2. \( \lambda \)-Calculus and Term Rewriting

(a) Reductions in Agda.
(b) Reduction systems.
(c) Termination, confluence, normalisation.
(d) Term rewriting systems.
(e) The untyped \( \lambda \)-calculus.
(f) The typed \( \lambda \)-calculus.
(g) The \( \lambda \)-Calculus in Agda.
(h) More on the typed \( \lambda \)-calculus.
(i) The typed \( \lambda \)-calculus with products.
(j) The nondependent product in Agda.
(k) The \( \lambda \)-calculus and term rewriting.
(l) Currying.

(a) Reductions in Agda

Functional programming is essentially based on term reduction:
Assume we introduce the natural numbers as an algebraic data type built from \( 0 \) and \( S \) (this is actual Agda code):
\[
data N = Z \mid S (n :: N)
\]
We write here \( Z \) instead of \( 0 \), since the symbol \( 0 \) will be reserved for the built-in integers.
\( S n \) stands for \( n + 1 \).

Use of :: vs. :

In ordinary type theory, one uses \( a : A \) for “\( a \) is of type \( A \)”.
In Haskell
- types can be inferred automatically, therefore they don’t occur often in Haskell programs;
- lists occur very often in programming in Haskell.
In need for a short notation for forming the result of adding a symbol in front of a list, the symbol “::” was chosen in Haskell:
- \([2, 3]\) is the list with elements 2 and 3.
- \(1 : [2, 3]\) is the result of adding 1 in front of this list, i.e. equal to \([1, 2, 3]\).
**N as a Reduction System**

- So the elements of \( N \) are
  
  \[
  Z \quad S Z \quad S (S Z) \quad S (S (S Z)) \quad \cdots
  \]

- We can now define \( + \) and \( \cdot \) in \( N \) by induction over the definition of \( N \).
- For those with mathematical problems: “Induction over the definition of \( N \)” means roughly case distinction on \( N \) in a terminating way.

**Definition of \((+)_1\)**

\[
(+) \quad (n, m :: N)
\]

\[
:: \quad N
\]

\[
= \quad \text{case } m \text{ of}
\]

\[
(Z) \quad \rightarrow \quad n
\]

\[
(S m') \quad \rightarrow \quad S (n + m')
\]

\((+)\) means that \( + \) can be defined infix, i.e. that we can write \( s + t \) instead of \((+) s t\).

The above means that we have the following reductions:

\[
\begin{align*}
  s + Z & \rightarrow s , \\
  s + S t & \rightarrow S (s + t).
\end{align*}
\]

**Definition of \((+)_2\)**

Note that \( S \) binds more than \( + \).

- So \( S r + s \) reads \( (S r) + s \).
- \( r + S s \) reads \( r + (S s) \).

We have \( 2 + 2 \rightarrow 4 \):

\[
\begin{align*}
  S (S Z) + S (S Z) & \rightarrow S (S (S Z) + S Z) \\
  & \rightarrow S (S (S (S Z) + Z)) \\
  & \rightarrow S (S (S (S Z)))
\end{align*}
\]

(In the above we have used usual conventions about omitting brackets, which are built into Agda).

**Working in Agda**

- Once, Agda is installed, the above can be defined as follows:
- One opens in (X)Emacs a file with extension “.agda”, e.g. “firstAgdaExampleNat.agda”.
- (X)Emacs will switch into Agda mode.
- Now we type in the definition of \( N \):

\[
data N = Z | S (n :: N)
\]

Definitions have to be separated by blank lines, so we leave an empty lines before defining \((+)\).
Agda doesn’t realise any changes in the buffer, unless we load it.

There are two menus for doing this:
- One which we get by right-clicking anywhere in the buffer.
- One by clicking on the word “Agda” in the Panel.

By choosing in one of those menus “Load Buffer” we load it, and get an acknowledgement.

When reloading a buffer, only the current buffer is reloaded, code from other buffers, loaded previously, is kept.

This might lead to confusion, if one moves to a new version of an Ada file, which has the same names for top-level identifiers as a previous version.

Agda will not allow to use the same name twice.

If one doesn’t want to keep the code from previous versions, one has to restart Agda.

Using menu “(Re)Start Agda”.

This is sometimes necessary as well, if there are other problems with Agda.

Goals are numbered by the order in which they were created.

Goals are displayed together with their type in separate buffer called “Goals”.

This can be activated as well by using menu “Show Goals”.

In (X)Emacs mode, goals have a special status.

When typing in text into a goal, the goal expands.

We can start defining (+) by leaving the definition first as a goal:

\[
(+) \quad (n, m :: N) \quad :: \quad N \quad = \quad \{! \} \]

Support of Case Distinction

Now we can insert into the goal the variable we want to make case distinction on (here: \( m \)).

Then we choose when having the active point inside emacs inside the goal right-mouse-click menu “agda-case”, and press return in the mini-buffer.

Then the goal expands and we get a case distinction:

\[
(+) \quad (n, m :: N) \\
:: \quad N \\
= \quad \text{case } m \text{ of} \\
\quad (Z) \quad \rightarrow \quad \{! \} \} \\
\quad (S \ n') \quad \rightarrow \quad \{! \} \}
\]

The variable \( n' \) in \((S \ n')\) is not very convenient.

We replace it by \( m' \), to remind us that it comes from \( m = S \ m' \).

\[
(+) \quad (n, m :: N) \\
:: \quad N \\
= \quad \text{case } m \text{ of} \\
\quad (Z) \quad \rightarrow \quad \{! \} \} \\
\quad (S \ m') \quad \rightarrow \quad \{! \} \}
\]

Support of Case Distinction

We do the same with the second goal and enter \( n + m' \), and do refine.

We obtain

\[
(+) \quad (n, m :: N) \\
:: \quad N \\
= \quad \text{case } m \text{ of} \\
\quad (Z) \quad \rightarrow \quad n \\
\quad (S \ m') \quad \rightarrow \quad n + m'
\]
Errors

- If one had forgotten to reload the buffer when replacing \( n' \) by \( m' \), then Agda will signal an error and report an error message in a buffer called "Agda Error".
- When there is an error, one can use Agda menu "Goto Error" in order to go to a position close to where the error occurred.
- In the above example the cursor would be close to the variable \( m' \) causing the error.
- In order to try this out, you might replace in the above code \( n + m' \) by \( n + k \), and reload the buffer.
- You should get an error, and if you go to the error, the cursor should be close to the wrong variable \( k \).

Indentation Sensitivity

- Agda is indentation sensitive.
- So often instead of having parentheses "\((\text{Code})\)", as in other languages, all lines belonging to \((\text{Code})\) have to be intended more than the surrounding code, and usually in the same way.
- Therefore top level definitions have to start in column1. Otherwise they are considered as being an extension of a previous definition.
- All code belonging to such a definition in later columns has to be intended at least once.

Example: The following causes an error:

```agda
data N = Z | S (n :: N)
(+) (n, m :: N)
:: N
= case m of
  (Z) → n
  (S m') → n + m'
```

We have to type instead:

```agda
data N = Z | S (n :: N)
(+) (n, m :: N)
:: N
= case m of
  (Z) → n
  (S m') → n + m'
```
Definition of (*)

Definition of (*):

(*) \( (n, m :: N) \)
\[ \begin{align*}
&:: N \\
&= \text{case } m \text{ of} \\
&(Z) \quad \rightarrow \quad Z \\
&(S \; m') \quad \rightarrow \quad n * m' + n
\end{align*} \]

This means that we have the following reductions:

\[ s * Z \rightarrow Z, \quad s * S \; t \rightarrow s * t + s . \]

Testing the above in Agda

Testing of reductions in Agda is currently still a bit inconvenient.

Improving it (which is easy) is one of the lecturer’s most wanted improvements of Agda.

We have to create a new definition of a term, but leaving the definition open.

Done by introducing it as a goal, written \{! !\}.

So one adds in Agda:

\[ \text{test} :: N \\
\quad = \quad \{! !\} \]

Complete Agda Code

\text{data N} \quad = \quad Z \mid S \; (n :: N) \\
(+) \quad (n, m :: N) \\
\quad :: N \\
\quad = \quad \text{case } m \text{ of} \\
&(Z) \quad \rightarrow \quad n \\
&(S \; m') \quad \rightarrow \quad S \; (n + m') \\
(*) \quad (n, m :: N) \\
\quad :: N \\
\quad = \quad \text{case } m \text{ of} \\
&(Z) \quad \rightarrow \quad Z \\
&(S \; m') \quad \rightarrow \quad n * m' + n \\
\text{test} :: N \\
\quad = \quad \{! !\} \]
Reduction Systems and Agda

- We want that \(2 + 2\) and \(4\) to be the same.
- In Agda, referring to our self-defined natural numbers, this means that \(S (S Z) + S (S Z)\) and \(S (S (S Z))\) should be the same.
- We have just seen that \(S (S Z) + S (S Z)\) reduces to \(S (S (S Z))\).
- The underlying principle behind this is:
  
  If a term reduces to another term, then these two terms are the same.

Towards General Reduction Systems

- In order to understand Agda better, we will study in the following study general reduction relations.
- Given by a set of Terms \(T\) and a reduction relation \(s \rightarrow t\) between terms \(s\) and \(t\).

(b) Reduction Systems

- A reduction system is a pair \((T, \rightarrow)\) consisting of a set \(T\) (of terms) and a binary relation \(\rightarrow\) on \(T\).
- We write \(s \rightarrow t\) for “\(s, t\) are in relation \(\rightarrow\)” and say usually “\(s\) reduces to \(t\)”.
- Concrete example:
  - Let \(T\) be the set of terms formed from \(0, S, +\) and \(*\) in the usual way.
  - So for instance \(0, S 0\) and \((S 0 + 0\) are elements of \(T\).
  - Let \(\rightarrow\) be the reduction relation defined as before.
  - Then \((T, \rightarrow)\) forms a reduction system.

