3. The λ-Calculus and Implication

(a) The untyped λ-calculus.

(b) The typed λ-calculus.

(c) The λ-Calculus in Agda.

(d) Logic with Implication

(e) Implicational Logic in Agda.

(f) More on the typed λ-calculus.

(a) The Untyped λ-Calculus

- Basic idea of the λ-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

- λ 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 φ.
  - 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 Γ, Delta for Δ.

The Greek Alphabet

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

Example for Use of λ

- Can be done by multiplying each student number by 10.
- Let $f : \mathbb{N} \to \mathbb{N}$, $f(x) := x * 10$.
- 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

$$\begin{aligned}
&\mathrm{map}(f, [210345, 345698, 296458]) \\
&\quad = [2103450, 3456980, 2964580] \quad .
\end{aligned}$$

Introduction to λ-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. $f(x) = x * 10$".
- Notation is $\lambda x.x * 10$.
- So we have

$$\begin{aligned} & \mathrm{map}(\lambda x.x * 10, [210345, 345698, 296458]) \\ = & [2103450, 3456980, 2964580] \quad . \end{aligned}$$

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

Notation

- 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 $(\cdots((r_0\ r_1)\ r_2)\ \cdots r_n)$.

Abbreviations

- We write $\lambda x, y. \cdots$ for $\lambda x.\lambda y. \cdots$.
- Similarly for $\lambda x, y, z.$ etc.
- 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 $\mathrm{S}\ x + \mathrm{S}\ y$ stands for $(\mathrm{S}\ x) + (\mathrm{S}\ y)$.
- $*$ binds more than $+$.
  Therefore $x + y * z$ stands for $x + (y * z)$,
  and $\mathrm{S}\ x + \mathrm{S}\ y * z$ stands for $(\mathrm{S}\ x) + ((\mathrm{S}\ y) * z)$.
- In Agda we can achieve this by using the code

$$\mathrm{infixl}\ 60\ \_+\_$$
$$\mathrm{infixl}\ 80\ \_*\_$$

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 λx
  - is "$x + 5$",
  - not "$x + 5, 3)$", which doesn't make sense.
- In $(\lambda x.x + 5)\ 3$, the scope of λx
  - is $x + 5$
  - not $x + 5)\ 3$, which doesn't make sense.

λ without a Dot

- Sometimes, $\lambda x\ t$ (without a dot) is used, if one wants to have the scope of λx **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.

λ-Terms

- Now we can define the terms of the untyped λ-calculus as follows:
- λ **terms** are:
  - Variables x,
  - If r and s are λ-terms, so is $(r\ s)$.
  - If x is a variable and r is a λ-term, so is $(\lambda x.r)$.
- As usual brackets can be omitted, using
  - the above mentioned conventions about the scope of λx,
  - and that $r\ s\ t$ is read as $(r\ s)\ t$.

λ-Terms

- Examples:
  - $\lambda x.x$,
  - $\lambda x.(\lambda y.y)\ x$,
  - $\lambda x.x\ x$,
  - $(\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)$,
  - $\lambda f.\lambda x.f\ (f\ x)$.

λ-Terms

- One might need additional constants to the language, then we have additionally:
  - Any constant is a λ-term.
- For instance,
  - if c is a constant, then $\lambda x.c$, $(\lambda x.x)\ c$ are λ-terms;
  - if $(+)$ is a constant, then $\lambda x.(+)\ x\ x$ is a a λ-term.
- For standard operators like $+$, $*$, one has
  - constants $(+)$, $(*)$,
  - infix operations $+$, $*$,
  - and writes in infix notation
    - $x + y$ instead of $(+)\ x\ y$,
    - $x * y$ instead of $(*)\ x\ y$,
    - etc.

Bound and Free Variables

- There are bound and of free variables in λ-terms:
  - **Bound variables** are variables x, which occur in the scope of a λ-abstraction "$\lambda x.$".
  - **Free variables** are the other variables.
  - **Example:** In $\lambda x.x + y$,
    - x is bound (since in the scope of λx),
    - y is free (since it is not in the scope of λy).

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 λy),
  - x is free.

Bound and Free Variables

- 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.

α-Conversion

- We identify λ-terms, which only differ in the choice of the bound variables (variables abstracted by λ):
  - 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 **α-equality**, and the step from one term to another α-equal term is called **α-conversion**.
- So $\lambda x.\lambda y.y$ and $\lambda y.\lambda x.x$ are **α-equal**, written as $\lambda x.\lambda y.y =_\alpha \lambda y.\lambda x.x$.

α-Conversion

- Note that $\lambda \mathbf{x}.\lambda x.x =_\alpha \lambda y.\lambda x.x$.
  - The x refers to the second lambda abstraction λx, not the first one ($\lambda \mathbf{x}.$).
  - Therefore, when changing the variable of the first λ-abstraction, x remains unchanged.

Evaluation of λ-Terms

- How do we evaluate $(\lambda x.x * 10)\ 5$?
  - We first replace in $x * 10$, the variable x by 5.
  - We obtain $5 * 10$.
  - Then we reduce this further, using other reduction rules (not introduced yet).
    Using suitable rules, we would reduce $5 * 10$ to 50.
  - In this Subsection we will look only at the pure λ-calculus without any additional reduction rules.
    There $(\lambda x.x * 10)\ 5$ reduces to $5 * 10$, 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 $5 + 10$.
  - $(\lambda x.x)\ \text{"Student"}$ 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 $\lambda y.y$.

Substitution

- The last example shows that substitution by λ-terms can become more complicated, and we therefore instudy it in the following more carefully.
- If t and s are λ-terms, $t[x := s]$ denotes the result of substituting in t the variable x by s, e.g.
  - $(x + 10)[x := 5] \equiv 5 + 10$,
  - $x[x := \text{"Student"}] \equiv \text{"Student"}$,
  - $x[x := \lambda y.y] \equiv \lambda y.y$.

Substitution and Parentheses

- Subsitution 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.

Substitution and Parentheses

- E.g.

$$(x * x)[x := 2 + 2] = (2 + 2) * (2 + 2) \quad .$$

- 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

$$2 + 2 * 2 + 2$$

which is different from

$$(2 + 2) * (2 + 2) \quad .$$

Substitution and Bound Variables

- If we carry out a substitution in a λ-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.

Substitution and Bound Variables

- More examples:
  - $(\lambda x.x)[x := \text{"Student"}] \equiv \lambda x.x$.
    - The x in $\lambda x.x$ is bound by λx, so no substitution is carried out.
  - $((\lambda x.{\color{red}x})\ {\color{blue}x})[x := \text{"Student"}] \equiv (\lambda x.x)\ \text{"Student"}$.
    - The first ${\color{red}x}$ is bound, so no substitution is carried out.
    - The second ${\color{blue}x}$ is free, so substitution is carried out.
  - $(\lambda y.x + y)[x := 3] \equiv \lambda y.3 + y$.
    - x in $\lambda y.x + y$ is free, so it will be substituted by 3 in the above example.

Substitution and α-Conversion

- When substituting in λ-terms, we sometimes have to carry out an α-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 α-Conversion

- The **correct way** is as follows:
  - First we α-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:

$$(\lambda y.y + x)[x := y] =_\alpha (\lambda z.z + x)[x := y] \equiv \lambda z.z + y \quad .$$

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

$$(\lambda y.y+x)[x := y+y] =_\alpha (\lambda z.z+x)[x := y+y] \equiv \lambda z.z+(y+y)$$

Substitution and α-Conversion

- In general, the substitution $t[x := s]$ is carried out as follows:
  - α-convert t s.t.
    - if x occurs in t free and is in the scope of some λu,
    - then u doesn't occur free in s.
    - In other words, α-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: α-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 λy.
- $(\lambda x.(\lambda y.y)\ y)[y := x] =_\alpha (\lambda u.(\lambda y.y)\ y)[y := x] \equiv \lambda u.(\lambda y.y)\ x$.

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 λ-binding λx.
- $(\lambda x.z)[z := (\lambda x.x)\ x] =_\alpha (\lambda u.z)[z := (\lambda x.x)\ x] \equiv \lambda u.((\lambda x.x)\ x)$.
  Now x occurs free in $(\lambda x.x)\ x$ (the second occurrence is free), so we need to α-convert it.

Substitution and α-Conversion

- If you have problems understanding this, you can proceed as follows, and are on the safe side:
  - α-convert t so that all bound variable in t are different from all free variables in s.
  - Then carry out the substitution.
- An unnecessary α-conversion doesn't hurt.

$s[x]$, $s[t]$

- 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 $s[x] \equiv a\ x$ 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.

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

$s[x]$, $s[t]$

- Examples:
  - If $s[x] \equiv x$ then $s[t] \equiv t$.
  - If $s[x] \equiv a\ x$, then $s[t] \equiv a\ t$.
  - 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 α-conversion, before we can carry out the subsitutiton.
- 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).

