Overview

(a) The untyped $\lambda$-calculus.
(b) The typed $\lambda$-calculus.
(c) The $\lambda$-Calculus in Agda.
(d) Logic with Implication
(e) Implicational Logic in Agda.
(f) More on the typed $\lambda$-calculus.
Basic idea of the \( \lambda \)-calculus:
We want to define functions “on the fly” (so called “anonymous functions”).

**Example:**
- We want to apply a function to all elements of a list.
- For instance, we want to upgrade a list of student numbers to one with one extra digit.
Greek Letters

- $\lambda$ is the Greek letter lambda.
- On the next slide you find the greek alphabet in upper case and lower case.
  - Some letters have two options for lower case, in which case the second is sometimes (but not always) pronounced by adding “var” in front, e.g. varphi for $\varphi$.
  - Some letters are indistinguishable from the Roman alphabet. So one cannot use them as separate mathematical symbols. I put brackets around them.
  - If one wants to transcribe the capital greek letter in Roman alphabet, one writes the lower case transcription and starts it with a captial, e.g. Gamma for $\Gamma$, Delta for $\Delta$. 
## The Greek Alphabet

<table>
<thead>
<tr>
<th>(A)</th>
<th>α</th>
<th>alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B)</td>
<td>β</td>
<td>beta</td>
</tr>
<tr>
<td>Γ</td>
<td>γ</td>
<td>gamma</td>
</tr>
<tr>
<td>Δ</td>
<td>δ</td>
<td>delta</td>
</tr>
<tr>
<td>(E)</td>
<td>ε</td>
<td>epsilon</td>
</tr>
<tr>
<td>(Z)</td>
<td>ζ</td>
<td>zeta</td>
</tr>
<tr>
<td>(H)</td>
<td>η</td>
<td>eta</td>
</tr>
<tr>
<td>Θ</td>
<td>θ, ϑ</td>
<td>(var)theta</td>
</tr>
<tr>
<td>(I)</td>
<td>ι</td>
<td>iota</td>
</tr>
<tr>
<td>(K)</td>
<td>κ</td>
<td>kappa</td>
</tr>
<tr>
<td>Λ</td>
<td>λ</td>
<td>lambda</td>
</tr>
<tr>
<td>(M)</td>
<td>μ</td>
<td>mu</td>
</tr>
<tr>
<td>(N)</td>
<td>ν</td>
<td>nu</td>
</tr>
<tr>
<td>Ξ</td>
<td>ξ</td>
<td>xi</td>
</tr>
<tr>
<td>(O)</td>
<td>ϒ</td>
<td>omikron</td>
</tr>
<tr>
<td>Π</td>
<td>π</td>
<td>pi</td>
</tr>
<tr>
<td>(P)</td>
<td>ρ, ρ</td>
<td>(var)rho</td>
</tr>
<tr>
<td>Σ</td>
<td>σ, ς</td>
<td>(var)sigma</td>
</tr>
<tr>
<td>(T)</td>
<td>τ</td>
<td>tau</td>
</tr>
<tr>
<td>Ψ</td>
<td>υ</td>
<td>upsilon</td>
</tr>
<tr>
<td>Φ</td>
<td>φ, ϕ</td>
<td>(var)phi</td>
</tr>
<tr>
<td>(X)</td>
<td>χ</td>
<td>chi</td>
</tr>
<tr>
<td>Ψ</td>
<td>ψ</td>
<td>psi</td>
</tr>
<tr>
<td>Ω</td>
<td>ω</td>
<td>omega</td>
</tr>
</tbody>
</table>
Example for Use of $\lambda$

- Can be done by multiplying each student number by 10.
- Let $f : \mathbb{N} \to \mathbb{N}, f(x) := x \cdot 10$.
- In many languages (e.g. C++, Perl, Python, Haskell) there is a pre-defined operation `map`, which takes a function $f$, and a list $l$, and applies $f$ to each element of the list. So for the above $f$ we have

\[
\text{map}(f, [210345, 345698, 296458]) = [2103450, 3456980, 2964580].
\]
Introduction to $\lambda$-Terms

- Often the $f$ is only needed once, and introducing first a new name $f$ for it is tedious.
- So one needs a short notation for "the function $f$, s.t. $f(x) = x \ast 10$".
- Notation is $\lambda x.x \ast 10$.
- So we have

$$\text{map}(\lambda x.x \ast 10, [210345, 345698, 296458]) = [2103450, 3456980, 2964580].$$

- In general $\lambda x.t$ stands for the function $f$ s.t. $f(x) = t$, where $t$ might depend on $x$.
  - above $t = x \ast 10$. 

One writes in functional programming usually $s \, t$ for the application of $s$ to $t$ instead of $s(t)$ as usual. This is used since we have often to apply a function several times, writing something like $f(r)(s)(t)$. Instead we write $f \, r \, s \, t$.

As indicated by the example, $r \, s \, t$ stands for $(r \, s) \, t$, in general $r_0 \, r_1 \, r_2 \cdots r_n$ stands for $(\cdots ((r_0 \, r_1) \, r_2) \cdots r_n)$.
We write $\lambda x, y \cdots$ for $\lambda x. \lambda y \cdots$.

Similarly for $\lambda x, y, z \cdots$.

E.g. $\lambda x, y, z. x (y z)$ stands for $\lambda x. \lambda y. \lambda z. x (y z)$.
Infix Operators

- We use + and * infix.
  The corresponding operators are written as (+), (*).
  - So \( x + y \) stands for \((+)(x)(y)\).
  - \( x * y \) stands for \((*)(x)(y)\).
- + and * will bind less than any non-infix constants.
  Therefore \( Sx + Sy \) stands for \((Sx) + (Sy)\).
- * binds more than +.
  Therefore \( x + y * z \) stands for \(x + (y * z)\),
  and \( Sx + Sy * z \) stands for \((Sx) + ((Sy) * z)\).
- In Agda we can achieve this by using the code

\[
\begin{align*}
\text{infixl 60} & \quad _+\_ \\
\text{infixl 80} & \quad _*\_ 
\end{align*}
\]
Scope of $\lambda x$.

- How do we read $\lambda x.x + 5$?
  - As $(\lambda x.x) + 5$?
  - Or as $\lambda x.(x + 5)$?

- **Convention:** The scope of $\lambda x$. is **as long as possible**.
  - So $\lambda x.x + 5$ reads as $\lambda x.(x + 5)$.
  - $\lambda x. (\lambda y.y) 5$ reads as $\lambda x. ((\lambda y.y) 5)$.
Scope of $\lambda x$.

- In $(\lambda x.x) \ 5$, the scope $\lambda x$. cannot be extended beyond the closing bracket.
  - So it is "x",
  - not "x) 5", which doesn’t make sense.
- In $f(\lambda x.x + 5, 3)$, the scope of $\lambda x$
  - is "x + 5",
  - not "x + 5, 3)", which doesn’t make sense.
- In $(\lambda x.x + 5) \ 3$, the scope of $\lambda x$
  - is $x + 5$
  - not $x + 5) \ 3$, which doesn’t make sense.
Sometimes, $\lambda x \ t$ (without a dot) is used, if one wants to have the scope of $\lambda x$ as short as possible.

- E.g. $\lambda x \ x \ y$ would denote $(\lambda x. x) \ y$.

- In this lecture we don’t use this notation.
Now we can define the terms of the untyped $\lambda$-calculus as follows:

- $\lambda$ terms are:
  - Variables $x$,
  - If $r$ and $s$ are $\lambda$-terms, so is $(r\ s)$.
  - If $x$ is a variable and $r$ is a $\lambda$-term, so is $(\lambda x.r)$.

- As usual brackets can be omitted, using
  - the above mentioned conventions about the scope of $\lambda x$,
  - and that $r\ s\ t$ is read as $(r\ s)\ t$. 
λ-Terms

Examples:
- \( \lambda x.x \),
- \( \lambda x.(\lambda y.y)\ x \),
- \( \lambda x.x\ x \),
- \( (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x) \),
- \( \lambda f.\lambda x.f\ (f\ x) \).
One might need additional constants to the language, then we have additionally:

- Any constant is a \( \lambda \)-term.

For instance,

- if \( c \) is a constant, then \( \lambda x. c \), \( (\lambda x. x) \ c \) are \( \lambda \)-terms;
- if \( (+) \) is a constant, then \( \lambda x. (+) \ x \ x \) is a \( \lambda \)-term.

For standard operators like \( +, \ast \), one has

- constants \( (+), (\ast) \),
- infix operations \( +, \ast \),
- and writes in infix notation
  - \( x + y \) instead of \( (+) \ x \ y \),
  - \( x \ast y \) instead of \( (\ast) \ x \ y \),
  - etc.
There are bound and of free variables in $\lambda$-terms:

- **Bound variables** are variables $x$, which occur in the scope of a $\lambda$-abstraction “$\lambda x.$”.
- **Free variables** are the other variables.

**Example:** In $\lambda x. x + y$,

- $x$ is bound (since in the scope of $\lambda x$),
- $y$ is free (since it is not in the scope of $\lambda y$).
Bound and Free Variables

- In \((\lambda y. y + z) \ y\),
  - the first occurrence of \(y\), \(y\) is bound,
  - the second occurrence of \(y\), \(y\) is free,
  - \(z\) is free.

- In \((\lambda y. ((\lambda z. z) \ y)) \ x\), we have
  - \(z\) is bound,
  - \(y\) is bound (in the scope of \(\lambda y\)),
  - \(x\) is free.
Note that being bound and free has something to do with an occurrence of a variable in a term, not with the variable itself.

So more precisely we should speak of occurrences of bound and free variables.

By the free variables of a term $t$ we mean the variables $x$ which have free occurrences, respectively, in $t$.

Similarly we define the bound variables of a term $t$. 
\(\alpha\)-Conversion

- We identify \(\lambda\)-terms, which only differ in the choice of the bound variables (variables abstracted by \(\lambda\)):
  - So \(\lambda x.x + 5\) and \(\lambda y.y + 5\) are identified.
    - Makes sense, since they both denote the same function \(f\) s.t. \(f(x) = x + 5\).
  - \((\lambda x.x + 5) 3 + 7\) and \((\lambda y.y + 5) 3 + 7\) are identified.
  - \(\lambda x.\lambda y.y\) and \(\lambda y.\lambda x.x\) are identified.

- This equality is called \(\alpha\)-equality, and the step from one term to another \(\alpha\)-equal term is called \(\alpha\)-conversion.

- So \(\lambda x.\lambda y.y\) and \(\lambda y.\lambda x.x\) are \(\alpha\)-equal, written as \(\lambda x.\lambda y.y =_\alpha \lambda y.\lambda x.x\).
**α-Conversion**

- Note that $\lambda x.\lambda x.x \equiv_\alpha \lambda y.\lambda x.x$.
  - The $x$ refers to the second lambda abstraction $\lambda x$, not the first one ($\lambda x.$).
  - Therefore, when changing the variable of the first $\lambda$-abstraction, $x$ remains unchanged.
Evaluation of $\lambda$-Terms

- How do we evaluate $(\lambda x.x \ast 10) \ 5$?
  - We first replace in $x \ast 10$, the variable $x$ by 5.
  - We obtain $5 \ast 10$.
  - Then we reduce this further, using other reduction rules (not introduced yet).
    Using suitable rules, we would reduce $5 \ast 10$ to 50.
  - In this Subsection we will look only at the pure $\lambda$-calculus without any additional reduction rules.
    There $(\lambda x.x \ast 10) \ 5$ reduces to $5 \ast 10$, which cannot be reduced any further.
Basics of the λ-Calculus

- In general, the result of applying \( \lambda x.t \) to \( r \), is obtained by substituting in \( t \) the variable \( x \) by \( r \).