Example 1 (Reduction System)

- A simple reduction system is \(\mathbb{N}\) with reductions \(n + 1 \rightarrow n\) for \(n \in \mathbb{N}\):
  
  \[
  0 \leftarrow 1 \leftarrow 2 \leftarrow 3 \leftarrow 4 \leftarrow 5
  \]
  
  So we have reductions of the form:
  
  \[
  5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0
  \]
Example 2 (Reduction System)

Another one is \( \mathbb{N} \) with reductions \( n \rightarrow m \) for \( n, m \in \mathbb{N} \) s.t. \( n > m \):

\[
\begin{align*}
0 & \rightarrow 1 & 1 & \rightarrow 2 & 2 & \rightarrow 3 & 3 & \rightarrow 4 \\
\end{align*}
\]

So we have reductions of the form:
\[
23 \rightarrow 11 \rightarrow 3 \rightarrow 1 \rightarrow 0 .
\]

\[\rightarrow^* \text{ and } \leftrightarrow^*\]

In Agda we said we identify two terms which reduce to each other in possible multiple steps.

Therefore we study two concepts:

- One is \( s \rightarrow^* t \), which means that \( s \) reduces to \( t \) in possibly multiple steps.
- When Agda reduces a term \( s \), it returns a term \( t \) s.t. \( s \rightarrow^* t \), and \( t \) cannot reduce any further.

- One is \( s \leftrightarrow^* t \), which is the equality introduced by \( \rightarrow \).
- So Agda identifies terms \( s \) and \( t \) s.t. \( s \leftrightarrow^* t \).

Example 3 (Reduction System)

A third one is \( T = \mathbb{N} \cup \{ \bullet \} \), with reductions \( n + 1 \rightarrow n \) for \( n \in \mathbb{N} \), and \( \bullet \rightarrow n \) for \( n \in \mathbb{N} \):

\[
\begin{align*}
0 & \rightarrow 1 & 1 & \rightarrow 2 & 2 & \rightarrow 3 & 3 & \rightarrow 4 \\
\end{align*}
\]

So we have reductions of the form:
\[
\bullet \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0 .
\]

\[\rightarrow^*\]

If \( (T, \rightarrow) \) is a reduction system, we define

- \( a \rightarrow^* b \) if there exists a (possibly empty) sequence of reductions

\[
a \equiv a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n \equiv b
\]

- By empty reduction we mean: if \( a \equiv b \), then we have \( a \rightarrow^* b \)
- (We write \( \equiv \) for syntactic equality between terms in order to avoid confusion with equality modulo reductions introduced later and denoted by \( = \)).
In order to express the above shorter, one says that \( \rightarrow^* \) is the transitive and reflexive closure of \( \rightarrow \), i.e. the least transitive and reflexive relation containing \( \rightarrow \): 

- \( a \rightarrow b \) implies \( a \rightarrow^* b \).
- \( \rightarrow^* \) is reflexive, i.e. for all \( a \) we have \( a \rightarrow^* a \).
- \( \rightarrow^* \) is transitive, i.e. \( a \rightarrow^* b \rightarrow^* c \) implies \( a \rightarrow^* c \).
- If there is any relation \( \rightarrow' \) with the above 3 properties, then \( x \rightarrow^* y \) implies \( x \rightarrow' y \).

\[ \text{Example} \]

If we take \( \mathbb{N} \) with reductions 
\[ n + 1 \rightarrow n \] for \( n \in \mathbb{N} \):
\[ 0 \iff 1 \iff 2 \iff 3 \iff 4 \iff 5 \]
Then \( 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \), therefore \( 5 \rightarrow^* 2 \).
In general \( n \rightarrow^* m \iff n \geq m \).

If \( (T, \rightarrow) \) is a reduction system, we define 
\[ a \leftrightarrow^* b \iff \text{there exists a (possibly empty) sequence of reductions} \]
\[ a \equiv a_0 \leftrightarrow a_1 \leftrightarrow a_2 \leftrightarrow \cdots \leftrightarrow a_n \equiv b \]
where 
\[ a \leftrightarrow b :\iff (a \rightarrow b \lor b \rightarrow a) \]
Example

If have the following reduction system:

```
   3     4
  0 1   2
```

Then $0 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 2$, therefore $0 \rightarrow^* 2$.

Identification of Elements

If we have a reduction system $(A, \rightarrow)$, one writes $a = \rightarrow b$ or sometimes $a = b$ for $a \rightarrow^* b$.

In order to avoid confusion, we write $a \equiv b$ for $a$ and $b$ are syntactically equival (i.e. consist of the same sequence of symbols).

Determination of $\rightarrow^*$

In general it is infeasible to determine whether $a \rightarrow^* b$ holds.

- One has to check all possible ways of getting from $a$ to $b$, by both using $\rightarrow$ and $\leftarrow$.

- In many cases this can be determined by:
  - Simply reducing $a$ as long as possible to some term $a'$ s.t. $a \rightarrow^* a'$ and s.t. $a'$ has no further reductions, i.e. by “evaluating $a$”.
  - Doing the same with $b$ to some term $b'$.
  - Checking whether $a'$ is identical to $b'$.

- This is possible, if $\rightarrow$ is confluent and strongly normalising (see next subsection).

(c) Termination, Confluence, Normalisation
Strong Normalisation

A reduction system \((A, \rightarrow)\) is **terminating** or **strongly normalising**, iff there is no infinite sequence

\[a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \cdots\]

Examples

- The following reduction system is terminating:

\[
\begin{array}{cccccc}
0 & \leftarrow & 1 & \leftarrow & 2 & \leftarrow & 3 & \leftarrow & 4 & \leftarrow & 5
\end{array}
\]

Any reduction sequence will end in 0 and terminate.

- The following reduction system is not terminating:

\[
\begin{array}{cccccc}
0 & \rightarrow & 1 & \rightarrow & 2 & \rightarrow & 3 & \rightarrow & 4 & \rightarrow & 5
\end{array}
\]

(Take \(0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \cdots\)).

Normal Form and Irreducibility

Let \((A, \rightarrow)\) be a reduction system.

- \(a \in A\) is **irreducible**, if there exists no \(b \in A\) s.t. \(a \rightarrow b\).

- \(b\) is a **normal form** of \(a\) iff \(a \rightarrow^* b\) and \(b\) is irreducible.

- \((A, \rightarrow)\) is **weakly normalising** or **normalising**, if every \(a \in A\) has a normal form.
Example

The following system is weakly normalising, but not strongly normalising:

\[ 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow \cdots \]

- Every \( x \) has a normal form, namely \( \bullet \).
- But there exists an infinite sequence

\[ 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \cdots \]

Example 2

The following system is both weakly normalising and strongly normalising:

\[ 0 \leftarrow 1 \leftarrow 2 \leftarrow 3 \leftarrow 5 \leftarrow \cdots \]

- If one reduces any element as long as possible, one finally ends up with \( \bullet \), which doesn't reduce any further.
- So every element has the same normal form namely \( \bullet \).

Example 3

In the following system \( 0 \) has two normal forms, namely 1 and 2:

\[ 0 \]
\[ 1 \]
\[ 2 \]

This system is both strongly and weakly normalising, but is not confluent. ("Confluent" will be defined later).

Lemma

Let \( (A, \rightarrow) \) be a strongly normalising reduction system. Then \( (A, \rightarrow) \) is weakly normalising.

Proof:
A normal form of \( x \) can be obtained by simply reducing \( x \) as long as possible:
Since \( (A, \rightarrow) \) is strongly normalising, the reduction sequence terminates in some \( y \in A \).

\( y \) is a normal form of \( x \).
**Church-Rosser**

- We say a reduction system \((A, \rightarrow)\) is **confluent** or has the Church-Rosser property iff for all \(x, y, z \in A\) we have:
  - if \(x \rightarrow^* y\) and \(x \rightarrow^* z\),
  - then there exists an \(u\) s.t. \(y \rightarrow^* u\) and \(z \rightarrow^* u\).

**Diamond Property**

- Because of the shape of the picture on the previous slide, the Church-Rosser property is sometimes called as well the Diamond property or Triangle property.
- So Church-Rosser means: Every triangle (or better fork) can be closed to a diamond.

**Theorem**

- If \((A, \rightarrow)\) is confluent, then we have:
  - \(x \leftrightarrow^* y\)
  - iff there exists a \(z\) s.t. \(x \rightarrow^* z \land y \rightarrow^* z\)

**Idea of Proof**

- Common Reduct of \(x\) and \(y\)
Proof of the Theorem

We define
\[ x \downarrow y :\iff \exists z. \left( x \rightarrow^* z \land y \rightarrow^* z \right) . \]

So \( x \downarrow y \) means that \( x \) and \( y \) have a common reduct:

\[ \begin{array}{c}
  x \\
  * \\
  \exists z
\end{array} \quad \begin{array}{c}
  y \\
  * \\
  \exists z
\end{array} \]

Proof of the Theorem

So we have to show
\[ x \rightarrow^* y \iff x \downarrow y \]

\( \iff \) is easy. If \( x \rightarrow^* z, y \rightarrow^* z \), then we get
\[ x \rightarrow^* z \quad \rightarrow^* y \]

and therefore \( x \rightarrow^* y \) (where \( z \rightarrow^* y \iff y \rightarrow^* z \)).

Proof of the Theorem

For the more difficult direction \( \Rightarrow \) we give two proofs:
- One more concrete and intuitive one.
  - Will be presented during the lecture.
- One more abstract one.
  - Will not be presented during the lecture.

First Proof of \( \Rightarrow \)

Assume \( x \leftarrow^* y \).

This means that we have a chain
\[ x \equiv x_0 \leftarrow x_1 \leftarrow x_2 \leftarrow \ldots \leftarrow x_n \equiv y \]

here \( x \leftarrow y \) means \( x \rightarrow y \) or \( x \leftarrow y \).