β-Redexes

- The notion of β-reduction is one step in the sense of evaluation of a λ-term to another term.
- We first introduce the notion of a β-redex of a term t:
- A subterm $(\lambda x.r)\ s$ of a λ-term t is called a **β-redex** of t.
- **Examples:**
  - $(\lambda x.x)\ y\ z$ has β-redex $(\lambda x.x)\ y$.
    - Note that the bracketing is $((\lambda x.x)\ y)\ z$, **not** $(\lambda x.x)\ (y\ z)$.
  - A redex can be the term itself: $(\lambda x.x)\ y$ has β-redex $(\lambda x.x)\ y$.

β-Redexes

- A λ-term might have several β-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 $(\lambda y.y)\ z$.

β-Reduct

- A β-redex $(\lambda x.s)\ t$ can be reduced to $s[x := t]$.
  - $s[x := t]$ is called the **β-reduct** of $(\lambda x.s)\ t$.
  - The β-reduct of $(\lambda x.x + 10)\ 5$ is $5 + 10$,
  - The β-reduct of $(\lambda x.x)\ \text{"Student"}$ is "Student".
  - The β-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 $s[t]$."

β-Reduction

- $r \longrightarrow_\beta r'$, "$r$ β**-reduces to** r', or shorter $r \longrightarrow r'$, if r' is obtained from r by replacing one β-redex by its β-reduct.
- **Examples:**
  - $((\lambda \mathbf{x}.\mathbf{x} + \mathbf{5})\ \mathbf{3}) + 7 \longrightarrow (\mathbf{3} + \mathbf{5}) + 7$, since

$$(\lambda x.x + 5)\ 3 \longrightarrow 3 + 5\quad .$$

  - Assume we add a pairing operation $\langle s, t\rangle$ for the pair s, t (will be introduced later), then

$$\langle(\lambda \mathbf{x}.\mathbf{x} + \mathbf{5})\ \mathbf{3}, 7\rangle \longrightarrow \langle \mathbf{3} + \mathbf{5}, 7\rangle\quad ,$$

Examples

- We can apply β-reduction under a λ term as well:

$$\lambda x.((\lambda \mathbf{y}.\mathbf{y} + \mathbf{5})\ \mathbf{3}) \longrightarrow \lambda x.\mathbf{3} + \mathbf{5}\ \ .$$

- **Multiple redexes**:
  Because a λ-term might have several redexes, it might have two different reductions:
  - For instance
    - $(\lambda x.x\ x)\ ((\lambda y.y)\ z) \longrightarrow ((\lambda y.y)\ z)\ ((\lambda y.y)\ z)$
    - $(\lambda x.x\ x)\ ((\lambda y.y)\ z) \longrightarrow (\lambda x.x\ x)\ z$.

Examples of β-Reduction

$$(\lambda x.\lambda y.x)\ y \longrightarrow (\lambda y.x)[x := y] =_\alpha (\lambda u.x)[x := y] \equiv \lambda u.y$$

$$\begin{aligned}(\lambda z.\lambda x.\lambda y.z)\ x \longrightarrow (\lambda x.\lambda y.z)[z := x] &=_\alpha (\lambda u.\lambda y.z)[z := x] \\ &\equiv \lambda u.\lambda y.x\end{aligned}$$

$$(\lambda z.\lambda x.(\lambda y.y)\ z)\ y \longrightarrow (\lambda x.(\lambda y.y)\ z)[z := y] \equiv \lambda x.(\lambda y.y)\ y$$

$$\lambda x.(\lambda y.y)\ y \longrightarrow \lambda x.y$$

Example (Longer Reduction)

- In the steps marked $\equiv$ on the next slide, essentially the colouring is changed to mark the next β-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 β-reductions as well.
- Remember as well that

$$\lambda x, y.t$$

abbreviates

$$\lambda x.\lambda y.t$$

Example (Longer Reduction)

$$(\lambda x, y.x\ (x\ y))\ (\lambda u, v.u\ (u\ v))$$

$$\equiv (\lambda \mathbf{x}.\lambda y.\mathbf{x}\ (\mathbf{x}\ y))\ (\lambda u, v.u\ (u\ v))$$

$$\longrightarrow \lambda y.(\lambda u, v.u\ (u\ v))\ ((\lambda u, v.u\ (u\ v))\ y)$$

$$\equiv \lambda y.(\lambda u, v.u\ (u\ v))\ ((\lambda \mathbf{u}.\lambda v.\mathbf{u}\ (\mathbf{u}\ v))\ y)$$

$$\longrightarrow \lambda y.(\lambda u, v.u\ (u\ v))\ (\lambda v.y\ (y\ v))$$

$$\equiv \lambda y.(\lambda \mathbf{u}.\lambda v.\mathbf{u}\ (\mathbf{u}\ v))\ (\lambda v.y\ (y\ v))$$

$$\longrightarrow \lambda y.\lambda v.(\lambda v.y\ (y\ v))\ ((\lambda v.y\ (y\ v))\ v)$$

$$\equiv \lambda y.\lambda v.(\lambda v.y\ (y\ v))\ ((\lambda \mathbf{v}.y\ (y\ \mathbf{v}))\ v)$$

$$\longrightarrow \lambda y.\lambda v.(\lambda v.y\ (y\ v))\ (y\ (y\ v))$$

$$\equiv \lambda y.\lambda v.(\lambda \mathbf{v}.y\ (y\ \mathbf{v}))\ (y\ (y\ v))$$

$$\longrightarrow \lambda y.\lambda v.y\ (y\ (y\ (y\ v)))$$

$$\equiv \lambda y, v.y\ (y\ (y\ (y\ v)))$$

Examples of Non-Termination

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

$$\Omega \equiv \omega\ \omega \equiv (\lambda x.x\ x)\ \omega \longrightarrow \omega\ \omega \equiv \Omega\ \ .$$

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

$$\begin{aligned}
&\widetilde{\Omega}\ \widetilde{\Omega} \equiv (\lambda x.x\ (x\ x))\ \widetilde{\Omega} \\
&\longrightarrow \widetilde{\Omega}\ (\widetilde{\Omega}\ \widetilde{\Omega}) \\
&\longrightarrow \widetilde{\Omega}\ (\widetilde{\Omega}\ (\widetilde{\Omega}\ \widetilde{\Omega})) \\
&\longrightarrow \cdots
\end{aligned}$$

Remark on Previous Slide

- Note that in the λ-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.**

λ-Calc. as a Red. Sys

- By the **untyped λ-calculus** (short **λ-calculus**) we mean now
  - the set of λ-terms, T where α-equivalent λ-terms are identified,
  - together with β-reduction $\longrightarrow_\beta$.
- Therefore the λ-calculus forms a reduction system $(T, \longrightarrow_\beta)$.
- One might have the λ-calculus with additional constants.
  - Without additional constants, the (untyped) λ-calculus is called the **pure (untyped) λ-calculus**.

$\longrightarrow_\beta^*$ and $=_\beta$

- For reduction systems we introduced notations $\longrightarrow^*$, $a \longleftrightarrow^* b$.
- These notions can be used for the λ-calculus as well.
- We define $r =_\beta s$ ("r **and** s **are** β**-equivalent**") iff

  $r \longleftrightarrow_\beta^* s$.
- Since we identified α-equivalent λ-terms, there can be arbitrary many α-conversions in a chain for showing that $r =_\beta s$.
- Therefore we have $r =_\beta r'$ iff there exists a sequence $s_0, \ldots, s_n, t'_0, \ldots, t'_n$ ($n = 0$ is possible) s.t.

$$r \equiv s_0 =_\alpha t_0 \longleftrightarrow_\beta s_1 =_\alpha t_1 \longleftrightarrow_\beta s_2 =_\alpha t_2 \longleftrightarrow_\beta \cdots \longleftrightarrow_\beta s_n =_\alpha t_n \equiv r' \ .$$

Confluence of the λ-Calculus

- **Fact:** The λ-calculus is confluent (if we identify α-equivalent terms).
- Therefore two λ terms r and s are β-equivalent, iff there exits a term t s.t. $r \longrightarrow_{\beta}^{*} t$ and $s \longrightarrow_{\beta}^{*} t$.
- **Example:** $((\lambda y.y)\ z)\ ((\lambda y.y)\ z)$ and $(\lambda x.x\ x)\ z$ are β-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.

β-equality

- Note that this doesn't give yet an easy way of determining whether $r =_\beta s$ holds:
  - One needs to find a t s.t. $s \longrightarrow^* t$ and $r \longrightarrow^* t$.
  - But simply reducing r might never terminate.
- Example:
  - $(\lambda x.y)\ \Omega$ reduces in one step to y.
  - So $(\lambda x.y)\ \Omega =_\beta y$.
  - However, by reducing Ω we obtain Ω, therefore $(\lambda x.y)\ \Omega \longrightarrow (\lambda x.y)\ \Omega$.
  - So if we keep on following the second reduction, we will never find that this term is β-equivalent to y.

Need for Typed λ-Calculus

- Therefore we introduce the typed λ-calculus, which is strongly normalising, and in which therefore equality of λ-terms can be decided by determining α-equality of normal forms.

