ \vdash \neg \neg (\neg A \lor A) \]

(\text{\textsc{\neg I}})

\[ \vdash \neg A \lor A \]

(\text{\textsc{\neg \neg E}}}
A proof for excluded middle is

\[ \neg(\neg A \lor A), A \vdash \bot \]

\[ \neg(\neg A \lor A) \vdash \neg A \]

\[ \neg(\neg A \lor A) \vdash \neg A \lor A \]

\[ \neg(\neg A \lor A) \vdash \bot \]

\[ \vdash \neg \neg(\neg A \lor A) \]

\[ \vdash \neg A \lor A \]
A proof for excluded middle is

\[ \neg(\neg A \lor A), A \vdash \neg A \lor A \]  (\neg E)

\[ \neg(\neg A \lor A), A \vdash \bot \]  (\neg I)

\[ \neg(\neg A \lor A) \vdash \neg A \]  (\neg I)

\[ \neg(\neg A \lor A) \vdash \neg A \lor A \]  (\lor I)

\[ \neg(\neg A \lor A) \vdash \bot \]  (\neg E)

\[ \vdash \neg(\neg A \lor A) \]  (\neg I)

\[ \vdash \neg A \lor A \]  (\neg\neg E)
A proof for excluded middle is

\[ \neg(\neg A \lor A) \quad A \vdash A \quad (\lor I) \]

\[ \neg(\neg A \lor A) \quad A \vdash \neg A \lor A \quad (\neg E) \]

\[ \neg(\neg A \lor A) \quad A \vdash \bot \quad (\bot) \]

\[ \neg(\neg A \lor A) \vdash \neg A \quad (\neg I) \]

\[ \neg(\neg A \lor A) \vdash \neg A \lor A \quad (\lor I) \]

\[ \neg(\neg A \lor A) \vdash \bot \quad (\neg E) \]

\[ \vdash \neg(\neg A \lor A) \quad (\neg I) \]

\[ \vdash \neg A \lor A \quad (\neg \neg E) \]
A proof for excluded middle is

\[
\neg(\neg A \lor A), A \vdash A
\]

(ax)

\[
\neg(\neg A \lor A), A \vdash \neg A \lor A
\]

(\lor l)

\[
\neg(\neg A \lor A), A \vdash \bot
\]

(\neg E)

\[
\neg(\neg A \lor A) \vdash \neg A
\]

(\neg l)

\[
\neg(\neg A \lor A) \vdash \neg A \lor A
\]

(\lor l)

\[
\neg(\neg A \lor A) \vdash \bot
\]

(\neg E)

\[
\bot \vdash \neg(\neg A \lor A)
\]

(\neg l)

\[
\bot \vdash \neg \neg(\neg A \lor A)
\]

(\neg E)

\[
\vdash \neg A \lor A
\]

(\neg¬ E)
A term for excluded middle is

\[
\begin{align*}
\text{ax} & \quad k : \neg(\neg A \lor A), \quad a : A \vdash A \\
\text{\lor}_i & \quad k : \neg(\neg A \lor A), \quad a : A \vdash \neg A \lor A \\
\text{\neg}_e & \quad k : \neg(\neg A \lor A), \quad a : A \vdash \bot \\
\text{\neg}_i & \quad k : \neg(\neg A \lor A) \vdash \neg A \\
\text{\lor}_i & \quad k : \neg(\neg A \lor A) \vdash \neg A \lor A \\
\text{\neg}_e & \quad k : \neg(\neg A \lor A) \vdash \bot \\
\text{\neg}_i & \quad \vdash \neg(\neg A \lor A) \\
\text{\neg}_e & \quad \vdash \neg A \lor A
\end{align*}
\]
A term for excluded middle is

\[
\begin{align*}
\text{(ax)} & \quad k : \neg(\neg A \lor A), a : A & \vdash a : A \\
(\lor_{\iota}) & \quad k : \neg(\neg A \lor A), a : A & \vdash \neg A \lor A \\
(\neg_{E}) & \quad k : \neg(\neg A \lor A), a : A & \vdash \bot \\
(\neg_{I}) & \quad k : \neg(\neg A \lor A) & \vdash \neg A \\
(\lor_{\iota}) & \quad k : \neg(\neg A \lor A) & \vdash \neg A \lor A \\
(\neg_{E}) & \quad k : \neg(\neg A \lor A) & \vdash \bot \\
(\neg_{I}) & \quad k : \neg(\neg A \lor A) & \vdash \bot \\
(\neg_{E}) & \quad k : \neg(\neg A \lor A) & \vdash \bot \\
(\neg_{E}) & \quad k : \neg(\neg A \lor A) & \vdash \bot \\
\vdash & \quad \neg(\neg(\neg A \lor A)) & \vdash \\
(\neg_{E}) & \quad \vdash & \vdash \\
\vdash & \quad \neg(\neg A \lor A) & \vdash \\
\vdash & \quad \neg A \lor A & \vdash \\
\vdash & \quad \bot & \vdash \\
\vdash & \quad \bot & \vdash \\
\vdash & \quad \bot & \vdash
\end{align*}
\]
A term for excluded middle is

\[ k : \neg(\neg A \lor A), a : A \vdash a : A \]  \hspace{1cm} \text{(ax)}

\[ k : \neg(\neg A \lor A), a : A \vdash \iota_r(a) : \neg A \lor A \]  \hspace{1cm} \text{(
lor_i)}

\[ k : \neg(\neg A \lor A), a : A \vdash \bot \]  \hspace{1cm} \text{(\bot_i)}

\[ k : \neg(\neg A \lor A) \vdash \neg A \]  \hspace{1cm} \text{(\neg_I)}

\[ k : \neg(\neg A \lor A) \vdash \neg A \lor A \]  \hspace{1cm} \text{(\lor_E)}

\[ k : \neg(\neg A \lor A) \vdash \bot \]  \hspace{1cm} \text{(\bot_i)}

\[ \vdash \neg(\neg A \lor A) \]  \hspace{1cm} \text{(\neg_E)}

\[ \vdash \neg A \lor A \]  \hspace{1cm} \text{(\neg_E)}
A term for excluded middle is

\[
\begin{align*}
\frac{k : \neg(\neg A \lor A), a : A \vdash a : A}{(ax)} \\
k : \neg(\neg A \lor A), a : A \vdash \iota_r(a) : \neg A \lor A \quad (\lor_i) \\
k : \neg(\neg A \lor A), a : A \vdash k \iota_r(a) : \bot \quad (\neg E) \\
k : \neg(\neg A \lor A) \vdash \neg A \quad (\neg_i) \\
k : \neg(\neg A \lor A) \vdash \neg A \lor A \quad (\lor_i) \\
k : \neg(\neg A \lor A) \vdash \bot \quad (\neg E) \\
\vdash \neg \neg(\neg A \lor A) \quad (\neg\neg E) \\
\vdash \neg A \lor A
\end{align*}
\]
A term for excluded middle is

\[
\begin{align*}
\vdash & \quad k : \neg (\neg A \lor A), \ a : A \vdash \ a : A \quad \text{(ax)} \\
\vdash & \quad k : \neg (\neg A \lor A), \ a : A \vdash \ \iota_r(a) : \neg A \lor A \\
\vdash & \quad k : \neg (\neg A \lor A), \ a : A \vdash \ k \iota_r(a) : \bot \\
\vdash & \quad k : \neg (\neg A \lor A) \vdash \ \lambda a^A. k \iota_r(a) : \neg A \\
\vdash & \quad k : \neg (\neg A \lor A) \vdash \ \neg A \lor A \\
\vdash & \quad \vdash \ \bot \\
\vdash & \quad \vdash \ \neg (\neg A \lor A) \\
\vdash & \quad \vdash \ \neg A \lor A
\end{align*}
\]
A term for excluded middle is