We write \( x \leftarrow y \) for \( y \rightarrow x \).

Now what we are going to show successively:
- \( x_0 \downarrow x_0 \),
- \( x_0 \downarrow x_1 \),
- \( x_0 \downarrow x_2 \),
- \( \ldots \)
- \( x_0 \downarrow x_n \equiv y \) (the assertion).
First Proof of “⇒”

\[ x \not\equiv y \iff \exists z. (x \rightarrow^* z \land y \rightarrow^* z) . \]

- In order to show this we have to show the following:
  - For the first step, we need to show \( x_0 \not\equiv x_0 \), i.e. in general we need to show
    
    \[ (1) \quad x \not\equiv x \]
  - For the step from \( x_0 \not\equiv x_i \) to \( x_0 \not\equiv x_{i+1} \) we need to show:
    - If \( x_0 \not\equiv x_i \) and \( x_i \equiv x_{i+1} \) then \( x_0 \not\equiv x_{i+1} \).
    - In general we have to show:
      - If \( x \not\equiv y \) and \( y \equiv z \), then \( x \not\equiv z \),
      - in short:
        \[ x \not\equiv y \equiv z \implies x \not\equiv z . \]

Diagram

- That \( x \not\equiv y \equiv z \implies x \not\equiv z \) can be visualised as follows:

\[ x \not\equiv y \equiv z \implies x \not\equiv z \]

First Proof of “⇒”

- (1) \( x \not\equiv x \).
- \( x \not\equiv y \equiv z \implies x \not\equiv z \).

- Since \( y \equiv z \) means \( y \rightarrow z \) or \( y \not\equiv z \), we need to show
  
  \[ (2) \quad x \not\equiv y \rightarrow z \implies x \not\equiv z , \]
  \[ (3) \quad x \not\equiv y \not\equiv z \implies x \not\equiv z , \]

- So in total we have to show (1), (2), (3) above.

Formally: We have \( x \rightarrow^* x \) and \( x \rightarrow^* x \), therefore \( x \not\equiv x \).
(2) $x \downarrow y \rightarrow z$ implies $x \downarrow z$

- Assume $x \downarrow y, y \rightarrow z$. Show $x \downarrow z$.
- $x \rightarrow^* u, y \rightarrow^* u$ for some $u$.
- By Church-Rosser, $y \rightarrow^* u$ and $y \rightarrow^* z$ implies that there exists a $v$ s.t. $u \rightarrow^* v, z \rightarrow^* v$.
- But then
  - $x \rightarrow^* u \rightarrow^* v$ therefore $x \rightarrow^* v$,
  - $z \rightarrow^* v$,
  - therefore $x \downarrow z$.

(3) $x \downarrow y \leftarrow z$ implies $x \downarrow z$

- Assume $x \downarrow y, y \leftarrow z$. Show $x \downarrow z$.
- $x \rightarrow^* u, y \rightarrow^* u$ for some $u$.
- But then
  - $x \rightarrow^* u$,
  - $z \rightarrow^* y \rightarrow^* u$,
  - therefore $x \downarrow z$.

This completes the first proof of the Theorem.
Second Proof of “⇒”

- We show that \(\downarrow\) contains \(\to\) and is reflexive, symmetric and transitive. Therefore it contains \(\leftrightarrow^*\), and we get
  \[ x \leftrightarrow^* y \Rightarrow x \downarrow y. \]

Formally: If \(x \to y\) then we have with \(z := y\) that
\[ x \to^* z \text{ and } y \to^* z. \]
Second Proof of "⇒"

- \( \downarrow \) is transitive:

Formally:
- Assume \( x \downarrow y \) and \( y \downarrow z \).
- Then there exists \( u, v \) s.t. \( x \rightarrow^* u \), \( y \rightarrow^* v \), \( z \rightarrow^* v \).
- Then by confluence there exists an \( w \) s.t. \( u \rightarrow^* w \), \( v \rightarrow^* w \).
- Then \( x \rightarrow^* w \) and \( z \rightarrow^* w \).
- Therefore \( x \downarrow z \).

Unique Normal Forms

Lemma:
Let \( (A, \rightarrow) \) be a confluent reduction system. If \( x \in A \) has a normal form \( y \), then it is unique:
- If \( z \) is another normal form, then \( y \equiv z \).

Proof:
- We have \( x \rightarrow^* y \) and \( x \rightarrow^* z \).
- By confluence, there exists a \( u \) s.t. \( y \rightarrow^* u \) and \( z \rightarrow^* u \).
- But since \( y \) and \( z \) are normal forms, it follows \( y \equiv u \) and \( z \equiv u \).

Second Proof of "⇒"

- \( \downarrow \) is transitive:

Formally:
- Assume \( x \downarrow y \) and \( y \downarrow z \).
  Then there exists \( u, v \) s.t. \( x \rightarrow^* u \), \( y \rightarrow^* v \), \( z \rightarrow^* v \).
- Then by confluence there exists an \( w \) s.t. \( u \rightarrow^* w \), \( v \rightarrow^* w \).
- Then \( x \rightarrow^* w \) and \( z \rightarrow^* w \).
- Therefore \( x \downarrow z \).

Picture

Since \( y, z \) are in normal form, \( y = u = z \)
Lemma

Let \((A, \rightarrow)\) be a weakly normalising and confluent reduction system. Then

\[ x \xrightarrow{\ast} y \text{ iff the normal forms of } x \text{ and } y \text{ coincide.} \]

Proof:

\[ \Rightarrow \]: By Church Rosser \( x \xrightarrow{\ast} y \) implies the existence of a \( z \) s.t. \( x \xrightarrow{\ast} z \) and \( y \xrightarrow{\ast} z \).

Reduce \( z \) further to a normal form \( u \).

Then \( u \) is a normal form of both \( x \) and \( y \) as well.

Since by the above lemma, normal forms are unique, \( u \) is the normal form of \( x \) and \( y \).

\[ \Leftarrow \]: If the normal forms \( z \) coincide, then we have \( x \xrightarrow{\ast} z \xrightarrow{\ast} y \), therefore \( x \xrightarrow{\ast} y \).

Remark on Agda

- The underlying reduction system of Agda is strongly normalising, provided the code has been termination checked.
- The equality derived from this reduction system is used in order to typecheck terms.

(d) Term Rewriting Systems

- Term rewriting systems are special cases of reduction systems.
- They are reduction systems, which are generated by a (in many cases finite) set of rules (i.e. basic reductions).

Example of a Term Rewriting System

- Take \( T = \text{set of arithmetic expressions formed from variables, } 0 \text{ by using the successor operation } S \text{ (where } S \text{ } n \text{ stands } n + 1), +, \ast \text{ and brackets.} \)
  - So the following are elements of \( T \):
    - \( x + S 0 \),
    - \( S 0 + z \ast (S (S x) + 0) \),
    - \( S y \ast S 0 + S x \ast 0 \).
  - Take as rules the following:
    - \( x + 0 \xrightarrow{\text{Rule}} x \),
    - \( x + S y \xrightarrow{\text{Rule}} S (x + y) \),
    - \( x \ast 0 \xrightarrow{\text{Rule}} 0 \),
    - \( x \ast S y \xrightarrow{\text{Rule}} x \ast y + x \).
Example Reductions

The system will be in fact strongly normalising and confluent.

Example of a Term Rewriting System

Term Rewriting Systems

The reduction relation generated by these rules allows to replace in a term
- any subterm of the form \( s + 0 \) by \( s \),
- any subterm of the form \( s + S t \) by \( S (s + t) \),
- any subterm of the form \( s * 0 \) by \( 0 \),
- any subterm of the form \( s * S t \) by \( s * t + s \).

So we have for instance the following reductions:
- \( 0 + S (S 0) \rightarrow S (0 + S 0) \),
- Reduce \( 0 + S s \) to \( S (0 + s) \) using \( s \equiv S 0 \).
- \( S (\emptyset + S \emptyset) \rightarrow S (S (\emptyset + \emptyset)) \),
- Reduce \( s + S t \) to \( S (s + t) \), using \( s \equiv t \equiv 0 \),
- \( S (S (\emptyset + \emptyset)) \rightarrow S (S \emptyset) \).
Definition of Term Rewriting Systems

- A term rewriting system consists of
  - a set of terms $T$ built from variables, constants and some function symbols,
  - a relation $\rightarrow_R$ between terms
    (if $r \rightarrow_R s$ we say that $r \rightarrow_R s$ is a rule),
  - s.t., if $s \rightarrow_R t$, then
    - $s$ is not a variable, and
    - all variables in $t$ occur in $s$.

Condition on Variables

- The second variable condition has something to do with determinism:
  - Assume we have chosen a rule $r \rightarrow s$ and chosen a substitution of variables in $r$, which matches a term $t$.
  - Then the reduct with respect to this rule is uniquely determined.
  - There are no other free variables in $s$ which allow additional choices for substitutions.

Condition on Variables

- In the previous definition we had two variable conditions for $s \rightarrow_R t$:
  - $s$ is not a variable.
    - If we allowed $s$ to be a variable say $x$, then the rule would have the form $x \rightarrow t$.
    - That would mean that any term $r$ has a reduction, namely to $t[x := r]$.
  - All variables in $t$ occur in $s$.
  - Assume $y$ were a variable in $t$ but not in $s$.
    - If we substitute in $s$ and $t$ all variables from $s$ by closed terms, and obtain $s'$ and $t'$ then we would have that $s'$ would have potentially infinitely many reductions, namely for any substitution of the other variables of $t'$ by closed terms.

Both these case would cause problems in the theory of term-rewriting systems (we won’t touch those problems).
Reduction generated by $\longrightarrow^\text{Rule}$

Don’t worry if you don’t understand the following definition.