  E.g.
  - \( (\lambda x. x + 10) \) 5 evaluates to \( 5 + 10 \),
    - If we substitute in \( x + 10 \) the variable \( x \) by 5, we obtain \( 5 + 10 \).
  - \( (\lambda x. x) \) "Student" evaluates to "Student".
    - If we substitute in \( x \), the variable \( x \) by "Student", we obtain "Student".
  - \( (\lambda x. x) (\lambda y. y) \) evaluates to \( \lambda y. y \).
    - If we substitute in \( x \) the variable \( x \) by \( \lambda y. y \), we obtain \( \lambda y. y \).
The last example shows that substitution by \( \lambda \)-terms can become more complicated, and we therefore instudy it in the following more carefully.

If \( t \) and \( s \) are \( \lambda \)-terms, \( t[x := s] \) denotes the result of substituting in \( t \) the variable \( x \) by \( s \), e.g.

- \((x + 10)[x := 5] \equiv 5 + 10\),
- \( x[x := "Student"] \equiv "Student" \),
- \( x[x := \lambda y.y] \equiv \lambda y.y \).
Substitution and Parentheses

- Substitution might introduce **additional parentheses**.
  - When we write a term e.g.
    \[ t \equiv 2 + 2 \ , \]
    what we really mean is that there are brackets around that term, e.g.
    \[ t = (2 + 2) \ . \]
    We omit the outer parentheses usually for convenience.
  - When substituting a term, the parentheses might become relevant.
E.g.

\[(x \times x)[x := 2 + 2] = (2 + 2) \times (2 + 2)\].

So we have to reintroduce in that example the brackets around 2 + 2 before carrying out the substitution.

If we did it naively (without reintroducing brackets), we would obtain

\[2 + 2 \times 2 + 2\]

which is different from

\[(2 + 2) \times (2 + 2)\].
Substitution and Bound Variables

- If we carry out a substitution in a $\lambda$-term, we have to be careful.
  - $(\lambda x.x + 7)[x := 3] \equiv \lambda x.x + 7$.
  - It doesn’t make sense to substitute the $x$ in $\lambda x.x + 7$, since $x$ is bound by $\lambda x$.
    - $x$ is a bound variable, which is not changed by the substitution.

- In general, in $s[x := t]$ we only substitute **free** occurrences of $x$ in $s$ by $t$.

- All bound occurrences remain unchanged.
More examples:

1. \((\lambda x. x)[x := "Student"]\) \equiv \lambda x. x.
   - The \(x\) in \(\lambda x. x\) is bound by \(\lambda x\), so no substitution is carried out.

2. \(((\lambda x.x) \ x)[x := "Student"]\) \equiv (\lambda x.x) "Student".
   - The first \(x\) is bound, so no substitution is carried out.
   - The second \(x\) is free, so substitution is carried out.

3. \((\lambda y.x + y)[x := 3]\) \equiv \lambda y.3 + y.
   - \(x\) in \(\lambda y.x + y\) is free, so it will be substituted by 3 in the above example.
Substitution and $\alpha$-Conversion

- When substituting in $\lambda$-terms, we sometimes have to carry out an $\alpha$-conversion first:
  - If we substitute in $\lambda y. y + x$, the variable $x$ by 3, we obtain correctly $\lambda y. y + 3$, the function $f$ s.t. $f(y) = y + 3$.
  - If we substitute in $\lambda y. y + x$, the variable $x$ by $y$, we should obtain a function $f$ s.t. $f(z) = z + y$.
  - If we did this naively, we would obtain $\lambda y. y + y$.
    - So the free variable $y$, which we substituted for $x$, has become, when substituting it in $\lambda y. y + x$, to a bound variable.
  - This is *not the correct way* of doing it.
Substitution and $\alpha$-Conversion

- The **correct way** is as follows:
  - First we $\alpha$-convert $\lambda y. y + x$, so that the binding variable $y$ is different from the free variable we are substituting $x$ by:
    Replace for instance $\lambda y. y + x$ by $\lambda z. z + x$.
  - Now we can carry out the substitution:
    
    \[
    (\lambda y. y + x)[x := y] =_{\alpha} (\lambda z. z + x)[x := y] \equiv \lambda z. z + y
    \]

- Similarly, we compute $(\lambda y. y + x)[x := y + y]$ as follows:

  \[
  (\lambda y. y + x)[x := y + y] =_{\alpha} (\lambda z. z + x)[x := y + y] \equiv \lambda z. z + (y + y)
  \]
In general, the substitution $t[x := s]$ is carried out as follows:

- $\alpha$-convert $t$ s.t.
  - if $x$ occurs in $t$ free and is in the scope of some $\lambda u$,
  - then $u$ doesn’t occur free in $s$.
  - In other words, $\alpha$-convert $t$ s.t. one never would substitute for $x$ the $s$ in such a way that one of the free variables of $s$ becomes bound.

- Then carry out the substitution.

Intuitively this means: $\alpha$-convert the bound variables in $s$ in such a way that never a variable, which is free in $s$ becomes bound when replacing in $t$ variable $x$ by $s$. 
Examples

- $(\lambda x.\lambda y.z)[z := x] =_\alpha (\lambda u.\lambda y.z)[z := x] \equiv (\lambda u.\lambda y.x)$,
- $(\lambda x.\lambda y.z)[z := y] =_\alpha (\lambda x.\lambda u.z)[z := y] \equiv (\lambda x.\lambda u.y)$,
- $(\lambda x.(\lambda y.y) z)[z := y] \equiv \lambda x.(\lambda y.y) y$.
  There is no problem in substituting the $z$ by $y$, since it is not in the scope of $\lambda y$.
- $(\lambda x.(\lambda y.y) y)[y := x] =_\alpha (\lambda u.(\lambda y.y) y)[y := x] \equiv \lambda u.(\lambda y.y) x$.
Examples

- \((\lambda x. z)[z := \lambda x. x] \equiv \lambda x. \lambda x. x.\)
  There is no problem with this substitution, since \(x\) does not occur free in \(\lambda x. x.\).
  Note that the \(x\) in \(\lambda x. \lambda x. x\) refers to the second \(\lambda\)-binding \(\lambda x\).

- \((\lambda x. z)[z := (\lambda x. x) \ x] =_{\alpha} (\lambda u. z)[z := (\lambda x. x) \ x] \equiv \lambda u.((\lambda x. x) \ x).\)
  Now \(x\) occurs free in \((\lambda x. x) \ x\) (the second occurrence is free), so we need to \(\alpha\)-convert it.
Substitution and $\alpha$-Conversion

- If you have problems understanding this, you can proceed as follows, and are on the safe side:
  - $\alpha$-convert $t$ so that all bound variable in $t$ are different from all free variables in $s$.
  - Then carry out the substitution.
- An unnecessary $\alpha$-conversion doesn’t hurt.
Writing $s[x := t]$ is sometimes a bit lengthy.

Therefore we will introduce the notion $s[x], s[t]$.

- $s[x]$ stands for a term $s$ possibly depending on a variable $x$.
  - E.g. $s[x] \equiv x$ or $s[x] \equiv a \cdot x$ for some constant $a$ or $s[x] \equiv \lambda y . x$.

- After we have introduced a term $s[x]$, we define $s[t]$ as the result of substituting in $s[x]$ the variable $x$ by $t$, e.g.

$$s[t] := s[x][x := t]$$
Examples:

- If $s[x] \equiv x$ then $s[t] \equiv t$.
- If $s[x] \equiv a \times$, then $s[t] \equiv a \times$.
- If $s[x] \equiv \lambda y.x$, then $s[y] \equiv (\lambda y.x)[x := y] = \lambda z.y$.
  - In the last example we had first to carry out $\alpha$-conversion, before we can carry out the substitution.

- We will usually not say what $s[x]$ actually is. Then it can essentially be treated as a term $s$ with a hole, for which $x$ is substituted (and in the original term with holes, $x$ doesn’t occur).
The notion of $\beta$-reduction is one step in the sense of evaluation of a $\lambda$-term to another term.

We first introduce the notion of a $\beta$-redex of a term $t$:

A subterm $(\lambda x.r)s$ of a $\lambda$-term $t$ is called a $\beta$-redex of $t$.

Examples:

- $(\lambda x.x)yz$ has $\beta$-redex $(\lambda x.x)y$.
  - Note that the bracketing is $((\lambda x.x)y)z$, not $(\lambda x.x)(yz)$.
- A redex can be the term itself: $(\lambda x.x)y$ has $\beta$-redex $(\lambda x.x)y$. 
A \(\lambda\)-term might have several \(\beta\)-redexes:

- E.g. In \((\lambda x.x\ x\ )\ (\ (\lambda y.y\ )\ z)\) we have
  - one redex \((\lambda x.x\ x\ )\ (\ (\lambda y.y\ )\ z)\)
  - and one redex \((\lambda y.y\ )\ z\).
A $\beta$-redex $(\lambda x.s) \ t$ can be reduced to $s[x := t]$.  
$s[x := t]$ is called the $\beta$-reduct of $(\lambda x.s) \ t$.

- The $\beta$-reduct of $(\lambda x.x + 10) \ 5$ is $5 + 10$,
- The $\beta$-reduct of $(\lambda x.x) \ "Student"$ is $"Student$.
- The $\beta$-reduct of $(\lambda x.x) \ (\lambda y.y)$ is $\lambda y.y$.