\[
\begin{align*}
\kappa : \neg(\neg A \lor A), \ a : A & \vdash a : A \quad (\text{ax}) \\
\kappa : \neg(\neg A \lor A), \ a : A & \vdash \iota_r(a) : \neg A \lor A \quad (\lor_1) \\
\kappa : \neg(\neg A \lor A), \ a : A & \vdash \kappa \iota_r(a) : \bot \quad (\neg_\bot) \\
\kappa : \neg(\neg A \lor A) & \vdash \lambda a^A. \kappa \iota_r(a) : \neg A \quad (\forall_1) \\
\kappa : \neg(\neg A \lor A) & \vdash \iota_l(\lambda a^A. \kappa \iota_r(a)) : \neg A \lor A \quad (\lor_1) \\
\kappa : \neg(\neg A \lor A) & \vdash \bot \quad (\neg_\bot) \\
\vdash & \quad \neg(\neg A \lor A) \quad (\neg_\bot) \\
\vdash & \quad \neg A \lor A \quad (\neg_\bot)
\end{align*}
\]
A term for excluded middle is

\[
\begin{align*}
&k : \neg(\neg A \lor A), a : A \vdash a : A \tag{ax} \\
&k : \neg(\neg A \lor A), a : A \vdash \iota_r(a) : \neg A \lor A \tag{\lor_1'} \\
&k : \neg(\neg A \lor A), a : A \vdash k \iota_r(a) : \bot \tag{\neg E} \\
&k : \neg(\neg A \lor A) \vdash \lambda a^A.k \iota_r(a) : \neg A \tag{\lor_1} \\
&k : \neg(\neg A \lor A) \vdash \iota_l(\lambda a^A.k \iota_r(a)) : \neg A \lor A \tag{\neg E} \\
&k : \neg(\neg A \lor A) \vdash k \iota_l(\lambda a^A.k \iota_r(a)) : \bot \tag{\neg I} \\
&\vdash \neg \neg(\neg A \lor A) \tag{\neg \neg E} \\
&\vdash \neg A \lor A 
\end{align*}
\]
Classical logic: excluded middle

A term for excluded middle is

\[
\begin{align*}
  k & : \neg(\neg A \vee A), a : A \vdash a : A \quad \text{(ax)} \\
  k & : \neg(\neg A \vee A), a : A \vdash \iota_r(a) : \neg A \vee A \quad \text{(\neg_\exists)} \\
  k & : \neg(\neg A \vee A), a : A \vdash k \iota_r(a) : \bot \quad \text{(\neg_\exists)} \\
  k & : \neg(\neg A \vee A) \vdash \lambda a^A. k \iota_r(a) : \neg A \quad \text{(\neg_\exists)} \\
  k & : \neg(\neg A \vee A) \vdash \iota_l(\lambda a^A. k \iota_r(a)) : \neg A \vee A \quad \text{(\neg_\exists)} \\
  k & : \neg(\neg A \vee A) \vdash k \iota_l(\lambda a^A. k \iota_r(a)) : \bot \quad \text{(\neg_\exists)} \\
  \vdash \lambda k^{\neg(\neg A \vee A)}. k \iota_l(\lambda a^A. k \iota_r(a)) : \neg(\neg A \vee A) \quad \text{(\neg_\exists)} \\
  \vdash \neg A \vee A
\end{align*}
\]
Classical logic: excluded middle

A term for excluded middle is

\[
\begin{align*}
    k : \neg (\neg A \lor A), a : A & \vdash a : A \\
    k : \neg (\neg A \lor A), a : A & \vdash \iota_r(a) : \neg A \lor A \\
    k : \neg (\neg A \lor A), a : A & \vdash k \iota_r(a) : \bot \\
    k : \neg (\neg A \lor A) & \vdash \lambda a^A. k \iota_r(a) : \neg A \\
    k : \neg (\neg A \lor A) & \vdash \iota_l(\lambda a^A. k \iota_r(a)) : \neg A \lor A \\
    k : \neg (\neg A \lor A) & \vdash k \iota_l(\lambda a^A. k \iota_r(a)) : \bot \\
    \vdash \lambda k \neg (\neg A \lor A). k \iota_l(\lambda a^A. k \iota_r(a)) : \neg \neg (\neg A \lor A) \\
    \vdash C(\lambda k \neg (\neg A \lor A). k \iota_l(\lambda a^A. k \iota_r(a))) : \neg A \lor A
\end{align*}
\]
Part VII

Strong normalization
A term $t$ is **strongly normalizing** (or SN, or **terminating**) if there is no infinite sequence of reductions starting from $t$:

$$t \rightarrow_{\beta} t_1 \rightarrow_{\beta} t_2 \rightarrow_{\beta} t_3 \rightarrow_{\beta} \ldots$$
Strong normalization

Theorem (Strong normalization)
The simply-typed λ-calculus is strongly normalizing: given a typable λ-term $t$, there is no infinite sequence of β-reductions starting from $t$. 

# let omega = (fun x -> x x) (fun x -> x x);;

^ Error: This expression has type 'a -> 'b but an expression was expected of type 'a

The type variable 'a occurs inside 'a -> 'b
Strong normalization

**Theorem (Strong normalization)**

The simply-typed $\lambda$-calculus is strongly normalizing: given a typable $\lambda$-term $t$, there is no infinite sequence of $\beta$-reductions starting from $t$.

For instance, the $\lambda$-term

$$ (\lambda x.x)(\lambda x.x) $$

is not typable.
Theorem (Strong normalization)

The simply-typed $\lambda$-calculus is strongly normalizing: given a typable $\lambda$-term $t$, there is no infinite sequence of $\beta$-reductions starting from $t$.

For instance, the $\lambda$-term

$$(\lambda x.x x)(\lambda x.x x)$$

is not typable.

# let omega = (fun x -> x x) (fun x -> x x);;
   ^

Error: This expression has type 'a -> 'b
but an expression was expected of type 'a
The type variable 'a occurs inside 'a -> 'b
Deciding $\beta$-equivalence

Recall that $\beta$-equivalence is the smallest equivalence relation generated by $\beta$-reduction.

This means that $t \equiv_{\beta} u$ when there exists a sequence of reductions

$t \leftarrow^* t_1 \rightarrow^* t_2 \leftarrow^* t_3 \rightarrow^* t_4 \leftarrow^* \ldots \rightarrow^* u$

How can we decide whether two terms are $\beta$-equivalent or not?

(remember this is undecidable for untyped $\lambda$-calculus)
A first simplification:

**Theorem (Church-Rosser)**
*Two terms* $t$ *and* $u$ *are* $\beta$-equivalent *iff and only if* there exists *$w$ such that*

$$t \xrightarrow{\beta} w \xleftarrow{\beta} u$$

**Proof.**
The only if part is obvious. Suppose that we have

$$t = u_0 \xrightarrow{*} t_1 \xrightarrow{*} t_2 \xrightarrow{*} t_3 \ldots \xrightarrow{*} t_n = u_n = u$$

we show the result by induction on $n$. For $n = 0$, $t = u$ and the result is immediate.
The Church-Rosser theorem

A first simplification:

**Theorem (Church-Rosser)**

*Two terms* $t$ and $u$ *are* $\beta$-*equivalent iff and only if there exists* $w$ *such that*

$$
\begin{align*}
  t & \rightarrow^{\beta} w \leftarrow^{\beta} u
\end{align*}
$$

**Proof.**

The only if part is obvious. Suppose that we have

$$
\begin{align*}
  t &= u_0 \\
  t_1 &= u_1 \\
  t_2 &= u_2 \\
  \vdots \\
  t_n &= u_n = u
\end{align*}
$$

Otherwise,
The Church-Rosser theorem

A first simplification:

**Theorem (Church-Rosser)**
*Two terms $t$ and $u$ are $\beta$-equivalent iff and only if there exists $w$ such that*

$$t \rightarrow^* \beta w \leftarrow^* u$$

**Proof.**
The only if part is obvious. Suppose that we have

![Diagram](attachment:image.png)

Otherwise, by confluence.
The Church-Rosser theorem

A first simplification:

**Theorem (Church-Rosser)**
*Two terms $t$ and $u$ are $\beta$-equivalent iff and only if there exists $w$ such that*

$$t \xrightarrow{\beta} w \xleftarrow{\beta} u$$

**Proof.**
The only if part is obvious. Suppose that we have

$$t = u_0 \xrightarrow{\beta} u_1 \xrightarrow{\beta} \cdots \xrightarrow{\beta} u_n = u$$

Otherwise, by confluence and induction hypothesis.
We thus left with deciding whether two terms $t$ and $u$ are joinable, i.e. there exists $w$ such that

\[ t \xrightarrow{\beta} w \xleftarrow{\beta} u \]
Normal forms

A term $t$ is a **normal form** when there is no $t'$ such that $t \rightarrow_{\beta} t'$. 