(b) The Typed λ-Calculus

- Problem of the untyped λ-calculus:
  - Non-Termination, therefore $=_\beta$ difficult to check.
    - In fact $=_\beta$ 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 λ-Calculus

- $\lambda x.x + 5$ is a function,
  - taking an $x : \mathrm{Int}$,
  - and returning $x + 5 : \mathrm{Int}$.
- Therefore, we say that $(\lambda x.x + 5) : \mathrm{Int} \to \mathrm{Int}$.
  - In words, “$\lambda x.x + 5$ is of type Int arrow Int”.
- In order to clarify the type of x, we write instead of $\lambda x.x + 5$

$$\lambda x^{\mathrm{Int}}.x + 5 \quad .$$

or

$$\lambda(x : \mathrm{Int}).x + 5 \quad .$$

Basics of the Typed λ-Calculus

- $\lambda x^{\mathrm{Int}}.x + 5$ is
  - only applicable to some $s : \mathrm{Int}$,
  - therefore not applicable to elements of other types, e.g. to “Student” ($: \mathrm{String}$).
- So
  - $(\lambda x^{\mathrm{Int}}.x + 5)\ 3$ is allowed,
  - $(\lambda x^{\mathrm{Int}}.x + 5)$ “Student” is **not** allowed.

Simple Types

- The **simple types** used in the simply typed λ-calculus are defined inductively as follows:
  - The **ground type** o is a type.
  - If σ, τ are types, so is $(\sigma \to \tau)$.
- "Inductively" means that the set of simple types is the least set containing the ground type, and which closed under $\to$.
- One sometimes modifies the set of ground types, especially when adding constants to the λ-terms.
  - E.g. when using arithmetic expressions, one can say for instance that the ground types are Int and Float.
  - Then we talk about the **simple types based on ground types Int and Float**.

Simple Types

- Usually we denote types by Greek letters,
  - e.g. α (“alpha”), β (“beta”), γ (“gamma”), σ (“sigma”), τ (“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,
  - $\mathrm{o} \to \mathrm{o}$,
  - $(\mathrm{o} \to \mathrm{o}) \to \mathrm{o}$,
  - $((\mathrm{o} \to \mathrm{o}) \to \mathrm{o} \to \mathrm{o}) \to (\mathrm{o} \to \mathrm{o}) \to \mathrm{o} \to \mathrm{o}$,
    - which stands for $(((\mathrm{o} \to \mathrm{o}) \to (\mathrm{o} \to \mathrm{o})) \to ((\mathrm{o} \to \mathrm{o}) \to (\mathrm{o} \to \mathrm{o})))$.

Abbreviation

- 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.
  - $\mathrm{o2} := \mathrm{o} \to \mathrm{o}$,
  - $\mathrm{o3} := \mathrm{o2} \to \mathrm{o2}$,
  - etc.
- So
  - an element of type $\mathrm{o2}$ can be applied to an element of type o and one obtains an element of type o.
  - an element of type $\mathrm{o3}$ can be applied to an element of type $\mathrm{o2}$ and one obtains an element of type $\mathrm{o2}$.
  - etc.

Contexts

- To determine the type of a term makes only sense, if we know the types of its variables.
  - For instance, in case of the λ-term $x\ y$, we could have
    - $x : \mathrm{o2}$, $y : \mathrm{o}$ and therefore $x\ y : \mathrm{o}$,
    - or $x : \mathrm{o3}$, $y : \mathrm{o2}$, and therefore $x\ y : \mathrm{o2}$.
  - Therefore we will give a type to λ terms in a context, which determines the types of the variables.

Contexts

- A **context** is an expression of the form $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ where
  - x_i are variables,
  - σ_i are simple types,
    (when considering other type theories, σ_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 overriden by "$x : \rho$", so the assumption in this context is $x : \rho$.

Contexts

- **Examples**
  - $x : \mathrm{o}, y : \mathrm{o2}$ is a context.
  - $x : \mathrm{o2}, x : \mathrm{o}$ is a context in which we assume $x : \mathrm{o}$.
- Note that contexts are **lists** of elements of the form $x : \sigma$, so the order matters.
  - In case of the simply typed λ-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.

Contexts

- In the following, the capital Greek letters Γ ("Gamma"), Δ ("Delta") denote contexts.
- We write $\Gamma \Rightarrow s : \sigma$ for "in context Γ, s has type σ".
  - Expressions of this form are called **judgements**.
- Examples:
  - $x : \mathrm{o}2, y : \mathrm{o} \Rightarrow x\ y : \mathrm{o}$,
  - $x : \mathrm{Float} \to \mathrm{Int}, y : \mathrm{Float} \Rightarrow x\ y : \mathrm{Int}$ (assuming ground types Float and Int),
  - $x : \mathrm{o}3, y : \mathrm{o}2 \Rightarrow x\ y : \mathrm{o}2$.
- In case Γ is empty, we write $s : \sigma$ instead of $\emptyset \Rightarrow s : \sigma$.

Contexts

- If Γ, Δ are contexts, $\underset{\sim\sim\sim}{\Gamma, \Delta}$ denotes the concatenation of both contexts, e.g. if
  - $\Gamma \equiv x : \mathrm{o}, y : \mathrm{o2}$,
  - $\Delta \equiv z : \mathrm{o}$

  then
  - Γ, Δ denotes $x : \mathrm{o}, y : \mathrm{o2}, z : \mathrm{o}$,
  - Δ, Γ denotes $z : \mathrm{o}, x : \mathrm{o}, y : \mathrm{o2}$,
  - $\Gamma, u : \mathrm{o}$ denotes $x : \mathrm{o}, y : \mathrm{o2}, u : \mathrm{o}$.

Simply Typed λ-Calculus

Definition of the **simply typed λ-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 Δ.
   - Otherwise $x : \sigma$ is overriden by the assumption on x in Δ.

- 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

$$y : \mathrm{o} \to \mathrm{o} \to \mathrm{o} \Rightarrow \lambda x^{\mathrm{o}}.y\ x : \mathrm{o} \to \mathrm{o} \to \mathrm{o}$$

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

$$y : \mathrm{o} \to \mathrm{o} \to \mathrm{o}, x : \mathrm{o} \Rightarrow y\ x : \mathrm{o} \to \mathrm{o}$$

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

$$y : \mathrm{o} \to \mathrm{o} \to \mathrm{o}, x : \mathrm{o} \Rightarrow y : \mathrm{o} \to \mathrm{o} \to \mathrm{o}$$

  and

$$y : \mathrm{o} \to \mathrm{o} \to \mathrm{o}, x : \mathrm{o} \Rightarrow x : \mathrm{o}$$

Example (Overriding of Assum.)

- We have

$$x : \sigma, x : \tau \Rightarrow x : \tau$$

but **not**

$$x : \sigma, x : \tau \Rightarrow x : \sigma$$

Simply Typed λ-Calculus

2. **Application.**
   If s is of type $\sigma \to \tau$ and t of type σ, depending on context Γ, then $s\ t$ is of type τ under context Γ:

$$\frac{\Gamma \Rightarrow s : \sigma \to \tau \qquad \Gamma \Rightarrow t : \sigma}{\Gamma \Rightarrow s\ t : \tau}\ (\mathrm{Ap})$$

- **Explanation:**
  - Assume we have s of type $\sigma \to \tau$.
    - So s is a function, taking an $x : \sigma$ and returning an element of type τ.
  - Assume we have t is an element of type σ.
  - Then we can apply the function s to this t, written as $s\ t$, and obtain an element of type τ.

Example (Application)

- We continue with our derivation of

$$y : o \to o \to o \Rightarrow \lambda x^{o}.y\ x : o \to o \to o$$

- We have already derived using the assumption rule

$$\begin{array}{lcl} y : o \to o \to o, x : o & \Rightarrow & y : o \to o \to o \\ y : o \to o \to o, x : o & \Rightarrow & x : o \end{array}$$

- Using the application rule we conclude:

$$\frac{y{:}o{\to}o{\to}o, x{:}o{\Rightarrow}y{:}o{\to}o{\to}o \qquad y{:}o{\to}o{\to}o, x{:}o{\Rightarrow}x{:}o}{y : o \to o \to o, x : o \Rightarrow y\ x : o \to o}\ (\mathrm{Ap})$$

Note that $o \to o \to o \equiv o \to (o \to o)$.