Using the “$s[t]$-notation”, the above can be more briefly written as

“$(\lambda x.s[x]) \ t$ reduces to $s[t]$.”
\( \beta \)-Reduction

\[ r \overset{\beta}{\longrightarrow} r', \text{ "} r \ \beta \text{-reduces to} \ r' \text{", or shorter} \ r \overset{\beta}{\longrightarrow} r', \text{ if} \ r' \text{ is obtained from} \ r \text{ by replacing one} \ \beta \text{-redex by its} \ \beta \text{-reduct.} \]

\textbf{Examples:}

\[ ((\lambda x. x + 5) \ 3) + 7 \overset{\beta}{\longrightarrow} (3 + 5) + 7, \text{ since} \]

\[ (\lambda x. x + 5) \ 3 \overset{\beta}{\longrightarrow} 3 + 5. \]

\[ \text{Assume we add a pairing operation} \ \langle s, t \rangle \text{ for the pair} \ s, t \text{ (will be introduced later), then} \]

\[ \langle (\lambda x. x + 5) \ 3, 7 \rangle \overset{\beta}{\longrightarrow} \langle 3 + 5, 7 \rangle, \]
Examples

- We can apply $\beta$-reduction under a $\lambda$ term as well:

$$\lambda x.((\lambda y.y + 5)\ 3) \rightarrow \lambda x.3 + 5.$$ 

- **Multiple redexes**: Because a $\lambda$-term might have several redexes, it might have two different reductions:
  - For instance
    - $(\lambda x.x\ x)\ ((\lambda y.y)\ z) \rightarrow ((\lambda y.y)\ z)\ ((\lambda y.y)\ z)$
    - $(\lambda x.x\ x)\ ((\lambda y.y)\ z) \rightarrow (\lambda x.x\ x)\ z.$
Examples of $\beta$-Reduction

\[(\lambda x.\lambda y.x) \; y \quad \rightarrow \quad (\lambda y.x)[x := y] \quad =_\alpha \quad (\lambda u.x)[x := y] \equiv \lambda u.y \]

\[(\lambda z.\lambda x.\lambda y.z) \; x \quad \rightarrow \quad (\lambda x.\lambda y.z)[z := x] \quad =_\alpha \quad (\lambda u.\lambda y.z)[z := x] \quad \equiv \lambda u.\lambda y.x \]

\[(\lambda z.\lambda x.(\lambda y.y) \; z) \; y \quad \rightarrow \quad (\lambda x.(\lambda y.y) \; z)[z := y] \equiv \lambda x.(\lambda y.y) \; y \quad \lambda x.(\lambda y.y) \; y \quad \rightarrow \quad \lambda x.y \]
Example (Longer Reduction)

► In the steps marked \(\equiv\) on the next slide, essentially the colouring is changed to mark the next \(\beta\)-redex.

► These steps are not very well visible on the printed black-and-white slides (where I use italic/boldface in order to denote the differences).

► This applies to future slides containing more complex \(\beta\)-reductions as well.

► Remember as well that

\[ \lambda x, y. t \]

abbreviates

\[ \lambda x. \lambda y. t \]
Example (Longer Reduction)

\[(\lambda x, y. x (x y)) (\lambda u, v. u (u v))\]

\[\equiv (\lambda x. \lambda y. x (x y)) (\lambda u, v. u (u v))\]

\[\rightarrow \lambda y. (\lambda u, v. u (u v)) ((\lambda u, v. u (u v)) y)\]

\[\equiv \lambda y. (\lambda u, v. u (u v)) ((\lambda u. \lambda v. u (u v)) y)\]

\[\rightarrow \lambda y. (\lambda u, v. u (u v)) (\lambda v. y (y v))\]

\[\equiv \lambda y. (\lambda u. \lambda v. u (u v)) (\lambda v. y (y v))\]

\[\rightarrow \lambda y. \lambda v. ((\lambda v. y (y v)) ((\lambda v. y (y v)) v))\]

\[\equiv \lambda y. \lambda v. ((\lambda v. y (y v)) ((\lambda v. y (y v)) v))\]

\[\rightarrow \lambda y. \lambda v. ((\lambda v. y (y v)) (y (y v)))\]

\[\equiv \lambda y. \lambda v. y (y (y v))\]

\[\equiv \lambda y, v. y (y (y v))\]
Examples of Non-Termination

- **Reproduction** (Term reduces to itself).
  Let \( \omega := \lambda x. x x \), \( \Omega := \omega \omega \). Then

  \[
  \Omega \equiv \omega \omega \equiv (\lambda x. x x) \omega \rightarrow \omega \omega \equiv \Omega .
  \]

- **Expansion** (Term reduct becomes bigger).
  Let \( \tilde{\Omega} := \lambda x. x (x x) \).
  Then

  \[
  \tilde{\Omega} (\tilde{\Omega} (\tilde{\Omega} (\tilde{\Omega}))) \rightarrow \ldots
  \]
Remark on Previous Slide

- Note that in the $\lambda$-term above $\lambda x. x \,(x \,x)$
  is to be read as $\lambda x. (x \,(x \,x))$
  and not as $(\lambda x.x) \,(x \,x)$
- The scope of $\lambda x.$ is always as long as possible.
By the **untyped $\lambda$-calculus** (short $\lambda$-calculus) we mean now
- the set of $\lambda$-terms, $T$ where $\alpha$-equivalent $\lambda$-terms are identified,
- together with $\beta$-reduction $\rightarrow_\beta$.

Therefore the $\lambda$-calculus forms a reduction system $(T, \rightarrow_\beta)$.

One might have the $\lambda$-calculus with additional constants.
- Without additional constants, the (untyped) $\lambda$-calculus is called the **pure (untyped) $\lambda$-calculus**.
For reduction systems we introduced notations $\rightarrow{}^*$, $a \leftrightarrow{}^* b$.

These notions can be used for the $\lambda$-calculus as well.

We define $r \equiv_{\beta} s$ ("$r$ and $s$ are $\beta$-equivalent") iff $r \leftrightarrow{}_{\beta}^* s$.

Since we identified $\alpha$-equivalent $\lambda$-terms, there can be arbitrary many $\alpha$-conversions in a chain for showing that $r \equiv_{\beta} s$.

Therefore we have $r \equiv_{\beta} r'$ iff there exists a sequence $s_0, \ldots, s_n, t'_0, \ldots, t'_n$ ($n = 0$ is possible) s.t.

\[ r \equiv s_0 =_{\alpha} t_0 \leftrightarrow{}_{\beta} s_1 =_{\alpha} t_1 \leftrightarrow{}_{\beta} s_2 =_{\alpha} t_2 \leftrightarrow{}_{\beta} \cdots \leftrightarrow{}_{\beta} s_n =_{\alpha} t_n \equiv r'. \]
Confluence of the $\lambda$-Calculus

- **Fact:** The $\lambda$-calculus is confluent (if we identify $\alpha$-equivalent terms).
- Therefore two $\lambda$ terms $r$ and $s$ are $\beta$-equivalent, iff there exits a term $t$ s.t. $r \xrightarrow{\beta}^* t$ and $s \xrightarrow{\beta}^* t$.
- **Example:** $((\lambda y.y) z) (((\lambda y.y) z)$ and $(\lambda x.x x) z$ are $\beta$-equivalent:
  - $((\lambda y.y) z) (((\lambda y.y) z)$ reduces in two steps to $z z$
  - and $(\lambda x.x x) z$ reduces in one step to the same term.
Note that this doesn’t give yet an easy way of determining whether $r \equiv_{\beta} s$ holds:

- One needs to find a $t$ s.t. $s \rightarrow^* t$ and $r \rightarrow^* t$.
- But simply reducing $r$ might never terminate.

Example:

- $(\lambda x. y) \Omega$ reduces in one step to $y$.
- So $(\lambda x. y) \Omega \equiv_{\beta} y$.
- However, by reducing $\Omega$ we obtain $\Omega$, therefore $(\lambda x. y) \Omega \rightarrow (\lambda x. y) \Omega$.
- So if we keep on following the second reduction, we will never find that this term is $\beta$-equivalent to $y$. 
Therefore we introduce the typed $\lambda$-calculus, which is strongly normalising, and in which therefore equality of $\lambda$-terms can be decided by determining $\alpha$-equality of normal forms.
(b) The Typed λ-Calculus

- Problem of the untyped λ-calculus:
  - Non-Termination, therefore $\equiv_\beta$ difficult to check.
    - In fact $\equiv_\beta$ is semi-decidable (r.e.), but not decidable (recursive).
  - Caused by the possibility of self-application, which allows to write essentially fully recursive programs.
  - Avoided by the **simply typed λ-calculus**, which is strongly normalising.
Main Idea of the Typed $\lambda$-Calculus

- $\lambda x.x + 5$ is a function,
  - taking an $x : \text{Int}$,
  - and returning $x + 5 : \text{Int}$.
- Therefore, we say that $(\lambda x.x + 5) : \text{Int} \rightarrow \text{Int}$.
  - In words, "$\lambda x.x + 5$ is of type Int arrow Int".
- In order to clarify the type of $x$, we write instead of $\lambda x.x + 5$
  \[
  \lambda x^{\text{Int}}.x + 5 .
  \]
  or
  \[
  \lambda (x : \text{Int}).x + 5 .
  \]
Basics of the Typed $\lambda$-Calculus

$\lambda x^{\text{Int}}.x + 5$ is
- only applicable to some $s : \text{Int}$,
- therefore not applicable to elements of other types, e.g. to “Student” (: String).

So
- $(\lambda x^{\text{Int}}.x + 5) 3$ is allowed,
- $(\lambda x^{\text{Int}}.x + 5)$ “Student” is not allowed.
The simple types used in the simply typed $\lambda$-calculus are defined inductively as follows:

- The ground type $o$ is a type.
- If $\sigma$, $\tau$ are types, so is $(\sigma \rightarrow \tau)$.

“Inductively” means that the set of simple types is the least set containing the ground type, and which closed under $\rightarrow$.

One sometimes modifies the set of ground types, especially when adding constants to the $\lambda$-terms.

- E.g. when using arithmetic expressions, one can say for instance that the ground types are $\text{Int}$ and $\text{Float}$.
- Then we talk about the simple types based on ground types $\text{Int}$ and $\text{Float}$.
Simple Types

- Usually we denote types by Greek letters,
  - e.g. $\alpha$ ("alpha"), $\beta$ ("beta"), $\gamma$ ("gamma"), $\sigma$ ("sigma"), $\tau$ ("tau").
- We omit brackets as usual using the convention that $\alpha \to \beta \to \gamma$ stands for $\alpha \to (\beta \to \gamma)$.
- Examples types:
  - $o_1 := o$,
  - $o_2 := o_1 \to o_1$
    $= o \to o$
  - $o_3 := o_2 \to o_2$
    $= (o \to o) \to o \to o$
  - $o_4 := o_3 \to o_3$
    $= (((o \to o) \to o \to o) \to (o \to o) \to o \to o)$
    which stands for
    $(((o \to o) \to (o \to o)) \to ((o \to o) \to (o \to o)))$
In order to make writing down such types easier, one can use sometimes the following abbreviations (these are non-standard abbreviations, and should be defined explicitly when using outside this lecture.

\[
o_2 := o \to o,
\]

\[
o_3 := o_2 \to o_2,
\]

etc.

So

\[
\text{an element of type } o_2 \text{ can be applied to an element of type } o \text{ and one obtains an element of type } o.
\]

\[
\text{an element of type } o_3 \text{ can be applied to an element of type } o_2 \text{ and one obtains an element of type } o_2.
\]

etc.
To determine the type of a term makes only sense, if we know the types of its variables.

For instance, in case of the $\lambda$-term $x \; y$, we could have

- $x : o_2$, $y : o$ and therefore $x \; y : o$,
- or $x : o_3$, $y : o_2$, and therefore $x \; y : o_2$.

Therefore we will give a type to $\lambda$ terms in a context, which determines the types of the variables.
A context is an expression of the form $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ where

- $x_i$ are variables,
- $\sigma_i$ are simple types,
  (when considering other type theories, $\sigma_i$ will be types of that theory).
- $n = 0$ is allowed, and we write $\emptyset$ for the empty context.
- Multiple occurrences of the same variable (even with different types) is allowed.
  - If we have two occurrences of the same variable, only the second occurrence counts.
  - E.g. in $x : \sigma, y : \tau, x : \rho$, “$x : \sigma$” is overridden by “$x : \rho$”, so the assumption in this context is $x : \rho$. 
Contexts

- **Examples**
  - $x : o, y : o2$ is a context.
  - $x : o2, x : o$ is a context in which we assume $x : o$.