**Lemma**

Every typable term $t$ is $\beta$-equivalent to a normal form $\hat{t}$.

**Proof.**

Given a term $t$ reduce it as much as possible:

$t \rightarrow_{\beta} t_1 \rightarrow_{\beta} t_2 \rightarrow_{\beta} \cdots \rightarrow_{\beta} t_n = \hat{t}$

This process will stop because typable terms are strongly normalizing and $t_n$ is a normal form.

**Strongly normalizing:** any sequence of reductions will lead to a normal form.
A term $t$ is a **normal form** when there is no $t'$ such that $t \xrightarrow{\beta} t'$.

**Lemma**
Every typable term $t$ is $\beta$-equivalent to a normal form $\hat{t}$.

**Proof.**
Given a term $t$ reduce it as much as possible:

$$t \xrightarrow{\beta} t_1 \xrightarrow{\beta} t_2 \xrightarrow{\beta} \cdots \xrightarrow{\beta} t_n = \hat{t}$$

This process will stop because typable terms are strongly normalizing and $t_n$ is a normal form.
Normal forms

A term $t$ is a **normal form** when there is no $t'$ such that $t \rightarrow_{\beta} t'$.

**Lemma**

*Every typable term $t$ is $\beta$-equivalent to a normal form $\hat{t}$.***

**Proof.**

Given a term $t$ reduce it as much as possible:

$$
\begin{align*}
  t & \rightarrow_{\beta} t_1 \\
  t_1 & \rightarrow_{\beta} t_2 \\
  \vdots \\
  t_n & = \hat{t}
\end{align*}
$$

This process will stop because typable terms are strongly normalizing and $t_n$ is a normal form.

**Strongly normalizing:** any sequence of reductions will lead to a normal form.
Lemma
Two normal forms $t$ and $u$ are $\beta$-equivalent iff they are equal.

Proof.
The only if part is obvious. For the if part, by the Church-Rosser theorem we have

$$t \xrightarrow{\beta} w \xleftarrow{\beta} u$$

but since $t$ and $u$ are normal forms, we actually have

$$t = w = u$$
Deciding $\beta$-equivalence

Suppose given two typable terms $t$ and $u$. The following are equivalent

- $t \xrightarrow{\beta} u$
- $\hat{t} \xrightarrow{\beta} \hat{u}$
- $\hat{t} = \hat{u}$

Which can be pictured as

```
t \xrightarrow{?} u
```

```
\hat{t} \xrightarrow{?} \hat{u}
```
This still holds for extensions of $\lambda$-calculus: products, coproducts, natural numbers, etc.

In particular, for natural numbers it is important that the recursive calls are performed on smaller numbers, which ensures termination.
Part VIII

Type inference à la Curry
Curry style $\lambda$-calculus

In Curry style, $\lambda$-terms are

$$t ::= x \mid tu \mid \lambda x.t$$

and the rules are

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \text{(ax)} \quad \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} \text{ (}\rightarrow\text{E)} \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \rightarrow B} \text{ (}\rightarrow\text{I)}$$

How do we compute all those types?
In Curry style, $\lambda$-terms are

$$ t ::= x \mid t \ u \mid \lambda x.\ t $$

and the rules are

$$ \Gamma \vdash x : \Gamma(x) \quad \frac{\Gamma \vdash t : A \to B}{\Gamma \vdash t\ u : B} \quad \frac{\Gamma \vdash u : A}{\Gamma \vdash t\ u : B} \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\ t : A \to B} $$

A term can have multiple types:

$$ \frac{x : A \vdash x : A}{\vdash \lambda x.\ x : A \to A} \quad \frac{x : A \to A \vdash x : A \to A}{\vdash \lambda x.\ x : (A \to A) \to (A \to A)} $$
Curry style $\lambda$-calculus

In Curry style, $\lambda$-terms are

$$t ::= x \mid t \ u \mid \lambda x. t$$

and the rules are

$$\Gamma \vdash x : \Gamma(x) \hspace{1cm} \Gamma \vdash t : A \rightarrow B \hspace{1cm} \Gamma \vdash u : A \hspace{1cm} \Gamma, x : A \vdash t : B \hspace{1cm} \Gamma \vdash \lambda x. t : A \rightarrow B$$

\[(\rightarrow_E)\]

\[(\rightarrow_I)\]

A term can have multiple types:

$$x : A \vdash x : A \hspace{1cm} x : A \rightarrow A \vdash x : A \rightarrow A \rightarrow_A \hspace{1cm} x : A \rightarrow A \vdash x : (A \rightarrow A) \rightarrow (A \rightarrow A)$$

\[\rightarrow_I\]

\[(\rightarrow_I)\]

How do we compute all those types?
A substitution is a function which to type variables associate terms. For instance

\[ \sigma(X) = A \rightarrow B \quad \text{and} \quad \sigma(Y) = A \]
A substitution is a function which to type variables associate terms. For instance

\[ \sigma(X) = A \rightarrow B \quad \quad \sigma(Y) = A \]

We write \( A[\sigma] \) for the type \( A \) where variables have been replaced according to \( \sigma \):

\[ (X \rightarrow Y)[\sigma] = (A \rightarrow B) \rightarrow A \]
A type equation system is a finite set

\[ E = \{ A_1 \neq B_1, \ldots, A_n \neq B_n \} \]

of pairs of types \( A_i \) and \( B_i \).
A type equation system is a finite set

\[ E = \{ A_1 \not\cong B_1, \ldots, A_n \not\cong B_n \} \]

of pairs of types \( A_i \) and \( B_i \).

A substitution \( \sigma \) is a solution of \( E \) if

\[ A_i[\sigma] = B_i[\sigma] \]

for every index \( i \).
Type equation systems

A type equation system is a finite set

\[ E = \{ A_1 \not\equiv B_1, \ldots, A_n \not\equiv B_n \} \]

of pairs of types \( A_i \) and \( B_i \).

A substitution \( \sigma \) is a solution of \( E \) if

\[ A_i[\sigma] = B_i[\sigma] \]

for every index \( i \).

For instance, a solution of

\[ \{(X \to Y) \not\equiv (Z \to (Z \to Z))\} \]
A type equation system is a finite set

\[ E = \{ A_1 \neq B_1, \ldots, A_n \neq B_n \} \]

of pairs of types \( A_i \) and \( B_i \).

A substitution \( \sigma \) is a solution of \( E \) if

\[ A_i[\sigma] = B_i[\sigma] \]

for every index \( i \).

For instance, a solution of

\[ \{(X \to Y) \neq (Z \to (Z \to Z))\} \]

is \( \sigma(X) = Z, \ \sigma(Y) = Z \to Z \).
A type equation system is a finite set

\[ E = \{ A_1 \not\approx B_1, \ldots, A_n \not\approx B_n \} \]

of pairs of types \( A_i \) and \( B_i \).

A substitution \( \sigma \) is a solution of \( E \) if

\[ A_i[\sigma] = B_i[\sigma] \]

for every index \( i \).

For instance, a solution of

\[ \{ (X \rightarrow Y) \not\approx Y \} \]
A type equation system is a finite set
\[ E = \{ A_1 \not= B_1, \ldots, A_n \not= B_n \} \]
of pairs of types \( A_i \) and \( B_i \).

A substitution \( \sigma \) is a solution of \( E \) if
\[ A_i[\sigma] = B_i[\sigma] \]
for every index \( i \).