Simply Typed λ-Calculus

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

$$\frac{\Gamma, x : \sigma \Rightarrow t : \tau}{\Gamma \Rightarrow \lambda x^\sigma.t : \sigma \to \tau}\,(\mathrm{Abs})$$

- **Explanation:**
  - If we have under assumption $x : \sigma$ shown that $t : \tau$, then we can form a new λ-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

$$y : \mathrm{o} \to \mathrm{o} \to \mathrm{o} \Rightarrow \lambda x^{\mathrm{o}}.y\ x : \mathrm{o} \to \mathrm{o} \to \mathrm{o}$$

- We have already derived

$$\frac{y{:}\mathrm{o}{\to}\mathrm{o}{\to}\mathrm{o}, x{:}\mathrm{o} \Rightarrow y{:}\mathrm{o}{\to}\mathrm{o}{\to}\mathrm{o} \qquad y{:}\mathrm{o}{\to}\mathrm{o}{\to}\mathrm{o}, x{:}\mathrm{o} \Rightarrow x{:}\mathrm{o}}{y : \mathrm{o} \to \mathrm{o} \to \mathrm{o}, x : \mathrm{o} \Rightarrow y\ x : \mathrm{o} \to \mathrm{o}}\ (\mathrm{Ap})$$

- Using abstraction we obtain:

$$\frac{\dfrac{y{:}\mathrm{o}{\to}\mathrm{o}{\to}\mathrm{o}, x{:}\mathrm{o} \Rightarrow y{:}\mathrm{o}{\to}\mathrm{o}{\to}\mathrm{o} \qquad y{:}\mathrm{o}{\to}\mathrm{o}{\to}\mathrm{o}, x{:}\mathrm{o} \Rightarrow x{:}\mathrm{o}}{y : \mathrm{o} \to \mathrm{o} \to \mathrm{o}, x : \mathrm{o} \Rightarrow y\ x : \mathrm{o} \to \mathrm{o}}\ (\mathrm{Ap})}{y : \mathrm{o} \to \mathrm{o} \to \mathrm{o} \Rightarrow \lambda x^{\mathrm{o}}.y\ x : \mathrm{o} \to \mathrm{o} \to \mathrm{o}}\ (\mathrm{Abs})$$

(Note that $\mathrm{o} \to \mathrm{o} \to \mathrm{o} \equiv \mathrm{o} \to (\mathrm{o} \to \mathrm{o})$.)

Rules

- We had three **rules**:

  1. $\Gamma, x : \sigma, \Delta \Rightarrow x : \sigma$
     (where x must not occur in Δ).
  2. $$\frac{\Gamma \Rightarrow s : \sigma \to \tau \qquad \Gamma \Rightarrow t : \sigma}{\Gamma \Rightarrow s\, t : \tau}\,(\mathrm{Ap})$$
  3. $$\frac{\Gamma, x : \sigma \Rightarrow t : \tau}{\Gamma \Rightarrow \lambda x^{\sigma}.t : \sigma \to \tau}\,(\mathrm{Abs})$$

Rules

(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:

$$\frac{\Gamma \Rightarrow s : \sigma \to \tau \qquad \Gamma \Rightarrow t : \sigma}{\Gamma \Rightarrow s\ t : \tau}\ (\mathrm{Ap})$$

It expresses:

- Whenever we have derived $\Gamma \Rightarrow s : \sigma \to \tau$
  - (for arbitrary context Γ, types σ, τ, term s)
- and whenever we derived $\Gamma \Rightarrow t : \sigma$
  - (for the same Γ, σ, but arbitrary term t),
- then we can derive $\Gamma \Rightarrow s\ t : \tau$.

Rules

(3) The next rule is similar:

$$\frac{\Gamma, x : \sigma \Rightarrow t : \tau}{\Gamma \Rightarrow \lambda x^{\sigma}.t : \sigma \to \tau}\,(\mathrm{Abs})$$

It expresses:

- Whenever we have derived $\Gamma, x : \sigma \Rightarrow t : \tau$
  - (for arbitrary context Γ, types σ, τ, variable x and term t),

  then we can derive from this $\Gamma \Rightarrow \lambda x^{\sigma}.t : \sigma \to \tau$.

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 $\mathrm{o2} = \mathrm{o} \to \mathrm{o}$).

$$\frac{x : \mathrm{o} \Rightarrow x : \mathrm{o}}{\lambda x^{\mathrm{o}}.x : \mathrm{o2}}\,(\mathrm{Abs})$$

Example 2

$$\cfrac{\cfrac{\cfrac{x : \mathrm{o2}, y : \mathrm{o} \Rightarrow x : \mathrm{o2} \qquad x : \mathrm{o2}, y : \mathrm{o} \Rightarrow y : \mathrm{o}}{x : \mathrm{o2}, y : \mathrm{o} \Rightarrow x\ y : \mathrm{o}}\ (\mathrm{Ap})}{x : \mathrm{o2} \Rightarrow \lambda y^{\mathrm{o}}.x\ y : \mathrm{o2}}\ (\mathrm{Abs})}{\lambda x^{\mathrm{o2}}.\lambda y^{\mathrm{o}}.x\ y : \mathrm{o3}}\ (\mathrm{Abs})$$

Note that we have the following dependencies in the derived λ-term:

$$(\lambda \mathbf{x}^{\mathrm{o}\mathbf{2}}.\ \underbrace{\lambda y^{\mathrm{o}}.\ \underbrace{\underbrace{\mathbf{x}}_{:\mathrm{o}\mathbf{2}}\ \ \underbrace{y}_{:\mathrm{o}}}_{:\mathrm{o}}}_{:\mathrm{o}\rightarrow\mathrm{o}\,=\,\mathrm{o2}}) : \mathrm{o}\mathbf{2} \rightarrow \mathrm{o2} = \mathrm{o3}$$

Observe how these dependencies correspond to the derivation above.

β-Reduction

- β-reduction for typed λ-terms is defined as for untyped λ-terms.
  - One has only to carry around the types as well.
  - Formally we have

$$(\lambda x^{\sigma}.t)\ s \longrightarrow t[x := s]$$

    or using the alternative notation for typed λ-terms

$$(\lambda(x : \sigma).t)\ s \longrightarrow t[x := s]$$

  - And as before β-reduction can be applied to any subterm.
    - A subterm $(\lambda x^{\sigma}.t)\ s$ of a term s is called a **β-redex** of s.

Example

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

$$
\begin{array}{ll}
 & (\lambda \mathbf{x}^{o3}.\lambda \mathbf{y}^{o2}.\mathbf{x}\ (\mathbf{x}\ \mathbf{y}))\ (\lambda x^{o2}.\lambda y^{o}.x\ (x\ y)) \\
\longrightarrow & \lambda \mathbf{y}^{o2}.(\lambda x^{o2}.\lambda y^{o}.x\ (x\ y))\ ((\lambda x^{o2}.\lambda y^{o}.x\ (x\ y))\ \mathbf{y}) \\
\equiv & \lambda \mathrm{y}^{o2}.(\lambda \mathrm{x}^{o2}.\lambda \mathrm{y}^{o}.\mathrm{x}\ (\mathrm{x}\ \mathrm{y}))\ ((\lambda \mathbf{x}^{o2}.\lambda \mathbf{y}^{o}.\mathbf{x}\ (\mathbf{x}\ \mathbf{y}))\ y) \\
=_\alpha & \lambda \mathrm{y}^{o2}.(\lambda \mathrm{x}^{o2}.\lambda \mathrm{y}^{o}.\mathrm{x}\ (\mathrm{x}\ \mathrm{y}))\ ((\lambda \mathbf{x}^{o2}.\lambda \mathbf{z}^{o}.\mathbf{x}\ (\mathbf{x}\ \mathbf{z}))\ y) \\
\longrightarrow & \lambda \mathrm{y}^{o2}.(\lambda \mathrm{x}^{o2}.\lambda \mathrm{y}^{o}.\mathrm{x}\ (\mathrm{x}\ \mathrm{y}))\ (\lambda \mathbf{z}^{o}.y\ (y\ \mathbf{z})) \\
\equiv & \lambda \mathrm{y}^{o2}.(\lambda \mathbf{x}^{o2}.\lambda \mathbf{y}^{o}.\mathbf{x}\ (\mathbf{x}\ \mathbf{y}))\ (\lambda z^{o}.y\ (y\ z)) \\
=_\alpha & \lambda \mathrm{y}^{o2}.(\lambda \mathbf{x}^{o2}.\lambda \mathbf{u}^{o}.\mathbf{x}\ (\mathbf{x}\ \mathbf{u}))\ (\lambda z^{o}.y\ (y\ z)) \\
\longrightarrow & \lambda \mathrm{y}^{o2}.\lambda \mathbf{u}^{o}.(\lambda z^{o}.y\ (y\ z))\ ((\lambda z^{o}.y\ (y\ z))\ \mathbf{u}) \\
\equiv & \lambda \mathrm{y}^{o2}.\lambda \mathrm{u}^{o}.(\lambda \mathrm{z}^{o}.\mathrm{y}\ (\mathrm{y}\ \mathrm{z}))\ ((\lambda \mathbf{z}^{o}.\mathbf{y}\ (\mathbf{y}\ \mathbf{z}))\ u) \\
\longrightarrow & \lambda \mathrm{y}^{o2}.\lambda \mathrm{u}^{o}.(\lambda \mathbf{z}^{o}.\mathbf{y}\ (\mathbf{y}\ \mathbf{z}))\ (\mathbf{y}\ (\mathbf{y}\ u)) \\
\equiv & \lambda \mathrm{y}^{o2}.\lambda \mathrm{u}^{o}.(\lambda \mathbf{z}^{o}.\mathbf{y}\ (\mathbf{y}\ \mathbf{z}))\ (y\ (y\ u)) \\
\longrightarrow & \lambda \mathrm{y}^{o2}.\lambda \mathrm{u}^{o}.\mathbf{y}\ (\mathbf{y}\ (y\ (y\ u)))
\end{array}
$$

Theorem

- As for the untyped λ-calculus, the simply typed λ-calculus is **confluent**.
- The simply typed λ-calculus is **strongly normalising**.
- Therefore every typed λ-term has a unique normal form, which can be obtained by β-reducing the term by choosing arbitrary β-redexes.
- Furthermore, two λ-terms are β-equal, if their normal forms are equal (up to α-conversion).