- Note that contexts are **lists** of elements of the form $x : \sigma$, so the order matters.
  - In case of the simply typed $\lambda$-calculus, it wouldn’t make a difference to have as context unordered sets of expressions of the form $x : \sigma$ (as long as all variables in a context are different in order to avoid overriding).
  - However, when moving later to dependent type theory, the order of the expressions $x : \sigma$ will be relevant.
In the following, the capital Greek letters $\Gamma$ ("Gamma"), $\Delta$ ("Delta") denote contexts.

We write $\Gamma \Rightarrow s : \sigma$ for "in context $\Gamma$, $s$ has type $\sigma$".

Expressions of this form are called **judgements**.

Examples:

- $x : o2, y : o \Rightarrow x \, y : o$,
- $x : \text{Float} \rightarrow \text{Int}, y : \text{Float} \Rightarrow x \, y : \text{Int}$ (assuming ground types Float and Int),
- $x : o3, y : o2 \Rightarrow x \, y : o2$.

In case $\Gamma$ is empty, we write $s : \sigma$ instead of $\emptyset \Rightarrow s : \sigma$. 
If $\Gamma$, $\Delta$ are contexts, $\Gamma, \Delta$ denotes the concatenation of both contexts, e.g. if

- $\Gamma \equiv x : o, y : o2$,
- $\Delta \equiv z : o$

then

- $\Gamma, \Delta$ denotes $x : o, y : o2, z : o$,
- $\Delta, \Gamma$ denotes $z : o, x : o, y : o2$,
- $\Gamma, u : o$ denotes $x : o, y : o2, u : o$. 
Simply Typed $\lambda$-Calculus

**Definition** of the simply typed $\lambda$-terms, depending on a context, together with their type.

1. **Assumption.**
   Variables, occurring in the context, are terms having the type they have in the context:
   \[ \Gamma, \, x : \sigma, \, \Delta \Rightarrow x : \sigma \]

   **Condition on** $x$: $x$ must not occur in $\Delta$.
   
   ▶ Otherwise $x : \sigma$ is overridden by the assumption on $x$ in $\Delta$.
   
   ▶ Note that $\Gamma, \, x : \sigma, \, \Delta$ stands for any context, in which $x : \sigma$ occurs.
   
   ▶ **Explanation:** From the assumption $x : \sigma$ we can derive $x : \sigma$. 
Example (Assumption)

- We will illustrate the rules using a derivation of 

\[ y : o \to o \to o \Rightarrow \lambda x^o. y \ x : o \to o \to o \]

- In order to derive it we will need to derive first 

\[ y : o \to o \to o, x : o \Rightarrow y \ x : o \to o \]

- In order to derive that we use twice the assumption rule and obtain 

\[ y : o \to o \to o, x : o \Rightarrow y : o \to o \to o \]

and 

\[ y : o \to o \to o, x : o \Rightarrow x : o \]
Example (Overriding of Assum.)

- We have

\[ x : \sigma, x : \tau \implies x : \tau \]

but not

\[ x : \sigma, x : \tau \implies x : \sigma \]
2. **Application.**

If \( s \) is of type \( \sigma \rightarrow \tau \) and \( t \) of type \( \sigma \), depending on context \( \Gamma \), then \( s \ t \) is of type \( \tau \) under context \( \Gamma \):

\[
\Gamma \Rightarrow s : \sigma \rightarrow \tau \quad \Gamma \Rightarrow t : \sigma \\
\Gamma \Rightarrow s \ t : \tau
\]

**Explanation:**

- Assume we have \( s \) of type \( \sigma \rightarrow \tau \).
  - So \( s \) is a function, taking an \( x : \sigma \) and returning an element of type \( \tau \).
- Assume we have \( t \) is an element of type \( \sigma \).
- Then we can apply the function \( s \) to this \( t \), written as \( s \ t \), and obtain an element of type \( \tau \).
Example (Application)

- We continue with our derivation of

\[ y : o \rightarrow o \rightarrow o \Rightarrow \lambda x^o.y x : o \rightarrow o \rightarrow o \]

- We have already derived using the assumption rule

\[ y : o \rightarrow o \rightarrow o, x : o \Rightarrow y : o \rightarrow o \rightarrow o \]
\[ y : o \rightarrow o \rightarrow o, x : o \Rightarrow x : o \]

- Using the application rule we conclude:

\[ y : o \rightarrow o \rightarrow o, x : o \Rightarrow y : o \rightarrow o \rightarrow o \]
\[ y : o \rightarrow o \rightarrow o, x : o \Rightarrow x : o \]

\[ y : o \rightarrow o \rightarrow o \Rightarrow y x : o \rightarrow o \]

\[ (Ap) \]

Note that \[ o \rightarrow o \rightarrow o \equiv o \rightarrow (o \rightarrow o). \]
3. **Abstraction.**

If $t$ is a term of type $\tau$, depending on context $\Gamma, x : \sigma$, then $\lambda x^\sigma.t$ is a term of type $\sigma \to \tau$ depending on context $\Gamma$:

\[
\frac{\Gamma, x : \sigma \Rightarrow t : \tau}{\Gamma \Rightarrow \lambda x^\sigma.t : \sigma \to \tau} \tag{Abs}
\]

▶ **Explanation:**

▶ If we have under assumption $x : \sigma$ shown that $t : \tau$, then we can form a new $\lambda$-term by binding that $x$, and form $\lambda x^\sigma.t$.

▶ The result is a function taking as input $x : \sigma$ and returning $t : \tau$, so we obtain an element of $\sigma \to \tau$. 
Example (Abstraction)

- We finish our derivation of

\[ y : o \to o \to o \Rightarrow \lambda x^o . y \ x : o \to o \to o \]

- We have already derived

\[
\frac{y : o \to o \to o, x : o \Rightarrow y : o \to o \to o}{y : o \to o \to o, x : o \Rightarrow y \ x : o \to o} \quad \text{(Ap)}
\]

- Using abstraction we obtain:

\[
\frac{y : o \to o \to o, x : o \Rightarrow y : o \to o \to o}{y : o \to o \to o, x : o \Rightarrow y \ x : o \to o} \quad \text{(Ap)}
\]

\[
\frac{x : o \Rightarrow x^o \Rightarrow x : o \to o}{y : o \to o \to o \Rightarrow \lambda x^o . y \ x : o \to o \to o} \quad \text{(Abs)}
\]

(Note that \( o \to o \to o \equiv o \to (o \to o) \).)
We had three rules:

1. $\Gamma, x : \sigma, \Delta \Rightarrow x : \sigma$ (where $x$ must not occur in $\Delta$).

2. $\Gamma \Rightarrow s : \sigma \rightarrow \tau \quad \Gamma \Rightarrow t : \sigma$ 
   $\quad \Gamma \Rightarrow s \ t : \tau$ (Ap)

3. $\Gamma, x : \sigma \Rightarrow t : \tau$
   $\Gamma \Rightarrow \lambda x^{\sigma}. t : \sigma \rightarrow \tau$ (Abs)
(1) \( \Gamma, x : \sigma, \Delta \Rightarrow x : \sigma \)

is a special kind of rule, an axiom.

Axioms derive typing judgements without having to prove something first (no premises).

(2) The next rule is a genuine rule:

\[
\frac{\Gamma \Rightarrow s : \sigma \rightarrow \tau \quad \Gamma \Rightarrow t : \sigma}{\Gamma \Rightarrow s \ t : \tau} \quad (Ap)
\]

It expresses:

- Whenever we have derived \( \Gamma \Rightarrow s : \sigma \rightarrow \tau \)
  - (for arbitrary context \( \Gamma \), types \( \sigma, \tau \), term \( s \))
- and whenever we derived \( \Gamma \Rightarrow t : \sigma \)
  - (for the same \( \Gamma, \sigma \), but arbitrary term \( t \)),
- then we can derive \( \Gamma \Rightarrow s \ t : \tau \).
(3) The next rule is similar:

\[
\frac{\Gamma, x : \sigma \Rightarrow t : \tau}{\Gamma \Rightarrow \lambda x^{\sigma}. t : \sigma \rightarrow \tau} \quad \text{(Abs)}
\]

It expresses:

- Whenever we have derived \( \Gamma, x : \sigma \Rightarrow t : \tau \)
  - (for arbitrary context \( \Gamma \), types \( \sigma, \tau \), variable \( x \) and term \( t \)),
  then we can derive from this \( \Gamma \Rightarrow \lambda x^{\sigma}. t : \sigma \rightarrow \tau \).
Derivations

- Using rules we can derive more complex judgements:
  - We start with axioms, and use rules with premises in order to derive further judgements.

- **Example 1:**
  (Note that $o2 = o \rightarrow o$).

\[
\frac{x : o \Rightarrow x : o}{\lambda x^o . x : o2} \quad \text{(Abs)}
\]
Example 2

\[
x : o^2, y : o \Rightarrow x : o^2 \quad x : o^2, y : o \Rightarrow y : o \\
\quad x : o^2, y : o \Rightarrow x \, y : o \\
\quad x : o^2 \Rightarrow \lambda y^o \cdot x \, y : o^2 \\
\quad \lambda x^{o^2} \cdot \lambda y^o \cdot x \, y : o^3
\]

(\text{Ap})

(Abs)

(Abs)

Note that we have the following dependencies in the derived $\lambda$-term:

\[
(\lambda x^{o^2} \cdot \lambda y^o \cdot x \, y) : o^2 \rightarrow o^2 = o^3
\]

Observe how these dependencies correspond to the derivation above.
\(\beta\)-Reduction

- \(\beta\)-reduction for typed \(\lambda\)-terms is defined as for untyped \(\lambda\)-terms.
  - One has only to carry around the types as well.
  - Formally we have
    \[
    (\lambda x^{\sigma}. t) s \rightarrow t[x := s]
    \]
    or using the alternative notation for typed \(\lambda\)-terms
    \[
    (\lambda(x: \sigma). t) s \rightarrow t[x := s]
    \]
  - And as before \(\beta\)-reduction can be applied to any subterm.
    - A subterm \((\lambda x^{\sigma}. t) s\) of a term \(s\) is called a \(\beta\)-redex of \(s\).
Example

(Changes of colour not well visible in black-and-white copies).