For most of the module it suffices to have an intuitive understanding of how to reduce terms.

If we have a term rewriting system $(T, \longrightarrow^\text{Rule})$ we obtain a reduction relation $\longrightarrow$ on $T$ as follows:

First we construct a relation $\longrightarrow'$ obtained from reductions rules $r \longrightarrow^\text{Rule} r'$ by substituting the variables in both $r$ and $r'$ by some terms.

So the same substitutions are carried out in both $r$ and $r'$.

If $s \longrightarrow' s'$ is obtained by carrying out such a substitution in $r \longrightarrow^\text{Rule} r'$, then $s \longrightarrow' s'$ is called an instance of rule $r \longrightarrow^\text{Rule} r'$.

Example (Instance of a Rule)

\[
\begin{align*}
x + 0 & \longrightarrow^\text{Rule} x \\
x + S\,y & \longrightarrow^\text{Rule} S\,(x + y) \\
x \ast 0 & \longrightarrow^\text{Rule} 0 \\
x \ast S\,y & \longrightarrow^\text{Rule} x \ast y + x
\end{align*}
\]

$0 + 0 \longrightarrow' 0$ is an instance, obtained by substituting in $x + 0 \longrightarrow^\text{Rule} x$ the variable $x$ by 0.

$S\,0 \ast S\,0 \longrightarrow' S\,0 \ast 0 + S\,0$

is an instance, obtained by substituting in $x \ast S\,y \longrightarrow^\text{Rule} x \ast y + x$ the variable $x$ by $S\,0$ and the variable $y$ by 0.

Then $s \longrightarrow s'$, if there exists an instance $t \longrightarrow' t'$ of a rule s.t. $s$ contains subterm $t$, and $s'$ is the result of substituting in $s$ the term $t$ by $t'$.

The subterm $s$ is called a redex w.r.t. the term rewriting system used.

"Redex" is short for reducible expression.

Plural of redex is redexes.

The reductions $s \longrightarrow s'$ obtained this way are the reductions generated by the term rewriting system.
Example 1

\[
\begin{align*}
    x + 0 & \rightarrow_{\text{Rule}} x , \\
    x + S y & \rightarrow_{\text{Rule}} S (x + y) , \\
    x * 0 & \rightarrow_{\text{Rule}} 0 , \\
    x * S y & \rightarrow_{\text{Rule}} x * y + x .
\end{align*}
\]

0 + S (S 0) \rightarrow S (0 + S 0) is obtained as follows:

The rule used is

\[
    x + S y \rightarrow_{\text{Rule}} S (x + y) .
\]

By substituting \( x \) by 0 and \( y \) by S 0 we obtain the instance

\[
    0 + S (S 0) \rightarrow ' S (0 + S 0) .
\]

In this example, the redex is the full term 0 + S (S 0) which is then reduced.

Example 2

\[
\begin{align*}
    x + 0 & \rightarrow_{\text{Rule}} x , \\
    x + S y & \rightarrow_{\text{Rule}} S (x + y) , \\
    x * 0 & \rightarrow_{\text{Rule}} 0 , \\
    x * S y & \rightarrow_{\text{Rule}} x * y + x .
\end{align*}
\]

S (S (0 + 0)) \rightarrow S (S 0).

The rule used is

\[
    x + 0 \rightarrow_{\text{Rule}} x .
\]

By substituting 0 for \( x \), we obtain the instance

\[
    0 + 0 \rightarrow ' 0 .
\]

The left hand side of the reduction S (S (0 + 0)) contains the redex 0 + 0.

By substituting it by 0 we obtain the right hand side of the reduction S (S 0).

Example 3

\[
\begin{align*}
    x + 0 & \rightarrow_{\text{Rule}} x , \\
    x + S y & \rightarrow_{\text{Rule}} S (x + y) , \\
    x * 0 & \rightarrow_{\text{Rule}} 0 , \\
    x * S y & \rightarrow_{\text{Rule}} x * y + x .
\end{align*}
\]

\[
\begin{align*}
    S (S (0 + 0)) & \rightarrow S (S 0) .
\end{align*}
\]

The rule used is

\[
    x + 0 \rightarrow_{\text{Rule}} x .
\]

By substituting 0 for \( x \), we obtain the instance

\[
    0 + 0 \rightarrow ' 0 .
\]

The left hand side of our reduction S (S (0 + 0)) contains now the redex 0 + S 0.

By substituting it by S (0 + 0) we obtain the right hand side of the reduction, S (S (0 + 0)).

(e) The Untyped \( \lambda \)-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.
Example for Use of $\lambda$

- Can be done by multiplying each student number by 10.
- Let $f : \mathbb{N} \to \mathbb{N}, f(x) := x \times 10$.
- 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
  \[
  \text{map}(f, [210345, 345698, 296458]) \\
  = [2103450, 3456980, 2964580].
  \]

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

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. $f(x) = x \times 10$”.
- Notation is $\lambda x. x \times 10$.
- So we have
  \[
  \text{map}(\lambda x. x \times 10, [210345, 345698, 296458]) \\
  = [2103450, 3456980, 2964580].
  \]
- In general $\lambda x. t$ stands for the function $f$ s.t. $f(x) = t$, where $t$ might depend on $x$.
- above $t = x \times 10$.

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 $S x + S y$ stands for $(S x) + (S y)$.
- * binds more than +.
  - Therefore $x + y * z$ stands for $x + (y * z)$,
  - and $S x + S y * z$ stands for $(S x) + ((S y) * z)$.
- These are the same conventions as in Agda.

**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.

**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 $\lambda x. ((\lambda y. y) 5)$.

**$\lambda$ without a Dot**

- 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 \((r\ s)\ t\).

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 \(+,\ *\), one has
  - constants \((+),\ (*),\)
  - infix operations \(+,\ *\),
  - and writes in infix notation
    - \(x + y\) instead of \((+x)\ y\),
    - \(x * y\) instead of \((*x)\ y\),
    - etc.

We have now bound and free variables:
- Free variables are variables \(x\), which don’t occur in the scope of a \(\lambda\)-abstraction “\(\lambda x.\)”.
- Bound variables are the other variables:
  - they are variables \(x\) that occur in the scope of a \(\lambda\)-abstraction “\(\lambda x.\)”.
  - 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 \((\lambda y \cdot y + z) \cdot y\),
  - the first occurrence of \(y\), \(y\) is bound,
  - the second occurrence of \(y\), \(y\) is free,
  - \(z\) is free.

- In \((\lambda y \cdot ((\lambda z \cdot y)) \cdot x)\), 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.

By the free and bound variables of a term \(t\) we mean the variables \(x\) which occur free or bound, respectively, in \(t\).

\(\alpha\)-Conversion

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

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

- So \(\lambda x \cdot \lambda y \cdot y\) and \(\lambda y \cdot \lambda x \cdot x\) are \(\alpha\)-equal, written as \(\lambda x \cdot \lambda y \cdot y \equiv_{\alpha} \lambda y \cdot \lambda x \cdot x\).

Note that \(\lambda x \cdot \lambda x \cdot x =_{\alpha} \lambda y \cdot \lambda x \cdot x\).

The \(x\) refers to the second lambda abstraction \(\lambda x\), not the first one \((\lambda x\cdot)\).

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 $\lambda$-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 $\lambda y.y$.

Substitution

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.
Substitution and Parentheses

E.g.

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

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

\[2 + 2 \ast 2 + 2\]

which is different from

\[(2 + 2) \ast (2 + 2)\].

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.

Substitution and \(\alpha\)-Conversion

When substituting in \(\lambda\)-terms, we sometimes have to carry out an \(\alpha\)-conversion first:

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

Substitution and $\alpha$-Conversion

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.

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 $\lambda y$.

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.

$\beta$-Redexes

- A $\lambda$-term might have several $\beta$-redexes:
  - E.g. In $(\lambda x.x) ((\lambda y.y) z)$ we have
    - one redex $(\lambda x.x) ((\lambda y.y) z)$
    - and one redex $(\lambda y.y) z$.

$\beta$-Reduct

- 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 $(\lambda x.r)$ of a $\lambda$-term $t$ is called a $\beta$-redex of $t$.
  - “Redex” is short for reducible expression.
  - Plural of redex is redexes.
- Examples:
  - $(\lambda x.x) y z$ has $\beta$-redex $(\lambda x.x) y$.
  - A redex can be the term itself: $(\lambda x.x) y$ has $\beta$-redex $(\lambda x.x) y$.
**β-Reduction**

\[ r \longrightarrow_\beta r', \] or shorter \( r \rightarrow_\beta r' \), if \( r' \) is obtained from \( r \) by replacing a β-redex by its β-reduct.

**Examples:**

1. \((\lambda x.x + 5) 3 + 7 \longrightarrow (3 + 5) + 7\), since \((\lambda x.x + 5) 3 \longrightarrow 3 + 5\).

2. Assume we add a pairing operation \((s, t)\) for the pair \(s, t\) (will be introduced later), then \(\langle (\lambda x.x + 5) 3, 7 \rangle \longrightarrow \langle 3 + 5, 7 \rangle\).

3. We can apply β-reduction under a λ term as well: \(\lambda x.((\lambda y.y + 5) 3) \longrightarrow \lambda x.3 + 5\).

**Multiple Redexes**

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

**Examples of β-Reduction**

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

**Examples (Longer Reduction)**

\[
\begin{align*}
(\lambda x. y.x (x y)) (\lambda u, v.u (u v)) & \longrightarrow \lambda y.(\lambda u, v.u (u v)) ((\lambda u, v.u (u v)) y) \\
& \longrightarrow \lambda y.((\lambda u, v.u (u v)) (\lambda v.y (y v))) \\
& \longrightarrow \lambda y.\lambda v.((\lambda v.y (y v)) ((\lambda v.y (y v)) v)) \\
& \longrightarrow \lambda y.((\lambda v.y (y v)) (y (y v))) \\
& \equiv \lambda y, v.y (y (y v)))
\end{align*}
\]
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 $\widetilde{\Omega} := \lambda x. (x x)$.
  Then
  $$\widetilde{\Omega} \equiv (\lambda x. (x x)) \widetilde{\Omega} \rightarrow \widetilde{\Omega} (\widetilde{\Omega} \widetilde{\Omega}) \rightarrow \widetilde{\Omega} (\widetilde{\Omega} (\widetilde{\Omega} \widetilde{\Omega})) \rightarrow \ldots$$

---

$\lambda$-Calc. as a Red. Sys

- 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**.