(c) The λ-Calculus in Agda

- Agda is based on dependent type theory.
- This extends the simply typed λ-calculus.

The Function Type in Agda

- In Agda one writes $A -> C$ for the **nondependent function type.**
  We write on our slides $\rightarrow$ instead of $->$.
- 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) α, β, γ instead.

Blanks around ->

- In Agda, there needs to be a blank before and after $->$,
- but there should be no blank between $-$ and $>$.
- $A->$ without a blank in between is understood as an identifier with name $A->$.
- $->A$ without a blank in between is understood as an identifier with name $->A$.
- Only brackets "$($", "$\{$", "$)$", "$\}$", the symbol "$=$", blanks (and possibly some other symbols not discovered yet by A. Setzer) break identifiers.

λ-Terms in Agda

- In Agda one writes $\backslash(x : A) -\!> r$ for $\lambda(x : A).r$.
- When presenting Agda code we will write λ **(x:A)** $\rightarrow$ **r** for the above, so λ means $\backslash$ and $\rightarrow$ means -> 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

$$\lambda x \rightarrow r$$

instead of

$$\lambda(x : A) \rightarrow r$$

Blanks in $\backslash(x : A){-}{>}\ r$

- In $\backslash(x : A){-}{>}\ r$,
  - 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 ${-}{>}$ 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

$$\begin{array}{lcl} f & : & A \longrightarrow B \\ a & : & A \end{array}$$

  Then we can conclude $f\ a : B$.

- And α- and β-equivalent terms are identified.
  - In Agda,

$$(\lambda x \longrightarrow x)\ a = a \quad .$$

    - So if $B\ a$ is a type depending on a, and we have $b : B\ a$ then we have as well

$$b : B\ ((\lambda x \longrightarrow x)\ a) \quad .$$

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 λ-calculus, is called for historic reasons Set.
- So in order to introduce a ground type A we write:

$$\mathrm{postulate}\ A : \mathrm{Set}$$

Postulate

- 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:

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

See **examplePostulate1.agda**

Basic λ-Terms

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

- 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:
  $$\begin{aligned} b &: B \\ b &= f\ a \end{aligned}$$
  Please note that blanks around "$=$".

Basic λ-Terms

$\mathrm{postulate}\ A : \mathrm{Set}$
$\mathrm{postulate}\ B : \mathrm{Set}$.
$\mathrm{postulate}\ f : A \to B$.
$\mathrm{postulate}\ a : A$.

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

- We can as well introduce $g := \lambda x^A.x : A \to A$ as follows:

$$g \quad : A \to A$$
$$g \quad = \ \lambda x \to x$$

- Note that there needs to be blanks around "$=$".

See **examplePostulate2.agda**

λ-Terms

$\mathrm{postulate}\ A : \mathrm{Set}$
$\mathrm{postulate}\ B : \mathrm{Set}$.
$\mathrm{postulate}\ f : A \to B$.
$\mathrm{postulate}\ a : A$.

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

$$\begin{aligned} g &: A \to A \\ g\ a &= a \end{aligned}$$

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

See **examplePostulate3.agda**

Equivalence of the two Notations

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

$$
\begin{array}{lcl}
\mathrm{postulate}\ A & : & \mathrm{Set} \\
g & : & A \to A \\
g & = & \lambda(a : A) \to a \\
g' & : & A \to A \\
g'\ a & = & a
\end{array}
$$

exampleEquivalenceLambdaNotations1.agda

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:

$$\mathrm{postulate}\ P : (A \rightarrow A) \rightarrow \mathrm{Set}$$

- If we define now

$$\begin{aligned} f &: P\ g \rightarrow P\ g' \\ f\ x &= x \end{aligned}$$

  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

λ-Notation in Agda

- In most cases, it is easier to use the second way of introducing λ-terms.
- However, λ-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:

$$\mathrm{map}\ \mathrm{S}\ (\mathrm{two} :: (\mathrm{three} :: [\,]))$$

  The **result** is

$$(\mathrm{three} :: (\mathrm{four} :: [\,]))$$

λ-Notation in Agda

- Here the elements of $\mathrm{NatList}$ are
  - $[\,]$ denoting the empty list,
  - and if $n : \mathbb{N}$, $l : \mathrm{NatList}$, then $n :: l : \mathrm{NatList}$.

  See **exampleMapAppliedToList.agda**.

Refinement

- Assume the following Agda code

$$\begin{aligned}
&\text{postulate } A : \text{Set}\\
&\text{postulate } B : \text{Set}\\
&\text{postulate } f : A \to B\\
&\text{postulate } a : A\\
&b \quad : \quad B\\
&b \quad = \quad \{!\ \ !\}
\end{aligned}$$

- 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 ($f : A \to B$).

Refinement

$$
\begin{array}{l}
\text{postulate } A : \mathrm{Set} \\
\text{postulate } B : \mathrm{Set} \\
\text{postulate } f : A \to B \\
\text{postulate } a : A \\
b \quad : \quad B \\
b \quad = \quad \{!\ \ !\}
\end{array}
$$

- 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 $\{!\ \ !\} : A$

Refinement

$$\begin{array}{l}\text{postulate } A : \mathrm{Set} \\ \text{postulate } B : \mathrm{Set} \\ \text{postulate } f : A \to B \\ \text{postulate } a : A \\ b \quad : \quad B \\ b \quad = \quad f \; \{! \;\; !\}\end{array}$$

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

exampleSimpleDerivation1.agda

Introducing New Types

- In the λ-calculus, we introduced abbreviations for types, like $\mathrm{o2} = \mathrm{o} \to \mathrm{o}$
- We can do the same in Agda (**exampleTypeAbbreviations.agda**):

$$\text{postulate } A : \mathrm{Set}$$

$$\begin{array}{lcl} A2 & : & \mathrm{Set} \\ A2 & = & A \to A \end{array}$$

$$\begin{array}{lcl} A3 & : & \mathrm{Set} \\ A3 & = & A2 \to A2 \end{array}$$

$$\begin{array}{lcl} a2 & : & A2 \\ a2 & = & \lambda x \to x \end{array}$$

$$\begin{array}{lcl} a3 & : & A3 \\ a3 & = & \lambda x \to x \end{array}$$

Introducing New Types

$$\text{postulate } A : \text{Set}$$

$$\begin{array}{lcl} A2 & : & \text{Set} \\ A2 & = & A \rightarrow A \end{array}$$

$$\begin{array}{lcl} a2 & : & A2 \\ a2 & = & \lambda(x : A) \rightarrow x \end{array}$$

- In the above example we have that the type of $a2$ is as well $A \rightarrow A$, since both types are equal: Although $a2$ is of type $A2$ instead of $A \rightarrow A$, we can define

$$\begin{array}{lcl} a2' & : & A \rightarrow A \\ a2' & = & a2 \end{array}$$

Introducing New Types

- We can as well check that $A \to A$ and $A2$ are the same by applying main menu
  Compute normal form C-c C-n to $A2$
  - We obtain $A \to A$.

Derivations in Agda

- In Agda, rules are implicit.
- The rule

$$\frac{f : A \to B \qquad a : A}{f\ a : B}\ (\mathrm{Ap})$$

corresponds to the following:

- Assume we have introduced:
  - $f : A \to B$, $a : A$.

  and want to solve the goal

$$\begin{aligned} b &: B \\ b &= \{!\ \ !\} \end{aligned}$$

exampleSimpleDerivation2.agda

Derivations in Agda (Cont.)

- Then we can fill this goal by typing in $f\ a$:
- $b = \{!\ f\ a\ !\}$
- If we then choose goal-menu **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

$$
\begin{aligned}
\text{let } a_1 &: A_1 \\
a_1 &= s_1 \\
a_2 &: A_2 \\
a_2 &= s_2 \\
&\cdots \\
a_n &: A_n \\
a_n &= s_n \\
\text{in } t &
\end{aligned}
$$

Let expressions in Agda (Cont.)

- This means that we introduce new local constants
  $a_1 : A_1$ s.t. $a_1 = s_1$,
  $a_2 : A_2$ s.t. $a_2 = s_2$,
  $\ldots$,
  $a_n : A_n$ 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 $a_0, \ldots, a_{i-1}$.

Simple Example

The following function computes $(n + n) * (n + n)$ for $n : \mathbb{N}$:

$$
\begin{array}{lcl}
f & : & \mathbb{N} \longrightarrow \mathbb{N} \\
f\ n & = & \mathrm{let}\ m \quad : \mathbb{N} \\
 & & \qquad m \quad = n + n \\
 & & \mathrm{in}\ m * m
\end{array}
$$

See **exampleLetExpression.agda**

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

- Using let, $n + n$ is computed only once,
- without let, we have to compute it twice.