For instance, a solution of
\[ \{ (X \to Y) \not= Y \} \]
does not exist.
Typing as solving constraints

We are going to associate to each term $t$

- a type $A_t$
- an equation system $E_t$

such that

- $t$ is typable iff $E_t$ has a solution $\sigma$,
- in which case the $A_t[\sigma]$ are the possible types of $t$. 
Typing as solving constraints

The rules

\[
\frac{\Gamma \vdash x : \Gamma(x)}{\Gamma \vdash x : \Gamma(x)} \quad (ax) \quad \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash t \ u : B} \quad (\rightarrow E) \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. t : A \rightarrow B} \quad (\rightarrow I)
\]

suggest that

- to every term variable \( x \), we associate a type variable \( X_x \),
- to every term \( t \), we associate a type \( A_t \),
- to every term \( t \), we associate an equation system \( E_t \)

by induction by

\[
E_x = \emptyset \quad A_x = X_x \\
E_{t \ u} = E_t \cup E_u \cup \{ A_t \neq (A_u \rightarrow X) \} \quad A_{t \ u} = X \quad \text{with } X \text{ fresh} \\
E_{\lambda x. t} = E_t \quad A_{\lambda x. t} = X_x \rightarrow A_t
\]
For instance, consider

\[ t = \lambda f.f(f(\lambda x.x)) \]

we have
Typing as solving constraints

For instance, consider

\[ t = \lambda f . f(f(\lambda x . x)) \]

we have

\[ E_x = \emptyset \quad A_x = X_x \]
Typing as solving constraints

For instance, consider

\[ t = \lambda f. f(f(\lambda x. x)) \]

we have

\[ E_x = \emptyset \]
\[ A_x = X \]
\[ E_{\lambda x.x} = \emptyset \]
\[ A_{\lambda x.x} = X \rightarrow X \]
Typing as solving constraints

For instance, consider

\[ t = \lambda f.f(f(\lambda x.x)) \]

we have

\[
\begin{align*}
E_x &= \emptyset \\
E_{\lambda x.x} &= \emptyset \\
E_{f(\lambda x.x)} &= \{ X_f \in \exists(X_x \rightarrow X_x) \rightarrow X \} \\
A_x &= X_x \\
A_{\lambda x.x} &= X_x \rightarrow X_x \\
A_{f(\lambda x.x)} &= X
\end{align*}
\]
For instance, consider

\[ t = \lambda f. f(f(\lambda x.x)) \]

we have

\[ E_x = \emptyset \]
\[ E_{\lambda x.x} = \emptyset \]
\[ E_{f(\lambda x.x)} = \{ X_f \neq (X_x \rightarrow X_x) \rightarrow X \} \]
\[ E_{f(f(\lambda x.x))} = \{ X_f \neq (X_x \rightarrow X_x) \rightarrow X, X_f \neq X \rightarrow Y \} \]
\[ A_x = X_x \]
\[ A_{\lambda x.x} = X_x \rightarrow X_x \]
\[ A_{f(\lambda x.x)} = X \]
\[ A_{f(f(\lambda x.x))} = Y \]
Typing as solving constraints

For instance, consider

\[ t = \lambda f.f(f(\lambda x.x)) \]

we have

\[
\begin{align*}
E_x &= \emptyset \\
A_x &= X_x \\
E_{\lambda x.x} &= \emptyset \\
A_{\lambda x.x} &= X_x \rightarrow X_x \\
E_{f(\lambda x.x)} &= \{X_f \nvdash (X_x \rightarrow X_x) \rightarrow X\} \\
A_{f(\lambda x.x)} &= X \\
E_{f(f(\lambda x.x))} &= \{X_f \nvdash (X_x \rightarrow X_x) \rightarrow X, \ X_f \nvdash X \rightarrow Y\} \\
A_{f(f(\lambda x.x))} &= Y \\
E_{\lambda f.f(f(\lambda x.x))} &= \{X_f \nvdash (X_x \rightarrow X_x) \rightarrow X, \ X_f \nvdash X \rightarrow Y\} \\
A_{\lambda f.f(f(\lambda x.x))} &= X_f \rightarrow Y
\end{align*}
\]
Typing as solving constraints

For instance, consider

\[ t = \lambda f. f(f(\lambda x.x)) \]

we have

\[ E_{\lambda f. f(f(\lambda x.x))} = \{ X_f \not\equiv (X_x \rightarrow X_x) \rightarrow X, X_f \not\equiv X \rightarrow Y \} \]

\[ A_{\lambda f. f(f(\lambda x.x))} = X_f \rightarrow Y \]

A solution is
Typing as solving constraints

For instance, consider

\[ t = \lambda f. f(f(\lambda x. x)) \]

we have

\[ E_{\lambda f. f(f(\lambda x. x))} = \{ X_f \not\subset (X \to X) \to X, X_f \not\subset X \to Y \} \quad A_{\lambda f. f(f(\lambda x. x))} = X_f \to Y \]

A solution is

\[ \sigma(X_x) = A \]
Typing as solving constraints

For instance, consider

\[ t = \lambda f.f(f(\lambda x.x)) \]

we have

\[ E_{\lambda f.f(f(\lambda x.x))} = \{ X_f \not\equiv (X_x \rightarrow X_x) \rightarrow X, X_f \not\equiv X \rightarrow Y \} \quad A_{\lambda f.f(f(\lambda x.x))} = X_f \rightarrow Y \]

A solution is

\[ \sigma(X_x) = A \quad \sigma(X) = A \rightarrow A \]
Typing as solving constraints

For instance, consider

$$t = \lambda f. f(f(\lambda x.x))$$

we have

$$E_{\lambda f. f(f(\lambda x.x))} = \{ X_f \not\vdash (X_x \to X_x) \to X, X_f \not\vdash X \to Y \}$$

so

$$A_{\lambda f. f(f(\lambda x.x))} = X_f \to Y$$

A solution is

$$\sigma(X_x) = A \quad \sigma(X) = A \to A \quad \sigma(Y) = A \to A$$
Typing as solving constraints

For instance, consider

\[ t = \lambda f . f(f(\lambda x . x)) \]

we have

\[ E_{\lambda f.f(f(\lambda x . x))} = \{ X_f \nvdash (X_x \rightarrow X_x) \rightarrow X, X_f \nvdash X \rightarrow Y \} \quad A_{\lambda f.f(f(\lambda x . x))} = X_f \rightarrow Y \]

A solution is

\[ \sigma(X_x) = A \quad \sigma(X) = A \rightarrow A \quad \sigma(Y) = A \rightarrow A \quad \sigma(X_f) = (A \rightarrow A) \rightarrow (A \rightarrow A) \]
Typing as solving constraints

For instance, consider

\[ t = \lambda f. f(f(\lambda x.x)) \]

we have

\[ E_{\lambda f.f(f(\lambda x.x))} = \{ X_f \not\equiv (X_x \to X_x) \to X, X_f \not\equiv X \to Y \} \]

\[ A_{\lambda f.f(f(\lambda x.x))} = X_f \to Y \]

A solution is

\[ \sigma(X_x) = A \quad \sigma(X) = A \to A \quad \sigma(Y) = A \to A \quad \sigma(X_f) = (A \to A) \to (A \to A) \]

The resulting type is

\[ (X_f \to Y)[\sigma] = ((A \to A) \to (A \to A)) \to A \to A \]
Typing as solving constraints

For instance, consider

\[ t = \lambda f.f(f(\lambda x.x)) \]

we have

\[ E_{\lambda f.f(f(\lambda x.x))} = \{ X_f \not\subseteq (X \rightarrow X) \rightarrow X, X_f \not\subseteq X \rightarrow Y \} \quad A_{\lambda f.f(f(\lambda x.x))} = X_f \rightarrow Y \]

A solution is

\[ \sigma(X_x) = A \quad \sigma(X) = A \rightarrow A \quad \sigma(Y) = A \rightarrow A \quad \sigma(X_f) = (A \rightarrow A) \rightarrow (A \rightarrow A) \]

The resulting type is

\[ (X_f \rightarrow Y)[\sigma] = ((A \rightarrow A) \rightarrow (A \rightarrow A)) \rightarrow A \rightarrow A \]

to be compared with

```
# fun f -> f (f (fun x -> x));
- : (('a -> 'a) -> 'a -> 'a) -> 'a -> 'a = <fun>
```
Typing as solving constraints

Note that the previous solution works for which ever type $A$ we choose.