---

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.
  $$r \equiv s_0 =_\alpha t_0 \leftrightarrow_\beta s_1 =_\alpha t_1 \leftrightarrow_\beta s_2 =_\alpha t_2 \leftrightarrow_\beta \ldots$$

- **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.
\(\beta\)-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. \(s \rightarrow^* t\) and \(r \rightarrow^* t\).
- But simply reducing \(r\) might never terminate.

**Example:**
- \((\lambda x.y) \Omega\) reduces in one step to \(y\).
- So \((\lambda x.y) \Omega =_\beta y\).
- 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 \(y\).

---

### Need for Typed \(\lambda\)-Calculus

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.

---

### (f) The Typed \(\lambda\)-Calculus

**Problem of the untyped \(\lambda\)-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 \(\lambda\)-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 \(\text{int} \rightarrow \text{int}\)”.
- 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.

Simple Types

The simple types used in the simply typed $\lambda$-calculus are defined inductively as follows:
- The ground type $\text{o}$ is a type.
- If $\sigma, \tau$ 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 $\lambda$-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. $\alpha$ (“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:
  - $\text{o}$,
  - $\text{o} \to \text{o}$,
  - $(\text{o} \to \text{o}) \to \text{o}$,
  - $((\text{o} \to \text{o}) \to (\text{o} \to \text{o})) \to (\text{o} \to \text{o}) \to \text{o} \to \text{o}$,
  - which stands for
    $(((\text{o} \to \text{o}) \to (\text{o} \to \text{o})) \to (\text{o} \to \text{o}) \to (\text{o} \to \text{o})).$

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).
  - $\text{o}2 := \text{o} \to \text{o}$,
  - $\text{o}3 := \text{o}2 \to \text{o}2$,
  - etc.
- So
  - an element of type $\text{o}2$ can be applied to an element of type $\text{o}$ and one obtains an element of type $\text{o}$.
  - an element of type $\text{o}3$ can be applied to an element of type $\text{o}2$ and one obtains an element of type $\text{o}2$.
  - 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 λ-term \( x \, y \), we could have

- \( x : o_2, \, y : o \) and therefore \( x \, y : o \),
- or \( x : o_2, \, y : o_3 \), 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,
- \( \sigma_i \) will be types of that theory.
- \( x_i \) are different.
- \( n = 0 \) is allowed, and we write \( \emptyset \) for the empty context.

Examples

- \( x : o, \, y : o_2 \) is a context.
- \( x : o_2, \, x : o \) is not a context.

Note that contexts are lists of elements of the form \( \sigma \), 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 \( x : \sigma \).

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 \( \Gamma : s : \sigma \) for “in context \( \Gamma \), \( s \) has type \( \sigma \)”.

Expressions of this form are called judgements.

Examples:

- \( x : o_2, \, y : o \Rightarrow x \, y : o \),
- \( x : \text{float} \rightarrow \text{int}, \, y : \text{float} \Rightarrow x \, y : \text{int} \) (assuming ground types \( \text{float and int} \)),
- \( x : o_3, \, y : o_2 \Rightarrow x \, y : o_2 \).

In case \( \Gamma \) is empty, we write \( s : \sigma \) instead of \( \emptyset : s : \sigma \).
**Contexts**

- If $\Gamma$, $\Delta$ are contexts, $\Gamma, \Delta$ denotes the concatenation of both contexts, e.g. if
  - $\Gamma \equiv x : o, y : o^2$,
  - $\Delta \equiv z : o$
then
  - $\Gamma, \Delta$ denotes $x : o, y : o^2, z : o$,
  - $\Delta, \Gamma$ denotes $z : o, x : o, y : o^2$,
  - $\Gamma, u : o$ denotes $x : o, y : o^2, u : o$.

**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:

   $$\Gamma, x : \sigma, \Delta \Rightarrow x : \sigma$$

   - Note that $\Gamma, x : \sigma, \Delta$ stands for any context, in which $x : \sigma$ occurs.
   - **Explanation:** From the assumption $x : \sigma$ we can derive $x : \sigma$.

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 $\Gamma$:

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

   - **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 (Assumption)**

- We will illustrate the rules using a derivation of
  $$y : o \rightarrow o \rightarrow o \Rightarrow \lambda x^o.y \ x : o \rightarrow o$$

- In order to derive it we will need to derive first
  $$y : o \rightarrow o \rightarrow o, x : o \Rightarrow y \ x$$

- In order to derive that we use twice the assumption rule and obtain
  $$y : o \rightarrow o \rightarrow o, x : o \Rightarrow y : o \rightarrow o \rightarrow o$$

  and
  $$y : o \rightarrow o \rightarrow o, x : o \Rightarrow x : o$$
Example (Application)

- We continue with our derivation of
  \[ y : o \rightarrow o \rightarrow o \Rightarrow \lambda x^o. y \ x : o \rightarrow o \]

- 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 (note that \( o \rightarrow o \rightarrow o \equiv o \rightarrow (o \rightarrow o) \)):
  \[
  \frac{\quad y : o \rightarrow o \rightarrow o, x : o \Rightarrow y : o \rightarrow o \rightarrow o \quad \quad y : o \rightarrow o \rightarrow o, x : o \Rightarrow x : o \quad}{y : o \rightarrow o \rightarrow o, x : o \Rightarrow y \ x : o \rightarrow o} \ (Ap)
  \]

Example (Abstraction)

- We finish our derivation of
  \[ y : o \rightarrow o \rightarrow o \Rightarrow \lambda x^o. y \ x : o \rightarrow o \]

- We have already derived
  \[
  \frac{\quad y : o \rightarrow o \rightarrow o, x : o \Rightarrow y : o \rightarrow o \rightarrow o \quad \quad y : o \rightarrow o \rightarrow o, x : o \Rightarrow x : o \quad}{y : o \rightarrow o \rightarrow o, x : o \Rightarrow y \ x : o \rightarrow o} \ (Ap)
  \]

- Using abstraction we obtain: (note that \( o \rightarrow o \rightarrow o \equiv o \rightarrow (o \rightarrow o) \)):
  \[
  \frac{\quad y : o \rightarrow o \rightarrow o, x : o \Rightarrow y : o \rightarrow o \rightarrow o \quad \quad y : o \rightarrow o \rightarrow o, x : o \Rightarrow x : o \quad}{y : o \rightarrow o \rightarrow o \Rightarrow \lambda x^o. y \ x : o \rightarrow o \rightarrow o} \ (Abs)
  \]

Simply Typed \(\lambda\)-Calculus

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

   **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 \( \sigma \rightarrow \tau \).

Rules

- We had three rules:
  1. \( \Gamma, x : \sigma, \Delta \Rightarrow x : \sigma \)
  2. \[
  \frac{\Gamma \Rightarrow s : \sigma \rightarrow \tau}{\Gamma \Rightarrow s \ t : \tau} \ (Ap)
  \]
  3. \[
  \frac{\Gamma, x : \sigma \Rightarrow t : \tau}{\Gamma \Rightarrow \lambda x^\sigma. t : \sigma \rightarrow \tau} \ (Abs)
  \]
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:

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

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:

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

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

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

\[
\frac{x : o2, y : o \Rightarrow x y : o \quad \lambda y^o.x y : o2}{\lambda x^o2.\lambda y^o.x y : o3} \quad \text{(Abs)}
\]

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

\[
(\lambda x^o2.\lambda y^o.x y) : o2 \rightarrow o2 = o3
\]

\[
\frac{x \rightarrow o2 \quad x \rightarrow y \quad y \rightarrow o3}{x \rightarrow o \rightarrow o = o2}
\]
(g) The $\lambda$-Calculus in Agda

- Agda is based on dependent type theory.
- This extends the simply typed $\lambda$-calculus.

The Function Type in Agda

- In Agda one writes $A \to C$ for the nondependent function type.
  We write on our slides $\to$ instead of $\rightarrow$.

Installation of Agda

- Instructions on how to install Agda can be found under http://www-compsci.swan.ac.uk/~csetzer/othersoftware/agda/agdainstallation.html
- The installation will provide an Emacs/XEmacs mode for agda files.
- If a file with extension .agda is loaded into Emacs/XEmacs, then this mode is invoked.

$\lambda$-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 $\rightarrow$ in real Agda code.
- When reasoning in type theory itself (outside Agda), we use standard type theoretic notation $\lambda (x : A). r$.
  - We sometimes omit $A$, and write simply $\lambda x . r$. 
**Notations in Agda**

- As an abbreviation, one writes
  \[ \lambda(a, a' :: A) \to \cdots \]
  instead of
  \[ \lambda(a :: A) \to \lambda(a' :: A) \to \cdots \]

**Application in Agda**

- **Application** has the same syntax as in the rules of dependent type theory: If we have
  \[ f :: A \to B , \]
  \[ a :: A , \]
  then we can conclude \[ f a :: B. \]
- And we have that
  \[ (\lambda(x :: A) \to b) a \]
  and
  \[ b[x := a] \]
  are identified.

**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:
  \[
  \text{postulate } A :: \text{Set}
  \]

**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:

- postulate \(A :: \text{Set}\)
- postulate \(B :: \text{Set}\).
- postulate \(f :: A \to B\).
- postulate \(a :: A\).