Example

- As an example we define, assuming $A : \mathrm{Set}$ as a postulate, a function

$$f : ((A \to A) \to A) \to A$$

- We start with the goal

$$\begin{array}{rcl} f & : & ((A \to A) \to A) \to A \\ f & = & \{!\ \ !\} \end{array}$$

Example

$$
\begin{array}{lcl}
f & : & ((A \rightarrow A) \rightarrow A) \rightarrow A \\
f & = & \{!\ \ !\}
\end{array}
$$

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

$$
\begin{array}{lcl}
f & : & ((A \rightarrow A) \rightarrow A) \rightarrow A \\
f\ a{-}a{-}a & = & \{!\ \ !\}
\end{array}
$$

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

$$\begin{array}{rcl} f & : & ((A \rightarrow A) \rightarrow A) \rightarrow A \\ f\ a{-}a{-}a & = & \text{let}\ \begin{array}{rcl} a{-}a & : & \{!\ \ !\} \\ a{-}a & = & \{!\ \ !\} \end{array} \\ & & \text{in}\ \{!\ \ !\} \end{array}$$

- 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

$$\begin{array}{rcl} f & : & ((A \rightarrow A) \rightarrow A) \rightarrow A \\ f\ a{-}a{-}a & = & \text{let}\ \begin{array}{rcl} a{-}a & : & A \rightarrow A \\ a{-}a & = & \{!\ \ !\} \end{array} \\ & & \text{in}\ \{!\ \ !\} \end{array}$$

Example

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

$$\lambda a \to ?$$

and use refine or give and obtain

$$\begin{array}{lcl} f & : & ((A \to A) \to A) \to A \\ f\ a{-}a{-}a & = & \text{let}\ \begin{array}{lcl} a{-}a & : & A \to A \\ a{-}a & = & \lambda a \to \{!\ \ !\} \end{array} \\ & & \text{in}\ \{!\ \ !\} \end{array}$$

Example

- We solve the first goal by typing in a and using **Refine** and have completed the let-expression:

$$\begin{array}{lcl} f & : & ((A \rightarrow A) \rightarrow A) \rightarrow A \\ f\ a{-}a{-}a & = & \mathrm{let}\ \begin{array}{lcl} a{-}a & : & A \rightarrow A \\ a{-}a & = & \lambda a \rightarrow a \end{array} \\ & & \mathrm{in}\ \{!\ \ !\} \end{array}$$

Example

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

$$\begin{array}{lcl}
f & : & ((A \to A) \to A) \to A \\
f\ a{-}a{-}a & = & \mathrm{let}\ \begin{array}{lcl} a{-}a & : & A \to A \\ a{-}a & = & \lambda a \to a \end{array} \\
 & & \mathrm{in}\ a{-}a{-}a\ a{-}a
\end{array}$$

- See **exampleLetExpression2.agda**

(d) Logic with Implication

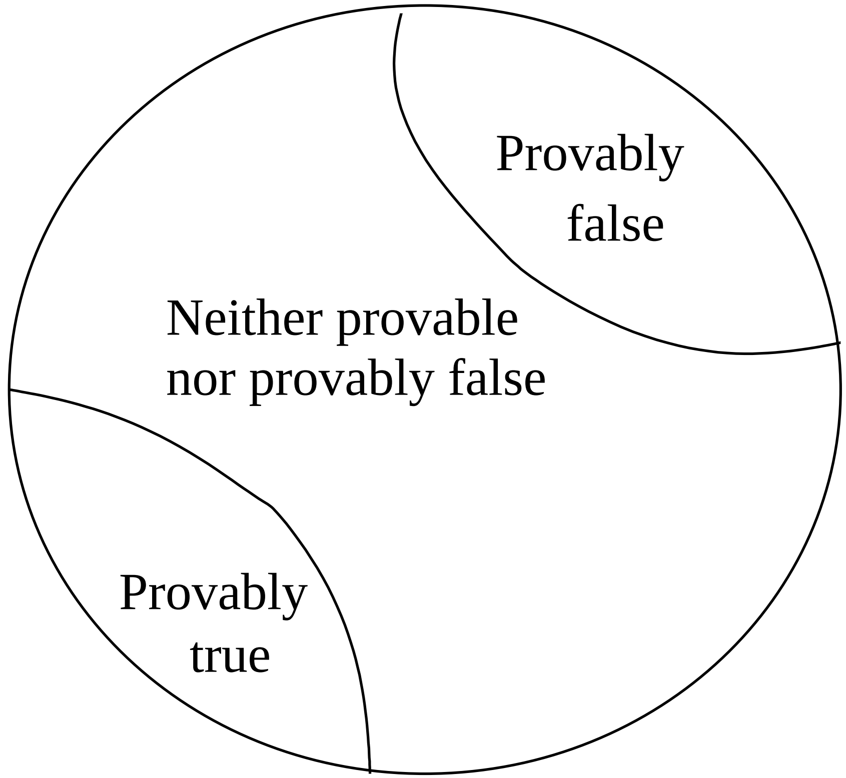
Propositions as Types

- 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
  - $\mathrm{Sorted}\ l$ will be a type,
  - $p : \mathrm{Sorted}\ l$ will be a witness (**proof**) that $\mathrm{Sorted}\ l$ holds.

Constructive Logic

- If $p : \mathrm{Sorted}\ l$ holds, then l should be sorted.
- If we have a proof $p : \neg(\mathrm{Sorted}\ l)$ then l should be not sorted.
  - Negation $\neg$ will be introduced later.
- If we know neither that $p : \mathrm{Sorted}\ l$ nor that $p : \neg\ (\mathrm{Sorted\ l})$, 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 $p : \neg A$.

Picture


Provably
false
Neither provable
nor provably false
Provably
true

Postulates as Formulae

- If we postulate $A : \mathrm{Set}$, 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 $((A \wedge B) \vee C) \to A$).
  - Formulae like $((A \wedge B) \vee C) \to A$) might be generally true (like $A \to A$), or might be true (like $A \vee C$) if certain of its propositional variables are provably true and others are provably false.
- If we postulate $A : \mathrm{Set}$, 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:

$$
\begin{array}{lll}
\mathrm{postulate}\ \ \mathrm{Person} & : & \mathrm{Set} \\
\mathrm{postulate}\ \ \mathrm{john} & : & \mathrm{Person} \\
\mathrm{postulate}\ \ \mathrm{mary} & : & \mathrm{Person} \\
\mathrm{postulate}\ \ \mathrm{IsStudent} & : & \mathrm{Person} \to \mathrm{Set} \\
\mathrm{postulate}\ \ \mathrm{maryIsStudent} & : & \mathrm{IsStudent}\ \mathrm{mary}
\end{array}
$$

Constructive Logic

- Proofs in dependent type theory will have always a **constructive meaning**.
- In case of implication the constructive meaning of a proof of $a\text{–}b : A \rightarrow B$ 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 $a\text{–}b : A \rightarrow B$ is a function mapping proofs of A to proofs of B.
  - This is nothing but the function type $A \rightarrow B$.

Example 1 (Implication)

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

Example 2 (Implication)

- $\lambda(x : A \to B).\lambda(y : A).x\ y$ is a proof of $(A \to B) \to A \to B$:
  - Assume a proof $x : A \to B$.
    - 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 $x : A \to B$, 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 λ-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.

Shorter Proof

- We could have given the following shorter proof of $(A \to B) \to A \to B$:

$$\lambda(x : A \to B).x : (A \to B) \to (A \to B)$$

- Note that $(A \to B) \to A \to B$ and $(A \to B) \to (A \to B)$ are the same.
- The above given λ-term corresponds to the following proof:
  - Assume $A \to B$.
  - Then the conclusion, namely $A \to B$ holds.

Curry Howard Isomorphism

- That one can write proofs as typed λ-terms is often referred to as well as the **Curry-Howard Isomorphism.**
  - Typed λ-terms are nothing but proofs of the formula given by their type!!

(e) Implicational Logic in Agda

- 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:

$$
\begin{array}{llcl}
\mathrm{postulate} & \mathrm{Person} & : & \mathrm{Set} \\
\mathrm{postulate} & \mathrm{john} & : & \mathrm{Person} \\
\mathrm{postulate} & \mathrm{mary} & : & \mathrm{Person} \\
\mathrm{postulate} & \mathrm{IsStudent} & : & \mathrm{Person} \rightarrow \mathrm{Set} \\
\mathrm{postulate} & \mathrm{maryIsStudent} & : & \mathrm{IsStudent\ mary} \\
\mathrm{postulate} & \mathrm{implication} & : & \mathrm{IsStudent\ mary} \rightarrow \mathrm{IsStudent\ joh}
\end{array}
$$

Example

- Then we can prove that John is a student:

$$
\begin{array}{lcl}
\mathrm{Lemma1} & : & \mathrm{Set} \\
\mathrm{Lemma1} & = & \mathrm{IsStudent\ john} \\
\\
\mathrm{proof{-}lemma1} & : & \mathrm{Lemma1} \\
\mathrm{proof{-}lemma1} & = & \mathrm{implicaton\ maryIsStudent}
\end{array}
$$

maryjohn1.agda

Example (Cont.)

- Note that we do not make use of the assumption x in the proof of Lemma1.
- If we added a new person $\mathrm{barbara}$ and tried to prove in the above situation the following wrong Lemma 2:

$$
\begin{array}{lcl}
\mathrm{postulate\ barbara} & : & \mathrm{Person} \\
\mathrm{Lemma2} & : & \mathrm{Set} \\
\mathrm{Lemma2} & = & \mathrm{IsStudent\ john} \rightarrow \mathrm{IsStudent\ barbara} \\
\\
\mathrm{proof{-}lemma2} & : & \mathrm{Lemma2} \\
\mathrm{proof{-}lemma2} & : & \{!\ \ !\}
\end{array}
$$

we will fail.