Therefore there is an infinite number of solutions!
Typing as solving constraints

**Theorem**

*We have*

- if $\Gamma \vdash t : A$ then there is a solution $\sigma$ of $E_t$ such that $A = A_t[\sigma]$ and $\Gamma(x) = \sigma(X_x)$ for every variable $x \in \text{FV}(t)$,
- for every solution $\sigma$ of $E_t$, if we write $\Gamma$ for a context such that $\Gamma(x) = \sigma(x)$ for every free variable $x \in \text{FV}(t)$, then $\Gamma \vdash t : A_t[\sigma]$ is derivable.

Otherwise said, there is a bijection between

- solutions $\sigma$ of $E_t$,
- pairs $(\Gamma, A)$ such that $\Gamma \vdash t : A$ is derivable.
The unification algorithm takes a type equation system $E$ and produces a solution $\sigma$ when there exists one, in polynomial time.
The **unification** algorithm takes a type equation system $E$ and produces a solution $\sigma$ when there exists one, in polynomial time.

Moreover, this solution is the *most general one*: any other solution $\tau$ satisfies

$$\tau = \tau' \circ \sigma.$$
The **unification** algorithm takes a type equation system $E$ and produces a solution $\sigma$ when there exists one, in polynomial time.

Moreover, this solution is the **most general one**: any other solution $\tau$ satisfies

$$\tau = \tau' \circ \sigma$$

We can therefore compute a most general type for Curry-style $\lambda$-terms in P-time!
Part IX

Bidirectional typechecking
Bidirectional typechecking

Looking closely at the operations we perform during type-checking there are two phases.

- **Type inference**: we guess the type of a term.
- **Type checking**: we check that a term has a given type.

For instance, consider the type inference of

\[
\begin{align*}
\Gamma & : x : \mathbb{N} \\
\Gamma & : \lambda x : \mathbb{N}.x : \mathbb{N} \\
\Gamma & : 5 : \mathbb{N} \\
\Gamma & : (\lambda x : \mathbb{N}.x)5 : \mathbb{N}
\end{align*}
\]
Bidirectional typechecking formalizes this two phases, allowing to add type annotations (we could mix Church and Curry style).
We consider Curry-style terms with type annotations:

\[ t ::= x \mid t \ u \mid \lambda x. t \mid (x : A) \]
Bidirectional typechecking

We consider Curry-style terms with type annotations:

\[ t ::= x \mid t u \mid \lambda x. t \mid (x : A) \]

We consider two kind of sequents:

- \( \Gamma \vdash t \Rightarrow A \): the term \( t \) allows to synthesize the type \( A \) (type inference),
- \( \Gamma \vdash t \Leftarrow A \): the term \( t \) allows checks against the type \( A \) (type checking).
Bidirectional typechecking

We consider Curry-style terms with type annotations:

\[ t ::= x \mid tu \mid \lambda x.t \mid (x : A) \]

We consider two kind of sequents:

- \( \Gamma \vdash t \Rightarrow A \): the term \( t \) allows to synthesize the type \( A \) (type inference),
- \( \Gamma \vdash t \Leftarrow A \): the term \( t \) allows checks against the type \( A \) (type checking).

We can then orient the typing rules

\[ \Gamma \vdash t : A \quad \text{as} \quad \Gamma \vdash t \Rightarrow A \quad \text{or} \quad \Gamma \vdash t \Leftarrow A \]
Bidirectional typechecking

Orientation of the base rules

- variable: we already have the information in the context

  \[ \Gamma \vdash x \Rightarrow \Gamma(x) \] (ax)

- application: we cannot come up with \( B \), we have to check the type of the argument

  \[ \Gamma \vdash t \Rightarrow A \rightarrow B \]
  \[ \Gamma \vdash u \Leftarrow A \]
  \[ \Gamma \vdash t \, u : B \rightarrow E \] (\( \rightarrow E \))

- abstraction: we cannot come up with \( A \) in Curry style (and typing is not unique)

  \[ \Gamma, x : A \vdash t \Leftarrow B \]
  \[ \Gamma \vdash \lambda x. t \Leftarrow A \rightarrow B \] (\( \rightarrow I \))
Bidirectional typechecking

Orientation of the base rules

- **variable**: we already have the information in the context

  \[ \Gamma \vdash x \Rightarrow \Gamma(x) \]

  (ax)

- **application**: we cannot come up with \( B \), we have to check the type of the argument

  \[
  \begin{align*}
  \Gamma & \vdash t \Rightarrow A \rightarrow B \\
  \Gamma & \vdash u \Leftarrow A \\
  \hline
  \Gamma & \vdash t \ u : B
  \end{align*}
  \]

  \((\rightarrow_E)\)
Bidirectional typechecking

Orientation of the base rules

• variable: we already have the information in the context

\[ \Gamma \vdash x \Rightarrow \Gamma(x) \] (ax)

• application: we cannot come up with \( B \), we have to check the type of the argument

\[ \Gamma \vdash t \Rightarrow A \rightarrow B \quad \Gamma \vdash u \Leftarrow A \]
\[ \Gamma \vdash t u : B \] (\( \rightarrow_E \))

• abstraction: we cannot come up with \( A \) in Curry style (and typing is not unique)

\[ \Gamma, x : A \vdash t \Leftarrow B \]
\[ \Gamma \vdash \lambda x. t \Leftarrow A \rightarrow B \] (\( \rightarrow_I \))
Bidirectional typechecking

We have two new rules:

- subsumption: if we can infer then we can check

\[ \Gamma \vdash t \Rightarrow A \]
\[ \Gamma \vdash t \Leftarrow A \]
Bidirectional typechecking

We have two new rules:

- subsumption: if we can infer then we can check

\[ \Gamma \vdash t \Rightarrow A \]
\[ \therefore \Gamma \vdash t \Leftarrow A \]

- casting:

\[ \Gamma \vdash t \Leftarrow A \]
\[ \therefore \Gamma \vdash (t : A) \Rightarrow A \]
Bidirectional typechecking

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) 5 \ 7 \Rightarrow \mathbb{R} \]
Bidirectional typechecking

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) \ 5 \Rightarrow R \to R \]

\[ \Gamma \vdash 7 \Leftarrow R \]

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) \ 5 \ 7 \Rightarrow R \]
Bidirectional typechecking

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) \, 5 \Rightarrow \mathbb{R} \rightarrow \mathbb{R} \]

\[ \Gamma \vdash 7 \Rightarrow \mathbb{R} \]

\[ \Gamma \vdash 7 \Leftarrow \mathbb{R} \]

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) \, 5 \, 7 \Rightarrow \mathbb{R} \]
Bidirectional typechecking

\[ \Gamma \vdash \text{mean} \left( \lambda x.x \times x \right) \ 5 \Rightarrow \mathbb{R} \rightarrow \mathbb{R} \]

\[ \Gamma \vdash 7 \Rightarrow \mathbb{R} \]

\[ \Gamma \vdash 7 \Leftarrow \mathbb{R} \]

\[ \Gamma \vdash \text{mean} \left( \lambda x.x \times x \right) \ 5 \ 7 \Rightarrow \mathbb{R} \]
Bidirectional typechecking

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) \Rightarrow \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \]

\[ \Gamma \vdash 7 \Rightarrow \mathbb{R} \]

\[ \Gamma \vdash 7 \Leftarrow \mathbb{R} \]

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) 5 \Rightarrow \mathbb{R} \rightarrow \mathbb{R} \]

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) 5 7 \Rightarrow \mathbb{R} \]
Bidirectional typechecking

\[ \Gamma \vdash \text{mean} \Rightarrow (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R} \to \mathbb{R} \quad \Gamma \vdash \lambda x. x \Leftarrow \mathbb{R} \to \mathbb{R} \]

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) \Rightarrow \mathbb{R} \to \mathbb{R} \to \mathbb{R} \quad : \quad \Gamma \vdash 7 \Rightarrow \mathbb{R} \]

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) 5 \Rightarrow \mathbb{R} \to \mathbb{R} \quad \Gamma \vdash 7 \Leftarrow \mathbb{R} \]