\[
(\lambda x^3 \cdot \lambda y^2 \cdot x (x \ y)) (\lambda x^2 \cdot \lambda y^0 \cdot x (x \ y))
\]

\[\rightarrow \quad \lambda y^0 \cdot (\lambda x^2 \cdot \lambda y^0 \cdot x (x \ y)) ((\lambda x^2 \cdot \lambda y^0 \cdot x (x \ y)) \ y)
\]

\[\equiv \quad \lambda y^0 \cdot (\lambda x^2 \cdot \lambda y^0 \cdot x (x\ y)) ((\lambda x^2 \cdot \lambda y^0 \cdot x (x \ y)) \ y)
\]

\[\equiv_{\alpha} \quad \lambda y^0 \cdot (\lambda x^2 \cdot \lambda y^0 \cdot x (x \ y)) ((\lambda x^2 \cdot \lambda z^0 \cdot x (x \ z)) \ y)
\]

\[\rightarrow \quad \lambda y^0 \cdot (\lambda x^2 \cdot \lambda y^0 \cdot x (x \ y)) (\lambda z^0 \cdot y (y \ z))
\]

\[\equiv \quad \lambda y^0 \cdot (\lambda x^2 \cdot \lambda y^0 \cdot x (x \ y)) (\lambda z^0 \cdot y (y \ z))
\]

\[\equiv_{\alpha} \quad \lambda y^0 \cdot (\lambda x^2 \cdot \lambda u^0 \cdot x (x \ u)) (\lambda z^0 \cdot y (y \ z))
\]

\[\rightarrow \quad \lambda y^0 \cdot \lambda u^0 \cdot (\lambda z^0 \cdot y (y \ z)) (((\lambda z^0 \cdot y (y \ z)) \ u)
\]

\[\equiv \quad \lambda y^0 \cdot \lambda u^0 \cdot (\lambda z^0 \cdot y (y \ z)) (((\lambda z^0 \cdot y (y \ z)) \ u)
\]

\[\rightarrow \quad \lambda y^0 \cdot \lambda u^0 \cdot (\lambda z^0 \cdot y (y \ z)) (y \ (y \ u))
\]

\[\equiv \quad \lambda y^0 \cdot \lambda u^0 \cdot (\lambda z^0 \cdot y (y \ z)) (y \ (y \ u))
\]

\[\rightarrow \quad \lambda y^0 \cdot \lambda u^0 \cdot y (y \ (y \ u)))
\]
Theorem

- As for the untyped \( \lambda \)-calculus, the simply typed \( \lambda \)-calculus is **confluent**.
- The simply typed \( \lambda \)-calculus is **strongly normalising**.
- Therefore every typed \( \lambda \)-term has a unique normal form, which can be obtained by \( \beta \)-reducing the term by choosing arbitrary \( \beta \)-redexes.
- Furthermore, two \( \lambda \)-terms are \( \beta \)-equal, if their normal forms are equal (up to \( \alpha \)-conversion).
Agda is based on dependent type theory.
This extends the simply typed $\lambda$-calculus.
In Agda one writes $A \rightarrow C$ for the nondependent function type. We write on our slides $\rightarrow$ instead of $\rightarrow$.

I tend to use capital letters instead of Greek letters for types in Agda. One could of course use as well “alpha”, “beta”, “gamma”, or (using special symbols) $\alpha$, $\beta$, $\gamma$ instead.
In Agda, there needs to be a blank before and after $\rightarrow$, but there should be no blank between $-$ and $\succ$.

$A \rightarrow$ without a blank in between is understood as an identifier with name $A \rightarrow$.

$\rightarrow A$ without a blank in between is understood as an identifier with name $\rightarrow A$.

Only brackets “(”, “{”, “)”, “}”, the symbol “=”, blanks (and possibly some other symbols not discovered yet by A. Setzer) break identifiers.
\textbf{\(\lambda\)-Terms in Agda}

- In Agda one writes \(\lambda(x : A) \to r\) for \(\lambda(x : A).r\).
- When presenting Agda code we will write \(\lambda(x:A) \to r\) for the above, so \(\lambda\) means \(\backslash\) and \(\to\) means \(\to\) in real Agda code.
- When reasoning in type theory itself (outside Agda), we use standard type theoretic notation \(\lambda(x : A).r\).
- We can in Agda often omit the type of \(x\), and write simply

\[
\lambda x \to r
\]

instead of

\[
\lambda(x : A) \to r
\]
Blanks in \((x : A) \rightarrow r\)

- In \((x : A) \rightarrow r\),
  - there needs to be a blank before and after the "\:".
    - \(x\): without a blank in between is considered by Agda as an identifier "\(x:\)".
    - \(A\) without a blank in between is considered by Agda as an identifier "\(A:\)".
  - There needs to be a blank between \(-\rightarrow\) and \(r\).
Notations in Agda

- As an abbreviation, one writes

\[ \lambda(a \ a' : A) \to \cdots \]

(note that there is no comma between \(a\) and \(a'\))

instead of

\[ \lambda(a : A) \to \lambda(a' : A) \to \cdots \]

and

\[ \lambda a \ a' \to \cdots \]

instead of

\[ \lambda a \to \lambda a' \to \cdots \]
Application in Agda

- **Application** has the same syntax as in the rules of dependent type theory: Assume we have derived

\[
\begin{align*}
  f & : A \rightarrow B \\
  a & : A 
\end{align*}
\]

Then we can conclude \( f \ a : B \).

- And \( \alpha \)- and \( \beta \)-equivalent terms are identified.
  - In Agda,
    \[
    (\lambda x \rightarrow x) \ a = a 
    \]
  - So if \( B \ a \) is a type depending on \( a \), and we have \( b : B \ a \) then we have as well
    \[
    b : B (\lambda x \rightarrow x) \ a
    \]
Postulate

- In Agda one has no predefined types, all types have to be defined explicitly (e.g. the type of natural numbers, the type of Booleans, etc.).
- In order to obtain ground types with no specific meaning (like \( o \) above), we have to postulate such types, (or use packages as introduced later).
- In Agda the lowest type level, which corresponds to types in the simply typed \( \lambda \)-calculus, is called for historic reasons \( \text{Set} \).
- So in order to introduce a ground type \( A \) we write:

  \[
  \text{postulate } A : \text{Set}
  \]
We can now introduce other constants. For instance, in order to introduce a function from $A$ to $B$ where $A$ and $B$ are ground types, and an element of type $A$, we write the following:

```agda
postulate A : Set
postulate B : Set.
postulate f : A → B.
postulate a : A.
```

See examplePostulate1.agda
Basic $\lambda$-Terms

postulate $A : \text{Set}$
postulate $B : \text{Set}$.
postulate $f : A \rightarrow B$.
postulate $a : A$.

- Assuming the above postulates, we can now introduce new terms.
- We have to give a name and a type to each new definition.
- **Example:**
  Using the above postulates, we can define $b := f \ a : B$ as follows:

  $b : B$
  $b = f \ a$

  Please note that blanks around “=”. 
Basic $\lambda$-Terms

postulate $A : \text{Set}$
postulate $B : \text{Set}$. 
postulate $f : A \rightarrow B$.
postulate $a : A$.

$b : B$

\[ b = f \ a \]

- We can as well introduce \( g := \lambda x^A.x : A \rightarrow A \) as follows:

\[ g : A \rightarrow A \]

\[ g = \lambda x \rightarrow x \]

- Note that there needs to be blanks around “\(=\)”. 

See examplePostulate2.agda
postulate $A : \text{Set}$
postulate $B : \text{Set}$.
postulate $f : A \rightarrow B$.
postulate $a : A$.

- Instead of defining $\lambda$-terms by using $\lambda$ directly, it is usually more convenient to use a notation of the following kind:

\[
g : A \rightarrow A \\
g a = a
\]

- Note that in the above example, the local $a$ overrides the global $a$.

See examplePostulate3.agda
The two ways of introducing functions are equivalent. One can check this by defining two versions:

- **Postulate:**
  - \( A : \text{Set} \)
  - \( g : A \rightarrow A \)
  - \( g = \lambda (a : A) \rightarrow a \)

- **Version 2:**
  - \( g' : A \rightarrow A \)
  - \( g' a = a \)
Equivalence of the two Notations

- We postulate now a predicate on $A \rightarrow A$, in order to check whether $g$ and $g'$ are the same:

  \[ \text{postulate } P : (A \rightarrow A) \rightarrow \text{Set} \]

- If we define now

  \[ f : P \; g \rightarrow P \; g' \]
  \[ f \; x = x \]

  then $f$ is (since we don’t know anything about $P$) only type correct, if $g = g'$.

- The above code type checks, so for Agda we have $g$ and $g'$ are the same.

exampleEquivalenceLambdaNotations1.agda
- Notation in Agda

- In most cases, it is easier to use the second way of introducing \( \lambda \)-terms.

- However, \( \lambda \)-notation allows to introduce anonymous functions (i.e. functions without giving them names):

  A typical example from functional programming is the map function, which applies a function to each element of a list:

  \[
  \text{map } S \ (\text{two} :: (\text{three} :: []))
  \]

  The result is

  \[
  (\text{three} :: (\text{four} :: []))
  \]
Here the elements of NatList are

- [] denoting the empty list,
- and if \( n : \mathbb{N}, \, l : \text{NatList} \), then \( n :: l : \text{NatList} \).

See `exampleMapAppliedToList.agda`. 
Assume the following Agda code

\begin{verbatim}
postulate A : Set
postulate B : Set
postulate f : A → B
postulate a : A
b  :  B
b  =  {! !}
\end{verbatim}

Assume that we don’t know what to insert. We only guess that it has to be of the form \( f \) applied to some arguments.

We can see this since the result type of \( f \) is \( B \) (\( f : A \rightarrow B \)).
Refinement

postulate \( A : \text{Set} \)
postulate \( B : \text{Set} \)
postulate \( f : A \rightarrow B \)
postulate \( a : A \)
\( b : B \)
\( b = \{! !\} \)

- Then we can insert \( f \) into this goal and use menu `Refine (C-c C-r)`
- The system shows \( b = f \{! !\} \).
- We can ask for the type of the new goal \( \{! !\} \), using goal menu `Goal-type C-c C-t`, and obtain \( \{! !\} : A \)
postulate $A : \text{Set}$
postulate $B : \text{Set}$
postulate $f : A \rightarrow B$
postulate $a : A$
postulate $b : B$

$\text{refine: } \quad b = f \{! !\}$

▶ Now we can solve this goal by filling in $a$ and using refine: $f \ a : B$. 

exampleSimpleDerivation1.agda
Introducing New Types

- In the $\lambda$-calculus, we introduced abbreviations for types, like $o^2 = o \to o$
- We can do the same in Agda ([exampleTypeAbbreviations.agda]):

  ```agda
  postulate A : Set
  A2  :  Set
  A2  =  A \to A
  A3  :  Set
  A3  =  A^2 \to A^2
  a2  :  A2
  a2  =  \lambda x . x
  a3  :  A3
  a3  =  \lambda x . x
  ```
Introducing New Types

postulate \( A : \text{Set} \)

\[
A_2 : \text{Set} \\
A_2 = A \rightarrow A \\
a_2 : A_2 \\
a_2 = \lambda(x : A) \rightarrow x
\]

- In the above example we have that the type of \( a_2 \) is as well \( A \rightarrow A \), since both types are equal: Although \( a_2 \) is of type \( A_2 \) instead of \( A \rightarrow A \), we can define

\[
a_2' : A \rightarrow A \\
a_2' = a_2
\]
Introducing New Types

- We can as well check that $A \rightarrow A$ and $A^2$ are the same by applying main menu Compute normal form C-c C-n to $A^2$
  - We obtain $A \rightarrow A$. 

CS_336/CS_M36/CS_M46 Sect. 3 (c)
In Agda, rules are implicit.