Example (Cont.)

- We can use a λ-abstraction

$$\begin{array}{rcl} \mathrm{proof{-}lemma2} & : & \mathrm{Lemma2} \\ \mathrm{proof{-}lemma2} & = & \lambda(x : \mathrm{IsStudent\ john}) \rightarrow \{!\ \ !\} \end{array}$$

- But there is no way of solving this goal (except by using full recursion, i.e. by calling recursively $\mathrm{proof{-}lemma2}$, 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**

Example2

- Assume postulates $A : \mathrm{Set}$, $B : \mathrm{Set}$.
- We can introduce the formula (or set) expressing $A \to (A \to B) \to B$ as follows:

$$\begin{aligned} \mathrm{Lemma1} &: \mathrm{Set} \\ \mathrm{Lemma1} &= A \to (A \to B) \to B \end{aligned}$$

- In order to prove Lemma1 we make the following goal:

$$\begin{array}{lcl} \mathrm{lemma1} & : & \mathrm{Lemma1} \\ \mathrm{lemma1} & = & \{!\ \ !\} \end{array}$$

Example 2

$$\begin{array}{lcl} \mathrm{Lemma1} & : & \mathrm{Set} \\ \mathrm{Lemma1} & = & A \to (A \to B) \to B \\ \mathrm{lemma1} & : & \mathrm{Lemma1} \\ \mathrm{lemma1} & = & \{!\ \ !\} \end{array}$$

- The type of the goal is $A \to (A \to B) \to B$.
- When the type of goal is an implication, it is usually shown
  - unless one has an assumption which matches the goal directly

  by λ-abstracting from the premises of the implication.
- Instead of introducing a λ-abstraction, we apply $\mathrm{lemma1}$ to variables a (of type A and $a - b$ (of type $A \to B$).

Example 2

- One obtains:

$$\begin{array}{lcl} \mathrm{lemma1} & : & \mathrm{Lemma1} \\ \mathrm{lemma1}\ a\ a{-}b & = & \{!\ \ !\} \end{array}$$

- Lemma1 was $A \to (A \to B) \to B$,
- we have abstracted from A and $A \to B$,
- so the type of the goal is the conclusion of the implication, namely B.

Example 2

$$\begin{aligned} \mathrm{lemma1} \quad &: \mathrm{Lemma1} \\ &= \lambda(a : A) \to \lambda(a\text{--}b : A \to B) \to \{!\ \ !\} \end{aligned}$$

Type of goal is B

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

Example 2

- We obtain the following proof:

$$
\begin{array}{lcl}
\mathrm{lemma1} & : & \mathrm{Lemma1} \\
\mathrm{lemma1}\ a\ a{-}b & = & a{-}b\ a
\end{array}
$$

- This is exactly the same as introducing a λ-term of type $A \to (A \to B) \to B$.
- See **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 $a-b\ a$ is an element of B,
  - therefore the typing is correct.

Recursive Definitions

- The type checker in Agda allows recursive definitions. For instance, the following passes the type checker:

$$\begin{array}{rcl} a & : & A \\ a & = & a \end{array}$$

- Necessary, since for instance the definition of $+$ is necessarily recursive, i.e. will make use of $+$:

$$\begin{array}{lcl} \_+\_ & : & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n + \mathrm{Z} & = & n \\ n + \mathrm{S}\ m & = & \mathrm{S}\ (n + m) \end{array}$$

Recursive Definitions and Proofs

- Recursive definitions spoil the principle of propositions as types:

$$\begin{array}{ccc} a & : & A \\ a & = & a \end{array}$$

  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

$$a \longrightarrow a \longrightarrow a \longrightarrow a \longrightarrow \cdots$$

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.

Termination Checker

- Agda has a builtin 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:

$$
\begin{array}{lcl}
\boxed{a} & : & A \\
a & = & \boxed{a}
\end{array}
$$

Termination Checker

- 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.

Limitations of the Termination Chec

- 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

- $$\begin{aligned} a &: A \\ a &= a \end{aligned}$$
  will not pass the termination checker.
- $$\begin{aligned} f &: A \to A \\ f\ a &= a \end{aligned}$$
  will pass the termination checker.
- $$\begin{aligned} \mathrm{lemma} &: (A \to B) \to A \to B \\ \mathrm{lemma}\ a - b\ a &= \mathrm{lemma}\ a{-}b\ a \end{aligned}$$
  will not pass the termination checker.

Examples

- $$\begin{array}{lcl} \mathrm{lemma} & : & (A \to B) \to A \to B \\ \mathrm{lemma}\ a\!-\!b\ a & = & a\!-\!b\ a \end{array}$$

  passes the termination checker.

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.

(f) More on the Typed λ-Calculus

The η-Rule

- If we have a function $f : \sigma \to \tau$, then this function applied to $a : \sigma$ gives result $f\ a$.
- If we apply $\lambda x^\sigma.f\ x : \sigma \to \tau$ 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 $f =_\beta \lambda x.f\ x$.

The η-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.
- η-expansion expresses that subterms $t : \sigma \to \tau$ can be η-expanded to $\lambda x.t\ x$ (where x does not occur free in t).
- Then any $f : \sigma \to \tau$ is always equal to $\lambda x.f\ x$ w.r.t. β, η-reduction (where x is fresh).
- One needs to restrict η-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 η-rule in Agda.

The η-Rule

- 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:

$$t \longrightarrow \lambda x.t\ x \equiv \lambda x.(\lambda y.s)\ x \longrightarrow_\beta \lambda x.s[y := x] \equiv t \quad ,$$

  - 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:

$$t\ r \longrightarrow (\lambda x.t\ x)\ r \longrightarrow_\beta t\ r$$

  - All other terms can be expanded without obtaining a new redex.

η-Expansion

- **η-expansion** (or **η-rule**) is the rule which expands one subterm of a λ-term
  - of the form $r : \sigma \to \tau$
  - 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
    - $r \longrightarrow_\eta s$ for s is obtained from r by the η-rule,
    - $r \longrightarrow_{\beta,\eta} s$ for s is obtained from r by using

      β-reduction or η-expansion.
    - Notions like $\longrightarrow^*_{\beta,\eta}$, $=_{\beta,\eta}$, $=_\eta$, **β,η-normal form**,

      etc. are to be understood correspondingly.

Example

- Assume $f : \mathrm{o}^3$. Then

$$
\begin{array}{ll}
r := (\lambda f^{\mathrm{o}3}.\lambda x^{\mathrm{o}2}.f\ x)\ f & \\
\longrightarrow_\beta \lambda x^{\mathrm{o}2}.f\ x & \\
\longrightarrow_\eta \lambda x^{\mathrm{o}2}.\lambda y^{\mathrm{o}}.f\ x\ y & \text{(by } \eta\text{-expanding } f\ x : \mathrm{o}2 \\
& \ \text{to } \lambda y^{\mathrm{o}}.f\ x\ y) \\
\longrightarrow_\eta \lambda x^{\mathrm{o}2}.\lambda y^{\mathrm{o}}.f\ (\lambda z^{\mathrm{o}}.x\ z)\ y =: s & \text{(by } \eta\text{-expanding } x : \mathrm{o}^2 \\
& \ \text{to } \lambda z^{\mathrm{o}}.x\ z)
\end{array}
$$

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

Example

$$r := \lambda f^{o3}.\lambda x^{o2}.f\ x)\ f \longrightarrow_{\beta} \quad \lambda x^{o2}.f\ x$$
$$\longrightarrow_{\eta}^{*} \quad \lambda x^{o2}.\lambda y^{o}.f\ (\lambda z^{o}.x\ z)\ y =: s$$

- There are no more η-expansions or β-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 η-expanded.
  - z and y are of ground type and therefore not to be η-expanded.

Example

$$\begin{aligned} r := \lambda f^{\mathrm{o}3}.\lambda x^{\mathrm{o}2}.f\ x)\ f &\longrightarrow_{\beta} \quad \lambda x^{\mathrm{o}2}.f\ x \\ &\longrightarrow_{\eta}^{*} \quad \lambda x^{\mathrm{o}2}.\lambda y^{\mathrm{o}}.f\ (\lambda z^{\mathrm{o}}.x\ z)\ y =: s \end{aligned}$$

- Because s cannot be expanded any further, it is the β,η-**normal form** of r.
- Since $f \longrightarrow_{\eta} \lambda x^{\mathrm{o}2}.f\ x$, the term s is as well the β,η-normal form of $f : \mathrm{o}3$.