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) 5 7 \Rightarrow \mathbb{R} \]
Bidirectional typechecking

\[ \Gamma \vdash \text{mean} \Rightarrow (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \]

\[ \Gamma, x : \mathbb{R} \vdash x \Leftarrow \mathbb{R} \]

\[ \Gamma \vdash \lambda x. x \Leftarrow \mathbb{R} \rightarrow \mathbb{R} \]

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) \Rightarrow \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \]

\[ \Gamma \vdash 7 \Leftarrow \mathbb{R} \]

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) 5 \Rightarrow \mathbb{R} \rightarrow \mathbb{R} \]

\[ \Gamma \vdash 7 \Rightarrow \mathbb{R} \]

\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) 5 7 \Rightarrow \mathbb{R} \]
Bidirectional typechecking

\[ \Gamma \vdash \text{mean} \Rightarrow (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R} \to \mathbb{R} \]

\[ \Gamma, x : \mathbb{R} \vdash x \Rightarrow \mathbb{R} \]

\[ \Gamma, x : \mathbb{R} \vdash x \Leftarrow \mathbb{R} \]

\[ \Gamma \vdash \lambda x . x \Leftarrow \mathbb{R} \to \mathbb{R} \]

\[ \Gamma \vdash \text{mean} (\lambda x . x \times x) \Rightarrow \mathbb{R} \to \mathbb{R} \]

\[ \Gamma \vdash \text{mean} (\lambda x . x \times x) 5 \Rightarrow \mathbb{R} \to \mathbb{R} \]

\[ \Gamma \vdash \text{mean} (\lambda x . x \times x) 5 7 \Rightarrow \mathbb{R} \]

\[ \Gamma \vdash 7 \Rightarrow \mathbb{R} \]

\[ \Gamma \vdash 7 \Leftarrow \mathbb{R} \]
Bidirectional typechecking

\[ \Gamma, x : \mathbb{R} \vdash x \Rightarrow \mathbb{R} \]
\[ \Gamma, x : \mathbb{R} \vdash x \Leftarrow \mathbb{R} \]
\[ \Gamma \vdash \text{mean} \Rightarrow (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \]
\[ \Gamma \vdash \lambda x. x \Leftarrow \mathbb{R} \rightarrow \mathbb{R} \]
\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) \Rightarrow \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \]
\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) 5 \Rightarrow \mathbb{R} \rightarrow \mathbb{R} \]
\[ \Gamma \vdash \text{mean} (\lambda x. x \times x) 5 7 \Rightarrow \mathbb{R} \]
If we try to define the mean function as

$$\lambda f x y. (f x + f y)/2$$

we cannot infer its type: we can only check the type of functions (i.e. when they are used as arguments of other functions).
Bidirectional typechecking

If we try to define the mean function as

\[ \lambda fxy. \left( f \, x + f \, y \right)/2 \]

we cannot infer its type: we can only check the type of functions (i.e. when they are used as arguments of other functions).

In order to define it, we have to provide its type

\[ \text{mean} = (\lambda fxy. \left( f \, x + f \, y \right)/2 : (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R} \to \mathbb{R}) \]
If we try to define the mean function as

$$\lambda fxy. (f x + f y)/2$$

we cannot infer its type: we can only check the type of functions (i.e. when they are used as arguments of other functions).

In order to define it, we have to provide its type

$$\text{mean} \ = \ (\lambda fxy. (f x + f y)/2 : (R \to R) \to R \to R \to R)$$

In Agda, the syntax for such definitions will be

$$\text{mean} \ : \ (R \to R) \to R \to R \to R$$
$$\text{mean} \ f \ x \ y \ = \ (x + y) / 2$$
The implementation is pretty direct.

We define types

type ty =
    | TVar of string
    | Arr of ty * ty
The implementation is pretty direct.

We define types

type ty =
  | TVar of string
  | Arr of ty * ty

and terms:

type term =
  | Var of string
  | App of term * term
  | Abs of string * term
  | Cast of term * ty
(** Type inference. *)

let rec infer env = function
  | Var x ->
    (try List.assoc x env
     with Not_found ->
       raise Type_error)
  | App (t, u) ->
    (match infer env t with
     | Arr (a, b) -> check env u a; b
     | _ -> raise Type_error
    )
  | Abs (x, t) -> raise Cannot_infer
  | Cast (t, a) -> check env t a; a

(** Type checking. *)

and check env t a =
  match t, a with
  | Abs (x, t), Arr (a, b) -> check ((x, a)::env) t b
  | _ -> if infer env t <> a then raise Type_error
(** Type inference. *)
let rec infer env = function
    | Var x -> (try List.assoc x env with Not_found -> raise Type_error)
    | App (t, u) -> (match infer env t with
                        | Arr (a, b) -> check env u a; b
                        | _ -> raise Type_error
                      )
    | Abs (x, t) -> raise Cannot_infer
    | Cast (t, a) -> check env t a; a

(** Type checking. *)
and check env t a = match t , a with
    | Abs (x, t) , Arr (a, b) -> check ((x, a)::env) t b
    | _ -> if infer env t <> a then raise Type_error
Part X

Proof of strong normalization
We now want to prove

**Theorem**
*Typed λ-terms are strongly normalizing.*

Given a term $t$ which is typable, there is no infinite sequence of reductions

$$t \rightarrow_{\beta} t_1 \rightarrow_{\beta} t_2 \rightarrow_{\beta} \ldots$$
Proof of strong normalization: first attempt

A naive try would be by induction on the derivation of $\Gamma \vdash t : A$.

If the last rule is

\[
\frac{}{\Gamma \vdash x : A} \quad \text{(ax)}
\]

$x$ is clearly strongly normalizing.
Proof of strong normalization: first attempt

A naive try would be by induction on the derivation of $\Gamma \vdash t : A$.

If the last rule is

$$
\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \rightarrow B} \quad (\rightarrow_i)
$$

A sequence of reductions is of the form

$$
\lambda x.t \rightarrow_\beta \lambda x.t_1 \rightarrow_\beta \lambda x.t_2 \rightarrow_\beta \ldots
$$

with

$$
t \rightarrow_\beta t_1 \rightarrow_\beta t_2 \rightarrow_\beta \ldots
$$

and we conclude by induction hypothesis.
Proof of strong normalization: first attempt

A naive try would be by induction on the derivation of $\Gamma \vdash t : A$.

If the last rule is

$$
\frac{
\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A
}{\Gamma \vdash t u : B}
$$

($\rightarrow_E$)

We cannot say anything: a reduction starting from $t u$ can be of the form

$$
t u \rightarrow t' u \quad \text{or} \quad t u \rightarrow t u'
$$

but also

$$(\lambda x.t)u \rightarrow t[u/x]$$

e.g. $\perp$, for which we cannot say anything.
Reducibility candidates

Instead, we take an optimistic approach and defined, for each type \( A \), a set

\[ R_A \]

of terms, the **reducibility candidates** for the type \( A \), such that

- for every term \( t \) of type \( A \) (in whichever context), we have \( t \in R_A \),
- the terms of \( R_A \) are obviously terminating.

**NB:**

- the definition has to be carefully crafted in order to be able to reason by induction,
- the terms of \( R_A \) are not necessarily of type \( A \) (although we could).
We define $R_A$ by induction on $A$ by

1. In the case of a variable, $R_X = \{ t \mid t$ is strongly normalizing $\}$
2. In the case of an arrow, $R_{A \rightarrow B} = \{ t \mid \text{for every } u \in R_A, \text{ we have } t \ u \in R_B \}$

We still have to show that

1. A term $t \in R_A$ is strongly normalizing,
2. If $\Gamma \vdash t : A$ is derivable then $t \in R_A$. 
We define $R_A$ by induction on $A$ by

- in the case of a variable,
  $$R_X = \{ t \mid t \text{ is strongly normalizing} \}$$
- in the case of an arrow,
  $$R_{A \to B} = \{ t \mid \text{for every } u \in R_A, \ t \ u \in R_B \}$$

We still have to show that

- a term $t \in R_A$ is strongly normalizing,
- if $\Gamma \vdash t : A$ is derivable then $t \in R_A$. 