The rule

\[
\frac{f : A \to B \quad a : A}{f \ a : B} \quad (Ap)
\]

corresponds to the following:

Assume we have introduced:

\( f : A \to B, \ a : A. \)

and want to solve the goal

\[
\begin{align*}
b & : B \\
b & = \{! \ \} 
\end{align*}
\]

exampleSimpleDerivation2.agda
Then we can fill this goal by typing in \( f \ a \):
\[
b = \{! f \ a !\}
\]
If we then choose goal-menu \textbf{Refine (C-c C-r)}, the system shows:
\[
b = f \ a.
\]
Let expressions in Agda

- When introducing elements of more complicated types, let expressions are often useful. They allow to introduce temporary variables.

- Let-expressions have the form

```
let a_1 : A_1
    a_1  = s_1
    a_2 : A_2
    a_2  = s_2
    ...  
    a_n : A_n
    a_n  = s_n
in  t
```
This means that we introduce new local constants
\[ a_1 : A_1 \text{ s.t. } a_1 = s_1, \]
\[ a_2 : A_2 \text{ s.t. } a_2 = s_2, \]
\[ \ldots, \]
\[ a_n : A_n \text{ s.t. } a_n = s_n, \]
which can now be used locally.

\( s_i \) can refer to all \( a_j \) defined before, but not to \( a_i \) itself, i.e. it can refer to \( a_0, \ldots, a_{i-1} \).
The following function computes \((n + n) \times (n + n)\) for \(n : \mathbb{N}\):

\[
f \quad : \quad \mathbb{N} \to \mathbb{N}
\]

\[
f \ n \quad = \quad \text{let} \ m \ : \ \mathbb{N}
\]

\[
m \quad = \quad n + n
\]

\[
in \ m \times m
\]

See exampleLetExpression.agda

Note that this version is more efficient than the function computing directly \((n + n) \times (n + n)\):

- Using \texttt{let}, \(n + n\) is computed only once,
- without \texttt{let}, we have to compute it twice.
Example

As an example we define, assuming \( A : \text{Set} \) as a postulate, a function

\[
f : ((A \to A) \to A) \to A
\]

We start with the goal

\[
f : ((A \to A) \to A) \to A
\]

\[
f = \{! !\} \]
Example

\[ f : ((A \to A) \to A) \to A \]
\[ f = \{! !\} \]

- We know that the first argument of \( f \) is an element of type \((A \to A) \to A\).
- We call this argument for better readability of the code \( a\neg a\neg a\).
- We obtain

\[ f : ((A \to A) \to A) \to A \]
\[ f \ a\neg a\neg a = \{! !\} \]
Example

- We can use $a - a - a$ in order to obtain $a$ provided we have defined some function $a - a : A \rightarrow A$.
- Therefore we first define in an auxiliary definition $a - a : A \rightarrow A$.
- In this example we could do this as a global definition, but will use here a let expression instead.
- We deactivate Agda (using main menu **De-activate Agda (C-c C-x C-d)**),
- replace the goal by a let expression,
- and then load the buffer again.
Example

\[ f : ((A \rightarrow A) \rightarrow A) \rightarrow A \]

\[ f \ a-a-a = \text{let } a-a : \{! \} \]
\[ a-a = \{! \} \]
\[ \text{in } \{! \} \]

- We type into the first goal the type \( A \rightarrow A \) of the variable \( a-a \) and use goal menu **Refine** or **Give** and obtain

\[ f : ((A \rightarrow A) \rightarrow A) \rightarrow A \]
\[ f \ a-a-a = \text{let } a-a : A \rightarrow A \]
\[ a-a = \{! \} \]
\[ \text{in } \{! \} \]
Example

- In the first goal, we know that this might be solved by using a \( \lambda \)-expression.
- We type into this goal

\[
\lambda a \rightarrow ?
\]

and use refine or give and obtain

\[
f \quad : \quad ((A \rightarrow A) \rightarrow A) \rightarrow A
\]

\[
f \ a-\ a-\ a \quad = \quad \text{let} \ \ a-\ a \ : \ A \rightarrow A
\]

\[
a-\ a \quad = \quad \lambda a \rightarrow \{! \quad !\}
\]

\[
\text{in} \ \{! \quad !\}
\]
Example

We solve the first goal by typing in \( a \) and using \textbf{Refine} and have completed the let-expression:

\[
\begin{align*}
  f & : ((A \to A) \to A) \to A \\
  f \ a\ a\ a & = \text{let } a\ a : A \to A \\
  & \quad a\ a = \lambda a \to a \\
  & \quad \text{in } \{! !\}
\end{align*}
\]
Example

- We can solve the remaining (main) goal by applying the variable \( a \rightarrow a \) to \( a \rightarrow a \). We type those values into the remaining goal and use **Give** or **Refine** and obtain:

\[
\begin{align*}
    f & : ( (A \rightarrow A) \rightarrow A ) \rightarrow A \\
    f \ a \rightarrow a & = \text{let } a \rightarrow a & : & A \rightarrow A \\
    & a \rightarrow a & = & \lambda a \rightarrow a \\
    & \text{in } a \rightarrow a \ a \rightarrow a
\end{align*}
\]

- See exampleLetExpression2.agda
(d) Logic with Implication
When considering the example of a sorted list, we have seen already that

- formulas (e.g. predicates) can be considered as types,
- where elements of such types are verifications that the formula holds ($\approx$ is true).
  - So elements of this type are proofs that the formula holds.

The principle to identify propositions (i.e. formulae) with types is called propositions as types.

So

- Sorted \( l \) will be a type,
- \( p : \text{Sorted} \ l \) will be a witness (proof) that Sorted \( l \) holds.
Constructive Logic

- If $p : \text{Sorted } l$ holds, then $l$ should be sorted.
- If we have a proof $p : \neg (\text{Sorted } l)$ then $l$ should be not sorted.
  - Negation $\neg$ will be introduced later.
- If we know neither that $p : \text{Sorted } l$ nor that $p : \neg (\text{Sorted } l)$, then we know neither that $l$ is sorted nor that $l$ is not sorted.
  - Happens e.g. if $l$ is a variable.
  - For certain closed quantified formula, like $A$ expressing that for all natural numbers $n$ a certain formula hold, it might be the case that we can neither determine a $p : A$ nor a $p : \neg A$. 
Provably true

Neither provable nor provably false

Provably false

Picture
If we postulate $A : \text{Set}$, we can consider $A$ as an **atomic formula** (i.e. formula which cannot be decomposed further).

- This is similar to a **propositional variable** (such as $A, B, C$ in $((A \land B) \lor C) \rightarrow A$).
- Formulae like $((A \land B) \lor C) \rightarrow A$ might be generally true (e.g. $A \rightarrow A$), or might be true if certain of its propositional variables are provably true and others are provably false (e.g. $A \lor C$).

If we postulate $A : \text{Set}$, we assume nothing about provability of $A$, since we assume nothing about the elements of $A$.

If we postulate additionally $a : A$, we postulate that $A$ is true.
Example

We postulate

- a set of persons
- a predicate “is student” on the set of persons,
- that John, Mary as persons,
- that Mary is a student:

\[
\begin{align*}
\text{postulate} & \quad \text{Person} & : & \quad \text{Set} \\
\text{postulate} & \quad \text{john} & : & \quad \text{Person} \\
\text{postulate} & \quad \text{mary} & : & \quad \text{Person} \\
\text{postulate} & \quad \text{IsStudent} & : & \quad \text{Person} \rightarrow \text{Set} \\
\text{postulate} & \quad \text{maryIsStudent} & : & \quad \text{IsStudent} \text{ mary}
\end{align*}
\]
Proofs in dependent type theory will have always a constructive meaning.

In case of implication the constructive meaning of a proof of $a \rightarrow b : A \rightarrow B$ will be:

- It is a function, which from a proof of $A$ determines a proof of $B$.
  - This is what is meant by $A \rightarrow B$: if $A$ holds, i.e. if we have a proof of $A$, then $B$ holds, i.e. we have a proof of $B$.
- So $a \rightarrow b : A \rightarrow B$ is a function mapping proofs of $A$ to proofs of $B$.
- This is nothing but the function type $A \rightarrow B$. 
Example 1 (Implication)

- $\lambda(x : A). x : A \rightarrow A$ is a proof that $A \rightarrow A$ holds:
  - it takes a proof $x : A$ and maps it to the proof $x : A$ of $A$.
- In ordinary logic, this $\lambda$-term corresponds to the following proof that $A \rightarrow A$ holds:
  - Assume $A$.
  - Then $A$ holds.
  - Therefore $A \rightarrow A$ holds.
Example 2 (Implication)

\( \lambda(x : A \rightarrow B).\lambda(y : A).x \, y \) is a proof of \((A \rightarrow B) \rightarrow A \rightarrow B\):

- Assume a proof \( x : A \rightarrow B \).
  - I.e. assume a function \( x \) which maps proofs of \( A \) to proofs of \( B \).
- Assume a proof \( y : A \).
- Then we obtain a proof \( x \, y : B \).
  This proof is obtained by
    - taking the proof \( x : A \rightarrow B \), which is a function mapping proofs of \( A \) to proofs of \( B \),
    - applying it to the proof \( y : A \),
    - then one obtains the proof \( x \, y \) of \( B \).
Example 2 (Implication)

\[(\lambda(x : A \to B).\lambda(y : A).x \; y) : (A \to B) \to A \to B\]

- In ordinary logic, the \(\lambda\)-type just introduced corresponds to the following derivation of \((A \to B) \to A \to B\):
  - Assume \(A \to B\).
  - Assume \(A\).
  - Then from \(A \to B\) and \(A\) we obtain \(B\).
  - This shows \((A \to B) \to A \to B\) holds.
We could have given the following shorter proof of 

\[(A \rightarrow B) \rightarrow A \rightarrow B:\]

\[\lambda(x : A \rightarrow B).x : (A \rightarrow B) \rightarrow (A \rightarrow B)\]

Note that \[(A \rightarrow B) \rightarrow A \rightarrow B\] and \[(A \rightarrow B) \rightarrow (A \rightarrow B)\] are the same.

The above given \(\lambda\)-term corresponds to the following proof:

- Assume \(A \rightarrow B\).
- Then the conclusion, namely \(A \rightarrow B\) holds.
Curry Howard Isomorphism

That one can write proofs as typed $\lambda$-terms is often referred to as well as the **Curry-Howard Isomorphism**.

Typed $\lambda$-terms are nothing but proofs of the formula given by their type!!
We have seen, that implication is nothing but the function type. Therefore we can represent implication by \( \rightarrow \) in Agda. Elements of formula constructed from \( \rightarrow \) will be proofs that the formula holds.
Example

Take the example of Mary and John as persons and Mary as a student. Assume additionally that if Mary is a student then John is a student as well:

```
postulate Person : Set
postulate john : Person
postulate mary : Person
postulate IsStudent : Person → Set
postulate maryIsStudent : IsStudent mary
postulate implication : IsStudent mary → IsStudent john
```
Then we can prove that John is a student:

Lemma1 : Set
Lemma1 = IsStudent john

proof-lemma1 : Lemma1
proof-lemma1 = implication maryIsStudent
Note that we do not make use of the assumption $x$ in the proof of Lemma 1.