See `examplePostulate1.agda`
Basic $\lambda$-Terms

postulate $A :: Set$
postulate $B :: Set.$
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:
\[
\begin{align*}
b & :: B \\
& = f \ a
\end{align*}
\]

We can as well introduce $g := \lambda x : A. x :: A \rightarrow A$ as follows:
\[
\begin{align*}
g & :: A \rightarrow A \\
& = \lambda(x :: A) \rightarrow x
\end{align*}
\]

See examplePostulate2.agda

$\lambda$-Terms

postulate $A :: Set$
postulate $B :: Set.$
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:
\[
g \ (a :: A) \\
:: A \\
= a
\]

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:
\[
\begin{align*}
\text{postulate } A :: \\
\text{postulate } g :: A \rightarrow A \\
& = \lambda(a' :: A) \rightarrow a \\
g' & (a :: A) \\
& :: A \\
& = a
\end{align*}
\]

Example
exampleEquivalenceLambdaNotations2.agda
Equivalence of the two Notations

If one then introduces an arbitrary goal, e.g.

\[
\text{Test :: } A \rightarrow A \\
= \{! !\}
\]

inserts into it the value \( g \), uses goal-menu **Compute WHNF**, (for compute weak head normal form), and does the same with \( g' \), one obtains twice the same result, namely

\[
\lambda(a :: A) \rightarrow a
\]

The above is a general method for evaluating terms (e.g. for evaluating the result of a applying a function to some arguments).

\[\lambda\text{-Notation in Agda}\]

**Abbreviation**

We can write

\[
h \ (a,a'::A) \\
:: B \\
= \ldots
\]

instead of

\[
h \ (a::A) \\
(a'::A) \\
:: B \\
= \ldots
\]

\[\lambda\text{-Notation in Agda}\]

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:

\[
\text{map } (\lambda(x::N) \rightarrow S \ x) \\
(\text{cons two } (\text{cons three } \text{nil}))
\]

The result would be

\[
\text{cons three } (\text{cons four } \text{nil})
\]

See exampleMapAppliedToList.agda.

\[\text{More on Goals}\]

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

As explained in Subsection a, parts of the code can be made goals and left open.

**Example:**

\[
g \ (x :: A) \\
:: A \\
= \{! !\}
\]

exampleGoal1.agda
More on Goals

- Using the goal-menu, we can find out what type is expected:
  - **Type of Goal C-c C-t.**
    Agda shows: \( ?0 :: A \)
  - There is as well a variant **Type of Goal (unfolded) C-c C-x C-r.**
    It evaluates the type of the goal using reduction rules (see later).
  - Using the main menu we can always show the types of all goals (**show-goals**).
  - We can as well jump to the next and previous goal using menu items **Next Goal** and **Previous Goal**.

Intro

- When introducing \( \lambda \)-terms, there is no need to introduce the \( \lambda \)-term by hand.
- Assume for instance the following goal:

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

- If the position is inside the goal, we can use the goal-menu **Intro**, in order to obtain a template for the \( \lambda \)-term:

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

Solve

- We usually don’t have to introduce the types of the \( \lambda \)-variables, they can usually be solved automatically using Agda-menu **Solve**.
- Applied in the above situation we obtain:

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

- It is recommended to rename the automatically generated variable \( h \) to something more meaningful:

  \[
  f :: A \to A
  = \lambda(a :: A) \to \{ ! ! \}
  \]
In some cases, the types inferred can be simplified. For instance if we start with

\[ g :: ((A \to A) \to A) \to A \]
\[ = \lambda(h :: \{! !\}) \to \{! !\} \]

we obtain using solve

\[ g :: ((A \to A) \to A) \to A \]
\[ = \lambda(h :: (h :: (h :: A) \to A) \to A) \to \{! !\} \]

which can be simplified to

\[ g :: ((A \to A) \to A) \to A \]
\[ = \lambda(a \_a \_a :: (A \to A) \to A) \to \{! !\} \]

In general, solve will solve goals, for which there is only one solution in the current context.

- Uniqueness is up to equality in Agda, which is equality of the normal forms.
- Therefore one can often replace the solution given by Agda by some more readable term, but it will always be – up to equality in Agda – the same term.

Assume the following Agda code

\[
\begin{align*}
\text{postulate } A :: & \text{ Set} \\
\text{postulate } B :: & \text{ Set} \\
\text{postulate } f :: & A \to B \\
\text{postulate } a :: & A \\
b :: & B \\
= & \{! !\}
\end{align*}
\]

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

postulate A :: Set
postulate B :: Set
postulate f :: A → B
postulate a :: A
b :: B
   = {! !}

Then we can insert f into this goal and use menu
Refine (C-c C-r)
The system shows f {! !} :: B.
We can ask for the type of the new goal {! !}, and get:
{! !} :: A
Now we can solve this goal by filling in a and using
refine: f a :: B.

Introducing New Types

In the λ-calculus, we introduced abbreviations for types,
like o2 = o → o
We can do the same in Agda:

postulate A :: Set
A2 :: Set
   = A → A
A3 :: Set
   = A2 → A2
a2 :: A2
   = λ(x :: A) → x
a3 :: A3
   = λ(x :: A2) → x
Derivations in Agda

- In Agda, rules are implicit.
- The rule
  \[
  \frac{f : A \rightarrow B \quad a : A}{f \ a : B} \quad (Ap)
  \]
  corresponds to the following:
- Assume we have introduced:
  \[ g :: D \rightarrow E, \ d :: D. \]
and want to solve the goal
  \[ \{! \! \} :: E. \]

`exampleSimpleDerivation4.agda`

Derivations in Agda (Cont.)

- Then we can fill this goal by typing in \( g \ d \):
  \[ \{! \ g \ d \! \} :: E \]
- If we then choose goal-menu `Refine (C-c C-r)`, the system shows:
  \[ g \ d :: E. \]

\((x :: A) \rightarrow \cdots \) vs. \(\lambda(x :: A) \rightarrow \cdots\)

- Sometimes users of Agda (including the lecturer himself) confuse \((x :: A) \rightarrow \cdots\) and \(\lambda(x :: A) \rightarrow \cdots\).
- Happens probably because of the similarity of both notions.
  \((x :: A) \rightarrow B\)
  - is a set (or type).
  - the set/type of functions, mapping \(x :: A\) to an element of type \(B\).
  - Therefore it makes sense to talk about \(s :: ((x :: A) \rightarrow B)\).

\((x :: A) \rightarrow \cdots\) vs. \(\lambda(x :: A) \rightarrow \cdots\)

- \(\lambda(x :: A) \rightarrow t\)
  - is a term.
  - the function, mapping an element \(x :: A\) to the element \(t\).
  - It does not make sense to say \(s\) is an element of a function.
  - Correspondingly it does not make sense to talk about \(s :: (\lambda(x :: A) \rightarrow t)\).
  - \((\lambda(x :: A) \rightarrow t)\) never occurs in a position where a set/type is required.
  - It therefore never occurs on the right hand side of `::`.
- It does however make sense to talk about \((\lambda(x : A) \rightarrow t) :: B\) for some set (or type) \(B\).
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

\[
\text{let } a_1 :: A_1 \\
\quad = s_1 \\
\quad a_2 :: A_2 \\
\quad = s_2 \\
\quad \ldots \\
\quad a_n :: A_n \\
\quad = s_n \\
\text{in } t
\]

Let expressions in Agda (Cont.)

- This means that we introduce new local constants
  \[ a_1 :: A_1 = s_1, \]
  \[ a_2 :: A_2 = s_2, \]
  \[ \ldots, \]
  \[ a_n :: A_n = s_n, \]
  which can now be used locally.
- \( s_i \) can refer to all \( a_j \) defined before, including \( a_i \) itself, i.e. it can refer to \( a_0, \ldots, a_i \).
- Reference to \( a_i \) might result in non-termination; termination will be discussed below.

Simple Example

The following function computes \((n + n) * (n + n)\) for \( n :: N \):

\[
f \ (n :: N) \\
\quad :: N \\
\quad = \text{let } m :: N \\
\quad \quad = n + n \\
\quad \quad \text{in } m * m
\]

See exampleLetExpression.agda

Note that this version costs computationally less than the function computing directly \((n + n) * (n + n)\):

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

As an example we treat we define a function

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

using the postulates \( A : \text{Set} \).

We start with the goal

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

\[ = \{! !\} \]

We use now \textit{intro}. After \textit{renaming of the variable} we obtain:

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

\[ = \lambda(\text{aaa} :: (A \rightarrow A) \rightarrow A) \]

\[ \rightarrow \{! !\} \]

Example

We can use \textit{aaa} in order to obtain a provided we have defined some function \( \text{aa} : A \rightarrow A \).

Therefore we first define in an auxiliary definition \( \text{aa} : A \rightarrow A \).

In this example we could do this as a global definition, but will use here a let expression instead.

We type into the open goal the variables (separated by blanks) which we want to introduce during this let expression.

In this example is only one variable \( \text{aa} \), which we type into the goal. Then using goal menu "\textit{make let expression}" we obtain (see next slide)

Example

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

\[ = \lambda(\text{aaa} :: (A \rightarrow A) \rightarrow A) \rightarrow \{! !\} \]

\[ \text{let } \text{aa} :: \{! !\} \rightarrow \{! !\} \]

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

Example

We type into the first goal the type \( A \rightarrow A \) of the variable \( \text{aa} \) and use goal menu \textit{Refine} or \textit{Give} and obtain

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

\[ = \lambda(\text{aaa} :: (A \rightarrow A) \rightarrow A) \rightarrow \{! !\} \]

\[ \text{let } \text{aa} :: A \rightarrow A \rightarrow \{! !\} \]

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

Example

We apply \textit{Intro} to the first goal and obtain

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

\[ = \lambda(\text{aaa} :: (A \rightarrow A) \rightarrow A) \rightarrow \{! !\} \]

\[ \text{let } \text{aa} :: A \rightarrow A \rightarrow \{! !\} \rightarrow \{! !\} \]

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

Using **Solve** we get the type of the first goal \( A \) for free:

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

Example

We solve the goal by typing in \( a \) and using **Refine** and are finished with the let-expression:

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

Example

We rename the variable \( h \) and obtain:

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

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

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

See **letexpressionSubSect2j.agda**
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

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

Postulates as Formulae

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 $((A \land B) \lor C) \to A$).