107
We define $R_A$ by induction on $A$ by

- in the case of a variable,
  $$R_X = \{ t \mid t \text{ is strongly normalizing} \}$$

- in the case of an arrow,
  $$R_{A \rightarrow B} = \{ t \mid \text{for every } u \in R_A, \text{ we have } t u \in R_B \}$$
We define $R_A$ by induction on $A$ by

- in the case of a variable,
  $$R_X = \{ t \mid t \text{ is strongly normalizing} \}$$

- in the case of an arrow,
  $$R_{A \rightarrow B} = \{ t \mid \text{for every } u \in R_A, \text{ we have } t u \in R_B \}$$

We still have to show that

- a term $t \in R_A$ is strongly normalizing,
- if $\Gamma \vdash t : A$ is derivable then $t \in R_A$.  


Proposition
Suppose given a property $P(t)$ on terms. Suppose that, for any term $t$, if $P(t')$ for every $t'$ with $t \rightarrow_{\beta} t'$, then $P(t)$:

Then $P(t)$ holds for every SN term $t$. 
Proposition
Suppose given a property $P(t)$ on terms. Suppose that, for any term $t$, if $P(t')$ for every $t'$ with $t \xrightarrow{\beta} t'$, then $P(t)$:

Then $P(t)$ holds for every SN term $t$.

Proof.
By contraposition. Suppose that $P(t)$ does not hold for some SN term $t$. 

$t' \leftarrow t' \leftarrow t' \leftarrow t' \leftarrow \ldots \leftarrow t'_k$

$P(t)$

$P(t_i')$ for every $i$
Proposition
Suppose given a property $P(t)$ on terms. Suppose that, for any term $t$, if $P(t')$ for every $t'$ with $t \rightarrow_{\beta} t'$, then $P(t)$:

Then $P(t)$ holds for every SN term $t$.

Proof.
By contraposition. Suppose that $P(t)$ does not hold for some SN term $t$.
• then there exists $t_1$ with $t \rightarrow_{\beta} t_1$ such that $P(t_1)$ does not hold,
Induction for SN terms

**Proposition**

*Suppose given a property $P(t)$ on terms. Suppose that, for any term $t$, if $P(t')$ for every $t'$ with $t \rightarrow_{\beta} t'$, then $P(t)$:

$$t \quad P(t)$$

\[t' \quad P(t') \text{ for every } i\]

Then $P(t)$ holds for every SN term $t$.

**Proof.**

By contraposition. Suppose that $P(t)$ does not hold for some SN term $t$.

- then there exists $t_1$ with $t \rightarrow_{\beta} t_1$ such that $P(t_1)$ does not hold,
- then there exists $t_2$ with $t_1 \rightarrow_{\beta} t_2$ such that $P(t_2)$ does not hold,
Proposition
Suppose given a property $P(t)$ on terms. Suppose that, for any term $t$, if $P(t')$ for every $t'$ with $t \rightarrow_{\beta} t'$, then $P(t)$:

Then $P(t)$ holds for every SN term $t$.

Proof.
By contraposition. Suppose that $P(t)$ does not hold for some SN term $t$.
- then there exists $t_1$ with $t \rightarrow_{\beta} t_1$ such that $P(t_1)$ does not hold,
- then there exists $t_2$ with $t_1 \rightarrow_{\beta} t_2$ such that $P(t_2)$ does not hold,
- ...
Proposition
Suppose given a property $P(t)$ on terms. Suppose that, for any term $t$, if $P(t')$ for every $t'$ with $t \rightarrow_\beta t'$, then $P(t)$:

Then $P(t)$ holds for every SN term $t$.

Proof.
By contraposition. Suppose that $P(t)$ does not hold for some SN term $t$.

- then there exists $t_1$ with $t \rightarrow_\beta t_1$ such that $P(t_1)$ does not hold,
- then there exists $t_2$ with $t_1 \rightarrow_\beta t_2$ such that $P(t_2)$ does not hold,
- ...  

Contradiction: we have an infinite sequence of reductions starting from $t$. 


A term $t$ is **neutral** when it is not an abstraction:

$$t = t_1 t_2 \quad \text{or} \quad t = x$$
Neutral terms

A term $t$ is **neutral** when it is not an abstraction:

$$ t = t_1 \ t_2 \quad \text{or} \quad t = x $$

A neutral term does not interact with its context:

**Lemma**

Given terms $t$ and $u$ with $t$ neutral, the only possible reductions of $t \ u$ are

- $t \ u \rightarrow^\beta t' \ u$ with $t \rightarrow^\beta t'$,
- $t \ u \rightarrow^\beta t \ u'$ with $u \rightarrow^\beta u'$. 
Lemma

(CR1) If $t \in R_A$ then $t$ is strongly normalizing.

(CR2) If $t \in R_A$ and $t \xrightarrow{\beta} t'$ then $t' \in R_A$.

(CR3) If $t$ is neutral, and for every $t'$ such that $t \xrightarrow{\beta} t'$ we have $t' \in R_A$, then $t \in R_A$. 
Lemma

(CR1) If $t \in R_A$ then $t$ is strongly normalizing.

(CR2) If $t \in R_A$ and $t \rightarrow_\beta t'$ then $t' \in R_A$.

(CR3) If $t$ is neutral, and for every $t'$ such that $t \rightarrow_\beta t'$ we have $t' \in R_A$, then $t \in R_A$.

Proof.

By induction on $A$. If $A = X$ is a variable then:

(CR1) is true by definition of $R_X$.

(CR2) If $t \rightarrow_\beta t'$ and $t$ is SN then $t'$ is SN.

(CR3) If $t$ reduces only in SN terms then it is SN.
Reducibility candidates

Lemma

(CR1) If $t \in R_A$ then $t$ is strongly normalizing.

(CR2) If $t \in R_A$ and $t \rightarrow^\beta t'$ then $t' \in R_A$.

(CR3) If $t$ is neutral, and for every $t'$ such that $t \rightarrow^\beta t'$ we have $t' \in R_A$, then $t \in R_A$.

Proof.
Consider the case $A \rightarrow B$.

(CR1) Fix $t \in R_{A \rightarrow B}$.
We have $x \in R_A$ by (CR3), therefore $tx \in R_B$ and is thus SN by (CR1).
Any infinite reduction $t \rightarrow^\beta t_1 \rightarrow^\beta t_2 \rightarrow^\beta \ldots$ would induce an infinite reduction $tx \rightarrow^\beta t_1 x \rightarrow^\beta t_2 x \rightarrow^\beta \ldots$ with $tx \in R_B$, which is impossible. Thus $t$ is SN.
Lemma

(CR1) If \( t \in R_A \) then \( t \) is strongly normalizing.

(CR2) If \( t \in R_A \) and \( t \xrightarrow{\beta} t' \) then \( t' \in R_A \).

(CR3) If \( t \) is neutral, and for every \( t' \) such that \( t \xrightarrow{\beta} t' \) we have \( t' \in R_A \), then \( t \in R_A \).

Proof.
Consider the case \( A \rightarrow B \).

(CR2) Fix \( t \in R_{A \rightarrow B} \) with \( t \xrightarrow{\beta} t' \).
Given \( u \in R_A \), we have \( tu \in R_B \) and \( tu \xrightarrow{\beta} t'u \), therefore \( t'u \in R_B \) by (CR2).
Therefore \( t' \in R_{A \rightarrow B} \) by definition of \( R_{A \rightarrow B} \).
Reducibility candidates

Lemma

(CR1) If $t \in R_A$ then $t$ is strongly normalizing.

(CR2) If $t \in R_A$ and $t \rightarrow_{\beta} t'$ then $t' \in R_A$.

(CR3) If $t$ is neutral, and for every $t'$ such that $t \rightarrow_{\beta} t'$ we have $t' \in R_A$, then $t \in R_A$.

Proof.
Consider the case $A \rightarrow B$.

(CR3) Fix $t$ neutral satisfying the property.
Given $u \in R_A$, $u$ is SN by (CR1), and we can reason by induction on it.
Since $t$ is neutral the term $t u$ reduces either to

- $t' u$ with $t \rightarrow_{\beta} t'$: we have $t' \in R_{A \rightarrow B}$ and thus $t' u \in R_B$,
- $t u'$ with $u \rightarrow_{\beta} u'$: we have $u' \in R_A$ by (CR2), and therefore $t u' \in R_B$ by IH.