If we added a new person $\text{barbara}$ and tried to prove in the above situation the following wrong Lemma 2:

$$\text{postulate barbara} : \text{Person}$$
$$\text{Lemma2} : \text{Set}$$
$$\text{Lemma2} = \text{IsStudent john} \rightarrow \text{IsStudent barbara}$$

$$\text{proof-lemma2} : \text{Lemma2}$$
$$\text{proof-lemma2} : \{! !\}$$

we will fail.
We can use a $\lambda$-abstraction

\[
\text{proof-lemma2} : \text{Lemma2}
\]

\[
\text{proof-lemma2} = \lambda(x : \text{IsStudent john}) \rightarrow \{! !\}
\]

But there is no way of solving this goal (except by using full recursion, i.e. by calling recursively proof-lemma2, which violates the termination checker.)

See later more on the termination checker.

So we have shown Lemma1, which is true,

and failed to prove Lemma2, which is false.

See maryjohn2.agda
Assume postulates $A : \text{Set}$, $B : \text{Set}$.

We can introduce the formula (or set) expressing $A \rightarrow (A \rightarrow B) \rightarrow B$ as follows:

\[
\begin{align*}
\text{Lemma1} & : \text{Set} \\
\text{Lemma1} & = A \rightarrow (A \rightarrow B) \rightarrow B
\end{align*}
\]

In order to prove Lemma1 we make the following goal:

\[
\begin{align*}
\text{lemma1} & : \text{Lemma1} \\
\text{lemma1} & = \{! \! \}
\end{align*}
\]
Example 2

Lemma1 : Set
Lemma1 = A → (A → B) → B
lemma1 : Lemma1
lemma1 = {! !}

- The type of the goal is $A \rightarrow (A \rightarrow B) \rightarrow B$.
- When the type of goal is an implication, it is usually shown
  - unless one has an assumption which matches the goal directly
    by $\lambda$-abstracting from the premises of the implication.
- Instead of introducing a $\lambda$-abstraction, we apply lemma1 to variables
  $a$ (of type $A$) and $a \rightarrow b$ (of type $A \rightarrow B$).
Example 2

One obtains:

\[ \text{lemma1} : \quad \text{Lemma1} \]
\[ \text{lemma1} \; \text{a a} - b = \{! !\} \]

Lemma1 was \( A \rightarrow (A \rightarrow B) \rightarrow B \),
we have abstracted from \( A \) and \( A \rightarrow B \),
so the type of the goal is the conclusion of the implication, namely \( B \).
Example 2

$$\text{lemma1} : \text{Lemma1}$$

$$= \lambda (a : A) \rightarrow \lambda (a - b : A \rightarrow B) \rightarrow \{! !\}$$

Type of goal is $B$

- At the position of the goal we have context $a : A$ and $a - b : A \rightarrow B$, because we have $\lambda$-abstracted those variables.
  - Can be checked by using goal-menu Context (environment).
- We can take $a - b : A \rightarrow B$ and apply it to $a : A$ in order to obtain $a - b \ a : B$, which solves the goal.
Example 2

- We obtain the following proof:

\[
\text{lemma1} : \quad \text{Lemma1} \\
\text{lemma1}\ a\ a - b = a - b\ a
\]

- This is exactly the same as introducing a \(\lambda\)-term of type \(A \rightarrow (A \rightarrow B) \rightarrow B\).

- See \texttt{exampleProofPropLogic1.agda}
Example 2

Note that in this example

- $a - b$ is an element of the function type $A \rightarrow B$.
- $a$ is an element of $A$
- therefore $a - b$ is an element of $B$,
- therefore the typing is correct.
The type checker in Agda allows recursive definitions. For instance, the following passes the type checker:

\[
\begin{align*}
a & : A \\
\text{rec} & = a
\end{align*}
\]

Necessary, since for instance the definition of \(+\) is necessarily recursive, i.e. will make use of \(+\):

\[
\begin{align*}
\_ + \_ & : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
n + \mathbb{Z} & = n \\
n + S \, m & = S \,(n + m)
\end{align*}
\]
Recursive Definitions and Proofs

- Recursive definitions spoil the principle of propositions as types:

\[
\begin{align*}
a & : A \\
a &= a
\end{align*}
\]

would give a proof of any formula A.

- This does not contradict the constructive meaning of proofs, since the \(a\) above does not carry any constructive information:
  - If we try to evaluate it, we get the infinite reduction sequence

\[
\begin{align*}
a & \rightarrow a \\
a & \rightarrow a \\
a & \rightarrow a \\
\ldots
\end{align*}
\]
Need for Termination Checker

- We have only a constructive proof $p$ of $A$ if $p$ can be reduced to a normal form which is a constructive witness of $A$.
- Therefore we need to restrict Agda to terminating programs.
  - In fact we only need the restriction to terminating proofs.
  - But proofs and programs are so closely tight together that it is difficult to separate them – in Agda we cannot separate termination-checks of programs from termination-checks of proofs.
Agda has a built-in termination checker:
If one loads the buffer, all variables which are defined by a possibly non-terminating recursive equation are marked in red.

The above example becomes:

\[
\begin{align*}
a &: A \\
\; a &= \; a
\end{align*}
\]
Since this colour coding is easily overlooked, it is recommended to run at the end of a session from a shell the command `agda` applied to each Agda file created.

- This will list all problems
  - errors,
  - problems due to failure of the termination checker,
  - still open goals.
- If there are any remaining problems, solve them, and then recheck the file again, until everything is correct.
The termination checker has limitations:

- **If the termination check succeeds**, all programs checked will terminate.
  - Therefore all proofs will be actual proofs of the corresponding propositions.

- **If the termination check fails**, it might still be the case that all programs terminate.
  (One cannot write a universal termination checker, since the Turing halting problem is undecidable).
  - So the proofs might be proofs, or might not be proofs.
Examples

- \( a : A \)
  \[ a = a \]
  will not pass the termination checker.

- \( f : A \rightarrow A \)
  \[ f \, a = a \]
  will pass the termination checker.

- \( \text{lemma} \)
  \[ (A \rightarrow B) \rightarrow A \rightarrow B \]
  \[ \text{lemma} \, a - b \, a = \text{lemma} \, a - b \, a \]
  will not pass the termination checker.
Examples

- **lemma**: 
  \[(A \rightarrow B) \rightarrow A \rightarrow B\]

  lemma \(a - b a = a - b a\)

  passes the termination checker.
In general, the termination checker will check whether there is any definition of a constant or a local variable, which depends on itself.

When later dealing with natural numbers and algebraic types, we will see that some circularities can be acceptable and are accepted by the termination checker.

But until then in general the rule is that recursive definitions, in which the definition of a constant refers directly or indirectly to itself, are not allowed.
The \( \eta \)-Rule

- If we have a function \( f : \sigma \rightarrow \tau \), then this function applied to \( a : \sigma \) gives result \( f \ a \).
- If we apply \( \lambda x^\sigma. f \ x : \sigma \rightarrow \tau \) to \( a : \sigma \), we get the same result \( f \ a \).
- Therefore \( f \) is as a function the same as \( \lambda x. f \ x \) (where \( x \) is fresh).
- However, if for instance \( f \) is a variable, we don't have \( f \equiv_\beta \lambda x. f \ x \).
The $\eta$-Rule

- Especially, when working later in dependent type theory we want to identify as many terms as possible, which are equal. This will make it easier to prove certain goals.

- $\eta$-expansion expresses that subterms $t : \sigma \rightarrow \tau$ can be $\eta$-expanded to $\lambda x.t \ x$ (where $x$ does not occur free in $t$).

- Then any $f : \sigma \rightarrow \tau$ is always equal to $\lambda x.f \ x$ w.r.t. $\beta, \eta$-reduction (where $x$ is fresh).

- One needs to restrict $\eta$-expansion slightly in order to obtain a normalising reduction system.
  - Details can be found on the next few slides, but won’t be treated in the lecture.
  - We jump directly to the $\eta$-rule in Agda.
However, we need to impose some restrictions, in order to avoid circularities (i.e. that a term reduces to itself) which destroy normalisation:

- If $t$ is of the form $\lambda y.s$ and if we then allowed to expand $t$, we would obtain the following circularly:

$$t \rightarrow \lambda x.t\ x \equiv \lambda x.(\lambda y.s)\ x \rightarrow_{\beta} \lambda x.s[y := x] \equiv t,$$

- If $t$ is applied to some other term, e.g. $t$ occurs as $t\ r$, and if we allowed to expand $t$ we would get the following circularity:

$$t\ r \rightarrow (\lambda x.t\ x)\ r \rightarrow_{\beta} t\ r$$

- All other terms can be expanded without obtaining a new redex.
\( \eta \)-Expansion

- **\( \eta \)-expansion** (or **\( \eta \)-rule**) is the rule which expands one subterm of a \( \lambda \)-term
  - of the form \( r : \sigma \rightarrow \tau \)
  - s.t. \( r \) is not of the form \( \lambda u^\sigma . t \)
  - and such that \( r \) is not applied to some other term to \( \lambda x^\sigma . r \ x \), where \( x \) does not occur free in \( r \).
- We write
  - \( r \overset{\eta}{\rightarrow} s \) for \( s \) is obtained from \( r \) by the \( \eta \)-rule,
  - \( r \overset{\beta, \eta}{\rightarrow} s \) for \( s \) is obtained from \( r \) by using \( \beta \)-reduction or \( \eta \)-expansion.
- Notions like \( \overset{*, \eta}{\rightarrow}, \overset{=, \eta}{\rightarrow}, \overset{\beta, \eta, \beta, \eta}{\rightarrow}, \beta, \eta \)-normal form, etc. are to be understood correspondingly.
Example

Assume $f : o^3$. Then

$$r := (\lambda f^o^3.\lambda x^o^2.f\ x)\ f$$
$$\rightarrow_\beta \lambda x^o^2.f\ x$$
$$\rightarrow_\eta \lambda x^o^2.\lambda y^o.f\ x\ y$$

(by $\eta$-expanding $f\ x : o^2$ to $\lambda y^o.f\ x\ y$)

$$\rightarrow_\eta \lambda x^o^2.\lambda y^o.f\ (\lambda z^o.x\ z)\ y =: s$$

(by $\eta$-expanding $x : o^2$ to $\lambda z^o.x\ z$)

Note that in the last step, $x$ was not in an applied position, since $f\ x\ y$ stands for $(f\ x)\ y$. 