Example 2

If we replace in the above example o by $\mathrm{o}2$ (and therefore $\mathrm{o}2$ by $\mathrm{o}3$ and $\mathrm{o}3$ by $\mathrm{o}4$) we obtain

$$
\begin{array}{l}
(\lambda f^{\mathrm{o}4}.\lambda x^{\mathrm{o}3}.f\ x)\ f \\
\longrightarrow_{\beta} \lambda x^{\mathrm{o}3}.f\ x \\
\longrightarrow_{\eta} \lambda x^{\mathrm{o}3}.\lambda y^{\mathrm{o}2}.f\ x\ y \\
\longrightarrow_{\eta} \lambda x^{\mathrm{o}3}.\lambda y^{\mathrm{o}2}.\lambda z^{\mathrm{o}}.f\ x\ y\ z \\
\longrightarrow_{\eta} \lambda x^{\mathrm{o}3}.\lambda y^{\mathrm{o}2}.\lambda z^{\mathrm{o}}.f\ (\lambda u^{\mathrm{o}2}.x\ u)\ y\ z \\
\longrightarrow_{\eta} \lambda x^{\mathrm{o}3}.\lambda y^{\mathrm{o}2}.\lambda z^{\mathrm{o}}.f\ (\lambda u^{\mathrm{o}2}.\lambda v^{\mathrm{o}}.x\ u\ v)\ y\ z \\
\longrightarrow_{\eta} \lambda x^{\mathrm{o}3}.\lambda y^{\mathrm{o}2}.\lambda z^{\mathrm{o}}.f\ (\lambda u^{\mathrm{o}2}.\lambda v^{\mathrm{o}}.x\ (\lambda w^{\mathrm{o}}.u\ w)\ v)\ y\ z \\
\longrightarrow_{\eta} \lambda x^{\mathrm{o}3}.\lambda y^{\mathrm{o}2}.\lambda z^{\mathrm{o}}.f\ (\lambda u^{\mathrm{o}2}.\lambda v^{\mathrm{o}}.x\ (\lambda w^{\mathrm{o}}.u\ w)\ v)\ (\lambda u^{\mathrm{o}}.y\ u)\ z
\end{array}
$$

which is as well the β, η-normal form of $f : \mathrm{o}4$.

Intuitive Application of η-Expansion

- Intuitively, η-expansion for terms in β-normal form is obtained as follows:
  - Consider subterms

    $$r := t_1\ t_2\ \cdots\ t_n$$

    of the term to be η-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 $r : \alpha \to \beta$ it is an η-redex.
    - Otherwise r is of ground type and not an η-redex.

Intuitive Application of η-Expansion

- If

$$r := t_1 \; t_2 \; \cdots \; t_n$$

  is an η-redex, expand it to

$$\lambda x^{\alpha}.t_1 \; t_2 \; \cdots \; t_n \; x \quad .$$

- Continue until there are no η-redexes left.

Theorem

- The typed λ-calculus with β-reduction and η-expansion is confluent and strongly normalising.

η-Rule

- With the η-rule, we obtain that if $r : \sigma \to \tau$, then $r =_{\beta,\eta} \lambda x^\sigma.r\ x$.
  - If $r : \sigma \to \tau$ is of the form $\lambda u^\sigma.t$ then we have $r =_\beta \lambda x^\sigma.r\ x$:

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

  - Otherwise $r \longrightarrow_\eta \lambda x^\sigma.r\ x$.
- Therefore one can say the η rule expresses: **every element of a function type is of the form $\lambda x.$something**.

η-Reduction

- In the literature one often uses instead of η-expansion **η-reduction**, which allows to reduce $\lambda x^{\sigma}.r\ x$ 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.

η-Rule in Agda

- In Agda syntax, the η**-rule** states that if

  $$f : A \rightarrow B$$

  then

  $$f = \lambda(x : A) \rightarrow f\ x \quad .$$

  - The η-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

  $$\Gamma, \Delta \Rightarrow t : \sigma \quad .$$

  Then we have as well

  $$\Gamma, x : \tau, \Delta \Rightarrow t : \sigma \quad ,$$

  provided $\Gamma, x : \tau, \Delta$ is a context (i.e. provided x does not occur in Γ, Δ).
- The process of extending the context is called **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

$$\dfrac{\dfrac{y : \mathrm{o}, x : \mathrm{o} \Rightarrow x : \mathrm{o}}{y : \mathrm{o} \Rightarrow \lambda x^{\mathrm{o}}.x : \mathrm{o}2}\,(\mathrm{Abs}) \qquad y : \mathrm{o} \Rightarrow y : \mathrm{o}}{y : \mathrm{o} \Rightarrow (\lambda x^{\mathrm{o}}.x)\ y : \mathrm{o}}\,(\mathrm{Ap})$$

we obtain a derivation of

$$y : \mathrm{o}, x : \mathrm{o} \Rightarrow (\lambda x^{\mathrm{o}}.x)\ y : \mathrm{o}$$

by inserting in each context in the derivation, after $y : \mathrm{o}$ the context $x : \mathrm{o}$.

Example (Weakening)

$$\dfrac{\dfrac{y : \mathrm{o}, x : \mathrm{o} \Rightarrow x : \mathrm{o}}{y : \mathrm{o} \Rightarrow \lambda x^{\mathrm{o}}.x : \mathrm{o}2}\,(\mathrm{Abs}) \qquad y : \mathrm{o} \Rightarrow y : \mathrm{o}}{y : \mathrm{o} \Rightarrow (\lambda x^{\mathrm{o}}.x)\ y : \mathrm{o}}\,(\mathrm{Ap})$$

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

$$\dfrac{\dfrac{y : \mathrm{o}, x : \mathrm{o}, x : \mathrm{o} \Rightarrow x : \mathrm{o}}{y : \mathrm{o}, x : \mathrm{o} \Rightarrow \lambda x^{\mathrm{o}}.x : \mathrm{o}2}\,(\mathrm{Abs}) \qquad y : \mathrm{o}, x : \mathrm{o} \Rightarrow y : \mathrm{o}}{y : \mathrm{o}, x : \mathrm{o} \Rightarrow (\lambda x^{\mathrm{o}}.x)\ y : \mathrm{o}}\,(\mathrm{Ap})$$

Weakening

- Because of the possibility of weakening, we will usually omit unused parts of contexts.
- So a derivation of $x : \mathrm{o2}, y : \mathrm{o} \Rightarrow x\ (x\ y) : \mathrm{o}$, which in full reads as follows

$$\dfrac{x{:}\mathrm{o2}, y{:}\mathrm{o} \Rightarrow x{:}\mathrm{o2} \qquad \dfrac{x{:}\mathrm{o2}, y{:}\mathrm{o} \Rightarrow x{:}\mathrm{o2} \qquad x{:}\mathrm{o2}, y{:}\mathrm{o} \Rightarrow y{:}\mathrm{o}}{x : \mathrm{o2}, y : \mathrm{o} \Rightarrow x\ y : \mathrm{o}}\,(\mathrm{Ap})}{x : \mathrm{o2}, y : \mathrm{o} \Rightarrow x\ (x\ y) : \mathrm{o}}\,(\mathrm{Ap})$$

will usually be presented as follows:

$$\dfrac{x{:}\mathrm{o2} \Rightarrow x{:}\mathrm{o2} \qquad \dfrac{x{:}\mathrm{o2} \Rightarrow x{:}\mathrm{o2} \qquad y{:}\mathrm{o} \Rightarrow y{:}\mathrm{o}}{x : \mathrm{o2}, y : \mathrm{o} \Rightarrow x\ y : \mathrm{o}}\,(\mathrm{Ap})}{x : \mathrm{o2}, y : \mathrm{o} \Rightarrow x\ (x\ y) : \mathrm{o}}\,(\mathrm{Ap})$$

Self-Application

- We introduced the typed λ-calculus, in order to avoid non-normalising terms, as they occur in the untyped λ-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
  - $\Omega \longrightarrow_\beta \Omega$.
- In the following, we will investigate, how self-application is avoided in the typed λ-calculus.

Self-Application

- In the simply typed λ-calculus we cannot assign a type to $\lambda x.x\ x$, i.e. there are no types σ, τ 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 λ-abstraction.
  - Then τ must be equal to $\sigma \to \tau_1$ for some τ_1, and the derivation reads then

$$\frac{x : \sigma \Rightarrow x\ x : \tau_1}{\lambda x^\sigma.x\ x : \sigma \to \tau_1}\,(\mathrm{Abs})$$

Self-Application

$$\frac{x : \sigma \Rightarrow x\ x : \tau_1}{\lambda x^\sigma . x\ x : \sigma \to \tau_1}\ (\mathrm{Abs})$$

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

$$\dfrac{\dfrac{x : \sigma \Rightarrow x : \tau_2 \to \tau_1 \qquad x : \sigma \Rightarrow x : \tau_2}{x : \sigma \Rightarrow x\ x : \tau_1}\ (\mathrm{Ap})}{\lambda x^\sigma . x\ x : \sigma \to \tau_1}\ (\mathrm{Abs})$$

Self-Application

$$\cfrac{\cfrac{x:\sigma \Rightarrow x:\tau_2 \rightarrow \tau_1 \qquad x:\sigma \Rightarrow x:\tau_2}{x:\sigma \Rightarrow x\ x:\tau_1}\,(\mathrm{Ap})}{\lambda x^\sigma.x\ x:\sigma \rightarrow \tau_1}\,(\mathrm{Abs})$$

- 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 τ_2.
- So we cannot find types σ, τ s.t. $\lambda x^\sigma.x\ x:\tau$.