The term $t u$ is neutral and is thus in $R_B$ by (CR3) at type $B$. 

□
Lemma
Given a term $t$ and type $A$ and $B$, if $t \ u \in R_B$ for every $u \in R_A$, then $\lambda x. t \in R_{A \rightarrow B}$.

Proof.
We show that $(\lambda x. t) u \in R_B$ by induction on $(t, u)$. The term $(\lambda x. t) u$ can reduce to
Lemma
Given a term \( t \) and type \( A \) and \( B \), if \( t \ u \in R_B \) for every \( u \in R_A \), then \( \lambda x. t \in R_{A \rightarrow B} \).

Proof.
We show that \( (\lambda x. t) \ u \in R_B \) by induction on \( (t, u) \). The term \( (\lambda x. t) u \) can reduce to

- \( t[u/x] \): in \( R_B \) by hypothesis,
Lemma
Given a term $t$ and type $A$ and $B$, if $t\,u \in R_B$ for every $u \in R_A$, then $\lambda x.t \in R_{A \rightarrow B}$.

Proof.
We show that $(\lambda x.t)\,u \in R_B$ by induction on $(t, u)$. The term $(\lambda x.t)\,u$ can reduce to

- $t[u/x]$: in $R_B$ by hypothesis,
- $(\lambda x.t')u$ with $t \rightarrow^\beta t'$: in $R_B$ by induction hypothesis,
Lemma

Given a term $t$ and type $A$ and $B$, if $t \ u \in R_B$ for every $u \in R_A$, then $\lambda x. t \in R_{A\rightarrow B}$.

Proof.

We show that $(\lambda x. t)u \in R_B$ by induction on $(t, u)$. The term $(\lambda x. t)u$ can reduce to

- $t[u/x]$: in $R_B$ by hypothesis,
- $(\lambda x. t')u$ with $t \rightarrow_\beta t'$: in $R_B$ by induction hypothesis,
- $(\lambda x. t)u'$ with $u \rightarrow_\beta u'$: in $R_B$ by induction hypothesis.
Lemma

Given a term $t$ and type $A$ and $B$, if $t \ u \in R_B$ for every $u \in R_A$, then $\lambda x. t \in R_{A \rightarrow B}$.

Proof.

We show that $(\lambda x. t) u \in R_B$ by induction on $(t, u)$. The term $(\lambda x. t) u$ can reduce to

- $t[u/x]$: in $R_B$ by hypothesis,
- $(\lambda x. t') u$ with $t \rightarrow_\beta t'$: in $R_B$ by induction hypothesis,
- $(\lambda x. t) u'$ with $u \rightarrow_\beta u'$: in $R_B$ by induction hypothesis.

The term $(\lambda x. t) u$ is neutral and reduces to terms in $R_B$.

By (CR3), it belongs to $R_B$.

□
Finally, we would like to show that

**Theorem**
*Given* $t$ *such that* $\Gamma \vdash t : A$ *is derivable, we have* $t \in R_A$.

**Proof.**
By induction on the derivation of $\Gamma \vdash t : A$. 
Finally, we would like to show that

**Theorem**

*Given* $t$ such that $\Gamma \vdash t : A$ *is derivable, we have* $t \in R_A$.

**Proof.**

By induction on the derivation of $\Gamma \vdash t : A$.

- If the last rule is

  $\Gamma \vdash x : A$ (ax)

  Then $x$ is neutral and reduces only to terms in $R_A$ (it does not reduce).

  By (CR3) $x \in R_A$.  

Finally, we would like to show that

**Theorem**
*Given* $t$ *such that* $\Gamma \vdash t : A$ *is derivable, we have* $t \in R_A$.

**Proof.**
By induction on the derivation of $\Gamma \vdash t : A$.

- If the last rule is

  $\frac{\Gamma \vdash t : A \to B}{\Gamma \vdash t \ u : B}$  \hspace{1cm} \( \to_E \)

  By IH, we have $t \in R_{A \to B}$ and $u \in R_A$.

  Therefore $t \ u \in R_B$ by definition of $R_{A \to B}$. 

---

112
Finally, we would like to show that

**Theorem**

*Given* $t$ *such that* $\Gamma \vdash t : A$ *is derivable, we have* $t \in R_A$.

**Proof.**

By induction on the derivation of $\Gamma \vdash t : A$.

- If the last rule is

  \[ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. t : A \to B} \quad (\to I) \]

  By IH, we have $t \in R_A$. And... ????

\[ \square \]
Finally, we would like to show that

**Theorem**

*Given* $t$ such that $\Gamma \vdash t : A$ *is derivable*, we have $t \in R_A$.

**Proof.**

By induction on the derivation of $\Gamma \vdash t : A$.

- If the last rule is

  \[
  \Gamma, x : A \vdash t : B \\
  \Gamma \vdash \lambda x.t : A \rightarrow B \quad (\rightarrow I)
  \]

  By IH, we have $t \in R_A$. And... ????

We actually need a stronger induction hypothesis!
Proposition

Given $t$ such that $x_1 : A_1, \ldots, x_n : A_n \vdash t : A$ is derivable, and for every terms $u_i \in R_{A_i}$, we have $t[u/x] = t[u_1/x_1, \ldots, u_n/x_n] \in R_A$.

Proof.

By induction on the derivation of $\Gamma \vdash t : A$. 

Proposition

Given $t$ such that $x_1 : A_1, \ldots, x_n : A_n \vdash t : A$ is derivable, and for every terms $u_i \in R_{A_i}$, we have $t[u/x] = t[u_1/x_1, \ldots, u_n/x_n] \in R_A$.

Proof.

By induction on the derivation of $\Gamma \vdash t : A$.

- If the last rule is \[ \Gamma \vdash x_i : A \] (ax)

  The $x_i[u/x] = u_i \in R_{A_i}$.  


Proposition
Given $t$ such that $x_1 : A_1, \ldots, x_n : A_n \vdash t : A$ is derivable, and for every terms $u_i \in R_{A_i}$, we have $t[u/x] = t[u_1/x_1, \ldots, u_n/x_n] \in R_A$.

Proof.
By induction on the derivation of $\Gamma \vdash t : A$.

- If the last rule is

\[
\Gamma \vdash t_1 : A \rightarrow B \quad \Gamma \vdash t_2 : A
\]

\[
\frac{}{\Gamma \vdash t_1 \; t_2 : B} \quad (\rightarrow_E)
\]

By IH, we have $t_1[u/x] \in R_{A\rightarrow B}$ and $t_2[u/x] \in R_A$. Therefore $(t_1 \; t_2)[u/x] = (t_1[u/x])(t_2[u/x]) \in R_B$ by definition of $R_{A\rightarrow B}$. 

Reducibility candidates
Reducibility candidates

**Proposition**

Given $t$ such that $x_1 : A_1, \ldots, x_n : A_n \vdash t : A$ is derivable, and for every terms $u_i \in R_{A_i}$, we have $t[u/x] = t[u_1/x_1, \ldots, u_n/x_n] \in R_A$.

**Proof.**

By induction on the derivation of $\Gamma \vdash t : A$.

- If the last rule is

  \[
  \frac{\Gamma \vdash t : B}{\Gamma \vdash \lambda x.t : A \rightarrow B} \quad (\rightarrow_1)
  \]

  By IH, we have $(t[u/x])[v/x] = t[u/x, v/x] \in R_B$ for every $v \in R_A$.

  Therefore, $\lambda x.t \in R_{A \rightarrow B}$ by previous lemma.
Theorem
For every $t$ such that $\Gamma \vdash t : A$ is derivable, we have $t \in R_A$.

Proof.
Suppose $\Gamma = x_1 : A_1, \ldots, x_n : A_n$.
By (CR3), we have $x_i \in R_{A_i}$.
By previous proposition, we have $t = t[x_1/x_1, \ldots, x_n/x_n] \in R_A$. \qed
Theorem
For every term $t$ such that $\Gamma \vdash t : A$ is derivable, $t$ is strongly normalizable.

Proof.
We have $t \in R_A$, and thus $t$ is SN by (CR1).