Example

\[ r := \lambda f^{o3} . \lambda x^{o2} . f \ x \) \ f \xrightarrow{\beta} \lambda x^{o2} . f \ x \xrightarrow{\eta}^* \lambda x^{o2} . \lambda y^{o} . f \ (\lambda z^{o} . x \ z) \ \ y =: s \]

- There are no more \( \eta \)-expansions or \( \beta \)-reductions possible in \( s \):
  - The terms \( f \) and \( x \) occur in a position where they are applied to another term, so they are not supposed to be \( \eta \)-expanded.
  - \( z \) and \( y \) are of ground type and therefore not to be \( \eta \)-expanded.
Example

\[ r := \lambda f^o_3.\lambda x^o_2.f \ x \ \rightarrow^\beta \ \lambda x^o_2.f \ x \]
\[ \quad \rightarrow^* \ \lambda x^o_2.\lambda y^o.f \ (\lambda z^o.x \ z) \ y =: s \]

- Because \( s \) cannot be expanded any further, it is the \( \beta, \eta \)-normal form of \( r \).
- Since \( f \rightarrow^\eta \lambda x^o_2.f \ x \), the term \( s \) is as well the \( \beta, \eta \)-normal form of \( f : o3 \).
Example 2

If we replace in the above example $o$ by $o2$ (and therefore $o2$ by $o3$ and $o3$ by $o4$) we obtain

$$(\lambda f^o4.\lambda x^o3.f\ x)\ f$$

$$\longrightarrow\beta\lambda x^o3.f\ x$$

$$\longrightarrow\eta\lambda x^o3.\lambda y^o2.f\ x\ y$$

$$\longrightarrow\eta\lambda x^o3.\lambda y^o2.\lambda z^o.f\ x\ y\ z$$

$$\longrightarrow\eta\lambda x^o3.\lambda y^o2.\lambda z^o.f\ (\lambda u^o2.x\ u)\ y\ z$$

$$\longrightarrow\eta\lambda x^o3.\lambda y^o2.\lambda z^o.f\ (\lambda u^o2.\lambda v^o.x\ u\ v)\ y\ z$$

$$\longrightarrow\eta\lambda x^o3.\lambda y^o2.\lambda z^o.f\ (\lambda u^o2.\lambda v^o.x\ (\lambda w^o.u\ w)\ v)\ y\ z$$

$$\longrightarrow\eta\lambda x^o3.\lambda y^o2.\lambda z^o.f\ (\lambda u^o2.\lambda v^o.x\ (\lambda w^o.u\ w)\ v)\ (\lambda u^o.y\ u)\ z$$

which is as well the $\beta, \eta$-normal form of $f : o4$. 
Intuitive Application of $\eta$-Expansion

Intuitively, $\eta$-expansion for terms in $\beta$-normal form is obtained as follows:

- Consider subterms

\[ r := t_1 \ t_2 \ \cdots \ t_n \]

of the term to be $\eta$-expanded which are longest, i.e. they don’t occur as

\[ t_1 \ t_2 \ \cdots \ t_n \ t_{n+1} \]

for some $t_{n+1}$.

- If $r : \alpha \rightarrow \beta$ it is an $\eta$-redex.
- Otherwise $r$ is of ground type and not an $\eta$-redex.
Intuitive Application of $\eta$-Expansion

- If

\[ r := t_1 \ t_2 \ \cdots \ \ t_n \]

is an $\eta$-redex, expand it to

\[ \lambda x^\alpha \cdot t_1 \ t_2 \ \cdots \ t_n \times . \]

- Continue until there are no $\eta$-redexes left.
Theorem

- The typed λ-calculus with β-reduction and η-expansion is confluent and strongly normalising.
With the \( \eta \)-rule, we obtain that if \( r : \sigma \rightarrow \tau \), then \( r =_{\beta, \eta} \lambda x^\sigma.r \ x \).

If \( r : \sigma \rightarrow \tau \) is of the form \( \lambda u^\sigma.t \) then we have \( r =_{\beta} \lambda x^\sigma.r \ x \):

\[
\begin{align*}
\lambda x^\sigma.r \ x & \equiv \lambda x^\sigma.(\lambda u^\sigma.t) \ x \\
\longrightarrow_{\beta} & \lambda x^\sigma.t[u := x] \\
=_{\alpha} & \lambda u^\sigma.t \\
\equiv & r
\end{align*}
\]

Otherwise \( r \longrightarrow_{\eta} \lambda x^\sigma.r \ x \).

Therefore one can say the \( \eta \) rule expresses: every element of a function type is of the form \( \lambda x.\text{something} \).
η-Reduction

- In the literature one often uses instead of η-expansion \textit{η-reduction}, which allows to reduce $\lambda x. r \ x$ to $r$, if $x$ doesn’t occur free in $r$.
- The computation of η-reduction is more difficult than η-expansion, since one has to check, whether $x$ doesn’t occur free in $r$. Therefore in the context of interactive theorem proving, we prefer η-expansion.
In Agda syntax, the $\eta$-rule states that if

\[ f : A \to B \]

then

\[ f = \lambda(x : A) \to f \, x \, . \]

The $\eta$-rule is implemented in Agda2.

We will in this lecture omit the remaining parts of this section.
Remark on Weakening

- If we have derived \( t : \sigma \) under some context, then the same holds for any other context, which expands the original one.

- Formally, this means: Assume
  \[
  \Gamma, \Delta \Rightarrow t : \sigma .
  \]
  Then we have as well
  \[
  \Gamma, x : \tau, \Delta \Rightarrow t : \sigma ,
  \]
  provided \( \Gamma, x : \tau, \Delta \) is a context (i.e. provided \( x \) does not occur in \( \Gamma, \Delta \)).

- The process of extending the context is called \textit{weakening}.  

Weakening in Logic

- Weakening occurs in many logic calculi as well.
- It occurs in natural language reasoning as well:
  - For instance from “I am living an Swansea” and “In Swansea the sun is shining” follows “Where I am living, the sun is shining”.
  - However, we can derive the above as well from the additional (unused) assumption “Assuming that I am a lecturer”.
  - So we have as well “Under the assumption that I am a lecturer, where I am living the sun is shining”, which is a weaker statement.
Proof of the Remark

- Assume a derivation of $\Gamma, \Delta \Rightarrow t : \sigma$.
- Insert at all corresponding positions in the contexts in the derivation $x : \tau$.
  - One needs to rename variables, in order to avoid conflicts with $x$.
- The result is a derivation of $\Gamma, x : \tau, \Delta \Rightarrow t : \sigma$. 
Example (Weakening)

- From the derivation

\[
\begin{align*}
y : o, x : o &\Rightarrow x : o \\
y : o &\Rightarrow \lambda x^o.x : o2 \\
y : o &\Rightarrow (\lambda x^o.x) y : o
\end{align*}
\]

we obtain a derivation of

\[
y : o, x : o \Rightarrow (\lambda x^o.x) y : o
\]

by inserting in each context in the derivation, after \( y : o \) the context \( x : o \).
Example (Weakening)

\[
\frac{\text{Abs}}{y : o, x : o \Rightarrow \lambda x^o.x : o2} \quad \text{(Abs)}
\]

\[
\frac{\text{Ap}}{y : o \Rightarrow \lambda x^o.x \Rightarrow y : o} \quad \text{(Ap)}
\]

We obtain the following derivation of \( y : o, x : o \Rightarrow (\lambda x^o.x) \ y : o \)

\[
\frac{\text{Abs}}{y : o, x : o, x : o \Rightarrow x : o} \quad \text{(Abs)}
\]

\[
\frac{\text{Ap}}{y : o, x : o \Rightarrow \lambda x^o.x : o2} \quad \text{(Ap)}
\]

\[
\frac{\text{Ap}}{y : o, x : o \Rightarrow (\lambda x^o.x) \ y : o} \quad \text{(Ap)}
\]
Because of the possibility of weakening, we will usually omit unused parts of contexts.

So a derivation of $x : o2, y : o \Rightarrow x (x y) : o$, which in full reads as follows

$$
\text{Ap}
\begin{align*}
&\quad \text{Ap} \\
&\quad \text{Ap}
\end{align*}
$$

will usually be presented as follows:

$$
\text{Ap}
\begin{align*}
&\quad \text{Ap} \\
&\quad \text{Ap}
\end{align*}
$$
We introduced the typed $\lambda$-calculus, in order to avoid non-normalising terms, as they occur in the untyped $\lambda$-calculus.

The non-normalising terms we introduced used some form of self application.

For instance we introduced

- $\omega := \lambda x.x \, x$, (where $x$ was applied to itself)
- $\Omega := \omega \, \omega$

and had

- $\Omega \rightarrow_\beta \Omega$.

In the following, we will investigate, how self-application is avoided in the typed $\lambda$-calculus.
In the simply typed $\lambda$-calculus we cannot assign a type to $\lambda x.x \ x$, i.e. there are no types $\sigma, \tau$ s.t. $\lambda x^\sigma. x \ x : \tau$.

- Assume we could derive this.
  The only way to derive $\lambda x^\sigma. x \ x : \tau$ is by the rule of $\lambda$-abstraction.
- Then $\tau$ must be equal to $\sigma \rightarrow \tau_1$ for some $\tau_1$, and the derivation reads

$$
\frac{x : \sigma \Rightarrow x \ x : \tau_1}{\lambda x^{\sigma}. x \ x : \sigma \rightarrow \tau_1} \quad (\text{Abs})
$$
Self-Application

\[
\frac{x : \sigma \Rightarrow x \ x : \tau_1}{\lambda x^\sigma. x \ x : \sigma \rightarrow \tau_1} \quad \text{(Abs)}
\]

- \(x : \sigma \Rightarrow x \ x : \tau\) must have been derived by the rule of application, so the derivation must look like this:

\[
\frac{x : \sigma \Rightarrow x : \tau_2 \rightarrow \tau_1}{x : \sigma \Rightarrow x \ x : \tau_1} \quad \frac{x : \sigma \Rightarrow x : \tau_2}{\lambda x^\sigma. x \ x : \sigma \rightarrow \tau_1} \quad \text{(Ap)}
\]

\[
\frac{x : \sigma \Rightarrow x : \tau_2 \rightarrow \tau_1}{x : \sigma \Rightarrow x \ x : \tau_1} \quad \frac{x : \sigma \Rightarrow x : \tau_2}{\lambda x^\sigma. x \ x : \sigma \rightarrow \tau_1} \quad \text{(Abs)}
\]
Self-Application

\[
\frac{x : \sigma \Rightarrow x : \tau_2 \rightarrow \tau_1 \quad x : \sigma \Rightarrow x : \tau_2}{x : \sigma \Rightarrow x \cdot x : \tau_1} \quad \text{(Ap)}
\]

\[
\frac{x : \sigma \Rightarrow x : \tau_2 \rightarrow \tau_1}{\lambda x^\sigma . x \cdot x : \sigma \rightarrow \tau_1} \quad \text{(Abs)}
\]

- The only way to derive \(x : \sigma \Rightarrow x : \tau_2 \rightarrow \tau_1\) and \(x : \sigma \Rightarrow x : \tau_2\) is by using the assumption rule.
- In order for \(x : \sigma \Rightarrow x : \tau_2 \rightarrow \tau_1\) to be derivable by the assumption rule, we need \(\sigma = \tau_2 \rightarrow \tau_1\).
- Similarly, in order to derive \(x : \sigma \Rightarrow x : \tau_2\), we need \(\tau_2 = \sigma\).
- So we have \(\tau_2 \rightarrow \tau_1 = \sigma = \tau_2\).
- But \(\tau_2 = \tau_2 \rightarrow \tau_1\) cannot be fulfilled, since \(\tau_2 \rightarrow \tau_1\) is longer than \(\tau_2\).
- So we cannot find types \(\sigma, \tau\) s.t. \(\lambda x^\sigma . x \cdot x : \tau\).