Formulae like $((A \land B) \lor C) \to A$ might be generally true (like $A \to A$), or might be true depending on the truth of its propositional formulae (like $A \lor C$).

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.

Constructive Logic

If $p : \text{Sorted}(l)$ holds, then $l$ should be sorted.
If we can prove that there exist no $p : \text{Sorted}(l)$, then $l$ should be not sorted.
If we know neither that $p : \text{Sorted}(l)$ nor that for no $p$ it follows $p : \text{Sorted}(l)$, then we know neither that $l$ is sorted nor that $l$ is not sorted.
Happens e.g. if $l$ is a variable.
For 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 that no $p : A$ exists.

Proofs in dependent type theory will have always a constructive meaning.
In case of implication the constructive meaning of a proof of $b : A \to B$ will be:
It is a function, which from a proof of $A$ determines a proof of $B$.
This is what is meant by $A \to B$: if $A$ holds, i.e. if we have a proof of $A$, then $B$ holds, i.e. we have a proof of $B$.
So $b : A \to B$ is a function mapping proofs of $A$ to proofs of $B$.
This is nothing but the function type $A \to B$.
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)

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

In ordinary logic, the \( \lambda \)-type just introduced corresponds to the following derivation of \( (A \rightarrow B) \rightarrow A \rightarrow B \):
- Assume \( A \rightarrow B \).
- Assume \( A \).
- Then from \( A \rightarrow B \) and \( A \) we obtain \( B \).
- This shows \( (A \rightarrow B) \rightarrow A \rightarrow B \) 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 \( B \).

Implication in Agda

Therefore we can represent implication by \( \rightarrow \) in Agda.

Elements of formula constructed from \( \rightarrow \) will be proofs that the formula holds.
Example (Implication in Agda)

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

\[
\text{Lemma1 :: Set} \\
= A \rightarrow (A \rightarrow B) \rightarrow B
\]

In order to prove Lemma1 we make the following goal:

\[
\text{lemma1 :: Lemma1} \\
= \{! !\}
\]

The Logical Connectives in Agda

One obtains:

\[
\text{lemma1 :: Lemma1} \\
= \lambda(h :: A) \rightarrow \lambda(h1 :: A \rightarrow B) \rightarrow \{! !\}
\]

After renaming of the variables one obtains:

\[
\text{lemma1 :: Lemma1} \\
= \lambda(a :: A) \rightarrow \lambda(ab :: A \rightarrow B) \rightarrow \{! !\}
\]

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 $B$.

The Logical Connectives in Agda

Lemma1 :: Set $= A \rightarrow (A \rightarrow B) \rightarrow B$

\[
\text{lemma1 :: Lemma1} \\
= \{! !\}
\]

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.

If one uses goal-menu Intro followed by Solve, in order to fill in the types of the $\lambda$-variables, this is done automatically.

Type of goal is $B$

At the position of the goal we have context $a :: A$ and $ab :: A \rightarrow B$, because we have $\lambda$-abstracted those variables.

Can be checked by using goal-menu context.

We can take $ab :: A \rightarrow B$ and apply it to $a :: A$ in order to obtain $ab a :: B$, which solves the goal.
The Logical Connectives in Agda

We obtain the following proof:

\[ \text{lemma1 :: Lemma1} = \lambda(a :: A) \to \lambda(ab :: A \to B) \to ab a \]

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

See exampleproofprologic1.agda

Implication in Agda

As for \( \lambda \)-terms, the following is equivalent to the definition of lemma1 above, and therefore an equivalent proof of \( A \to (A \to B) \to B \):

\[ \text{lemma1 (a :: A) (ab :: A \to B) :: B} = ab a \]

See exampleproofprologic2.agda

Recursive Definitions

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

\[ a :: A = a \]

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

\[ (+) (n, m :: N) :: N = \text{case m of} \]
\[ Z \to n \]
\[ S m' \to S (n + m') \]
Recursive Definitions and Proofs

- Recursive definitions spoil the principle of propositions as types:
  \[ a :: A = a \]
  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
    \[ a \rightarrow a \rightarrow a \rightarrow a \rightarrow \cdots \]

- We have only a constructive proof \( p \) of \( A \) if \( p \) can be reduced to a normal form which is a constructive witness of \( A \).

Need for Termination Checker

- 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 tied 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: Menu Check Termination.
  - 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.

- One should use the command Check Termination at the end of a session, in order to avoid black hole recursion and non-valid proofs.
Examples

- \( a :: A = a \) will not pass the termination checker.
- \( f :: A \rightarrow A = \lambda(a :: A) \rightarrow a \) will pass the termination checker.

Termination Check

- 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.

(h) More on the Typed \( \lambda \)-Calculus

\( \beta \)-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
  \[
  (\lambda x^\sigma.t) \; s \rightarrow t[x := s]
  \]
- 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

\[
(\lambda x^o^3.\lambda y^o^2.x \; (x \; y)) \; (\lambda x^o^2.\lambda y^o.x \; (x \; y)) \\
\rightarrow \lambda y^o^2.(\lambda x^o^2.\lambda y^o.x \; (x \; y)) \; ((\lambda x^o^2.\lambda z^o.x \; (x \; z)) \; y) \\
≡\alpha \lambda y^o^2.(\lambda x^o^2.\lambda y^o.x \; (x \; y)) \; ((\lambda x^o^2.\lambda z^o.x \; (x \; z)) \; y) \\
\rightarrow \lambda y^o^2.(\lambda x^o^2.\lambda y^o.x \; (x \; y)) \; (\lambda z^o.y \; (y \; z)) \\
≡\alpha \lambda y^o^2.(\lambda x^o^2.\lambda u^o.x \; (x \; u)) \; (\lambda z^o.y \; (y \; z)) \\
\rightarrow \lambda y^o^2.\lambda u^o.(\lambda z^o.y \; (y \; z)) \; ((\lambda z^o.y \; (y \; z)) \; u) \\
≡\lambda y^o^2.\lambda u^o.(\lambda z^o.y \; (y \; z)) \; ((\lambda z^o.y \; (y \; z)) \; u) \\
\rightarrow \lambda y^o^2.\lambda u^o.(\lambda z^o.y \; (y \; z)) \; (y \; u) \\
≡\lambda y^o^2.\lambda u^o.(\lambda z^o.y \; (y \; z)) \; (y \; u) \\
\rightarrow \lambda y^o^2.\lambda u^o.y \; (y \; (y \; u))
\]
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).

$\eta$-Rule

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

$\eta$-Rule

- If we have a function $f : \sigma \rightarrow \tau$, then this function applied to $x : \sigma$ gives result $f \ x$.
- 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 $f =_{\beta} \lambda x.f \ x$.
- 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.
- Therefore we introduce a rule, which expresses that $f$ is always equal to $\lambda x.f \ x$ w.r.t. $\beta, \eta$-reduction (where $x$ is fresh).

$\eta$-Rule

- However, we need to impose some restrictions, in order to avoid circularities:
  - If $t$ is of the form $\lambda y.s$ and if we then allowed to expand $t$, we would obtain the following circularity:
    $t \rightarrow \lambda x.t \ x \equiv \lambda x.(\lambda y.s) \ x \rightarrow_{\beta} \lambda x.s[y := x] =_{\alpha} t$,
  - 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

- \(\rho \eta\ s\) for \(s\) is obtained from \(r\) by the \(\eta\)-rule,
- \(\rho \beta \eta\ s\) for \(s\) is obtained from \(r\) by using \(\beta\)-reduction or \(\eta\)-expansion.

Notions like \(\rho \beta \eta\ b\), \(\equiv \beta \eta\), \(\equiv \eta\), \(\beta, \eta\)-normal form, etc. are to be understood correspondingly.

Example

\((\lambda f^{o3}.\lambda x^{o2}.f\ x)\ f\)

\[\vdash \beta \lambda x^{o2}.f\ x\]

\[\vdash \eta \lambda x^{o2}.\lambda y^0.f\ x\ y\]

\[\vdash \eta \lambda x^{o2}.\lambda y^0.f\ (\lambda z^o.x\ z)\ y\]

\(\lambda x^{o2}.\lambda y^0.f\ (\lambda z^o.x\ z)\ y\) is therefore the \(\beta, \eta\)-normal form of \((\lambda f^{o3}.\lambda x^{o2}.f\ x)\ f\).

Since \(f \vdash \eta \lambda x^{o2}.f\ x\), this 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

\((\lambda f^{o4}.\lambda x^{o3}.f\ x)\ f\)

\[\vdash \beta \lambda x^{o3}.f\ x\]

\[\vdash \eta \lambda x^{o3}.\lambda y^{o2}.f\ x\ y\]

\[\vdash \eta \lambda x^{o3}.\lambda y^{o2}.\lambda z^o.f\ (\lambda u^{o2}.x\ u)\ y\ z\]

\[\vdash \eta \lambda x^{o3}.\lambda y^{o2}.\lambda z^o.f\ (\lambda u^{o2}.\lambda v^{o2}.x\ (\lambda u^0.u\ w)\ v)\ y\ z\]

\[\vdash \eta \lambda x^{o3}.\lambda y^{o2}.\lambda z^o.f\ (\lambda u^{o2}.\lambda v^{o2}.x\ (\lambda u^0.u\ w)\ v)\ (\lambda u^0.y\ u)\ z\]

which is as well the \(\beta, \eta\)-normal form of \(f : o4\).

Theorem

The typed \(\lambda\)-calculus with \(\beta\)-reduction and \(\eta\)-expansion is confluent and strongly normalising.
**η-Rule**

With the η-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
\[
\[ r =^\beta \lambda x^\sigma.r \ x : \tau \ x : \tau \rightarrow \quad \lambda x^\sigma.(\lambda u^\sigma.t) \ x \rightarrow^\beta \lambda x^\sigma.t[u := x] \\
=^\alpha \lambda u^\sigma.t \\
\equiv r \]

Otherwise \( r \rightarrow^\eta \lambda x^\sigma.r \ x \).

Therefore, every function is of the form \( \lambda x.\text{something} \).

One can say the η rule expresses: every function is of the form \( \lambda x.\text{something} \).

---

**η-Rule in Agda**

In Agda syntax, the η-rule would state that if
\[
f :: A \rightarrow B
\]
then
\[
f = \lambda(x :: A) \rightarrow f \ x .
\]

- η-rule is computationally expensive and therefore not implemented.
- The lack of the η-rule causes sometimes problems.

---

**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.

---

**C[x], C[t]**

When explaining the notion of the missing η-rule we will use in the following notations \( C[x], C[t], s[x], s[t] \).

- \( C[x] \) stands for a type (or term) \( C \) possibly depending on a a variable \( x \).
  - E.g. \( C[x] \equiv x \) or \( C[x] \equiv D \ x \) for some other term \( D \) or \( C[x] \equiv \lambda y.x \).
- \( C[t] \) stands for the result of replacing \( x \) by \( t \), so for \( C[x][x := t] \).
Problem of the Missing \( \eta \)-Rule

Most problems of the missing \( \eta \)-rule occur in context with the missing \( \eta \)-rule for the product, to be discussed later.

In case of the \( \eta \)-rule for the function type, the problem occurs if we have

- a dependent type \( C[x] \) depending on \( x : A \to B \).
- know that \( s[x] : C[x] \) for some term
- and want to prove \( s[t] : C[t'] \) where \( t =_{\eta} t' \).
  - e.g. \( s[f] : C[\lambda x. f x] \).
  - \( C[x] \) can involve an equality type on \( A \to B \), or it can be a predicate \( C[x] \equiv D x \) for some \( D : (A \to B) \to \text{Set} \).

In this case it might not be provable that \( s[t] : C[t'] \), although intuitively one expects so.

Expensiveness of the \( \eta \)-Rule

The reason for not having the \( \eta \)-rule is that it is computationally expensive.

- Whether we can expand \( s \) to \( \lambda x^\sigma. s x \) depends on, whether \( s : \sigma \to \tau \) for some \( \tau \).
- All other reductions can be applied to terms without knowing the exact type of the term – in case of \( \eta \), we need to know this type.
- So, in order to allow \( \eta \)-rule as part of the reduction system, we need to carry along with each term (and as well with each subterm) its type, which requires a substantial additional overhead.

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

\[
\Gamma, \Delta \vdash t : \sigma.
\]

Then we have as well

\[
\Gamma, x : \tau, \Delta \vdash t : \sigma,
\]

provided \( \Gamma, x : \tau, \Delta \vdash t : \sigma \), is a context (i.e. provided \( x \) does not occur in \( \Gamma, \Delta \)).

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

\[
\frac{\frac{y : o, x : o \Rightarrow x : o}{(Abs)}}{y : o \Rightarrow \lambda x^0 . x : o} (\text{Ap})
\]

we obtain a derivation of

\[
y : o, x : o \Rightarrow \lambda x^0 . x y : o
\]
as follows:

Example (Weakening)

First rename in this derivation $x$ by $u$ in order to avoid conflicts (note that $\lambda x^0 . x$ is $\alpha$-equivalent to $\lambda u^0 . u$):

\[
\frac{\frac{y : o, u : o \Rightarrow u : o}{(Abs)}}{y : o \Rightarrow \lambda u^0 . u : o} (\text{Ap})
\]
Example (Weakening)

\[
\begin{align*}
\frac{y : o, u : o \Rightarrow u : o}{y : o \Rightarrow \lambda u^o. u : o^2} \quad (\text{Abs})
\end{align*}
\]

Now we obtain the following derivation of

\[
\begin{align*}
y : o, x : o & \Rightarrow \lambda x^o. x y : o \\
\frac{y : o, x : o, u : o \Rightarrow u : o}{y : o, x : o \Rightarrow \lambda u^o. u : o^2} \\
\frac{y : o, x : o \Rightarrow y : o}{y : o, x : o \Rightarrow \lambda u^o. u y : o} \quad (\text{Ap})
\end{align*}
\]

Note that \( \lambda x^o. x \) and \( \lambda u^o. u \) are identified.

Weakening

Because of the possibility of weakening, we will usually omit unused parts of contexts.

So a derivation of \( x : o^2, y : o \Rightarrow x (x y) : o \), which in full reads as follows

\[
\begin{align*}
\frac{x : o^2, y : o \Rightarrow x : o^2}{x : o^2, y : o \Rightarrow x (x y) : o} & \quad (\text{Ap})
\end{align*}
\]

will usually be presented as follows:

\[
\begin{align*}
\frac{x : o^2 \Rightarrow x : o^2}{x : o^2 \Rightarrow x : o^2} & \quad \frac{y : o \Rightarrow y : o}{y : o \Rightarrow y : o} \quad (\text{Ap})
\end{align*}
\]

Self-Application

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

\[
\omega := \lambda x. x x, \quad (\text{where } x \text{ was applied to itself})
\]

and had

\[
\Omega := \omega \omega.
\]

In the following, we will investigate, how self-application is avoided in the typed \( \lambda \)-calculus.

Self-Application

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. \( \lambda x^\sigma. x : \tau \).

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 then

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

\[ x : \sigma \Rightarrow x \colon \tau_1 \]  
\[ \lambda x^\sigma x : \sigma \to \tau_1 \text{(Abs)} \]

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

\[ x : \sigma \Rightarrow x \colon \tau_2 \to \tau_1 \quad x : \sigma \Rightarrow x \colon \tau_2 \]  
\[ x : \sigma \Rightarrow x \colon \tau_2 \]  
\[ \lambda x^\sigma x : \sigma \to \tau_1 \]  
\[ \lambda x^\sigma x : \sigma \to \tau_1 \text{(Abs)} \]  
\[ \text{(Ap)} \]

\( \Gamma \Rightarrow s : \sigma \quad \Gamma \Rightarrow t : \tau \) \[ \Gamma \Rightarrow \langle s, t \rangle : \sigma \times \tau \] \text{(Pair)}

The only way to derive \( x : \sigma \Rightarrow x \colon \tau_2 \to \tau_1 \) and \( x : \sigma \Rightarrow x \colon \tau_2 \) is by using the assumption rule.

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

Products

- The set of typed-\( \lambda \)-terms are defined as before but we have:
  - If \( s : \sigma, t : \tau \) then \( \langle s, t \rangle \) is defined as:

\[ \Gamma \Rightarrow s : \sigma \quad \Gamma \Rightarrow t : \tau \] \[ \Gamma \Rightarrow \langle s, t \rangle : \sigma \times \tau \] \text{(Pair)}

- If \( s : \sigma \times \tau \), then \( \pi_0(s) : \sigma \) and \( \pi_1(s) : \tau \):

\[ \Gamma \Rightarrow s : \sigma \times \tau \] \[ \Gamma \Rightarrow \pi_0(s) : \sigma \] \text{(Proj)}

\[ \Gamma \Rightarrow s : \sigma \times \tau \] \[ \Gamma \Rightarrow \pi_1(s) : \tau \] \text{(Proj)}
**Example**

We show

\[
(\lambda x^{(o \rightarrow o) \times (o \rightarrow o) \times o} . \pi_0(x)) \langle \lambda y^o . y , \lambda z^o . \lambda v^o . z \rangle : o \rightarrow o
\]

\[
(\lambda x^{(o \rightarrow o) \times (o \rightarrow o)} . \pi_0(x)) \langle \lambda y^o . y , \lambda z^o . \lambda v^o . z \rangle
\]

\[
(\lambda x^{(o \rightarrow o) \times (o \rightarrow o)} . \pi_0(x)) \langle \lambda y^o . y , \lambda z^o . \lambda v^o . z \rangle
\]

\[
((o \rightarrow o) \times (o \rightarrow o)) \rightarrow o
\]

**β-Reduction for Pairs**

- β-reduction for the pairs is the rule which allows to replace
  - any subterm of the form \( \pi_0(\langle r_0, r_1 \rangle) \) by \( r_0 \),
  - any subterm of the form \( \pi_1(\langle r_0, r_1 \rangle) \) by \( r_1 \).

The subterms \( \pi_i(\langle r_0, r_1 \rangle) \) are as before called \( \beta \)-redexes of the term in question.

- β-reduction for the typed \( \lambda \)-calculus with products includes both β-reduction for functions and β-reduction for pairs.

**Products with many Components**

- We write \( \sigma_0 \times \cdots \times \sigma_n \) for \( (\cdots ((\sigma_0 \times \sigma_1) \times \sigma_2) \cdots) \times \sigma_n \).
- Define for \( s_0 : \sigma_0, \ldots, s_n : \sigma_n \)
  \[
  \langle s_0, \ldots, s_n \rangle := \langle \cdots \langle \langle s_0, s_1 \rangle, s_2 \rangle, \cdots, s_n \rangle : \sigma_0 \times \cdots \times \sigma_n
  \]
- One can easily define corresponding projections \( \pi_i^n : (\sigma_0 \times \cdots \times \sigma_n) \rightarrow \sigma_i \), s.t.
  \[
  \pi_i^n(\langle s_0, \ldots, s_n \rangle) =_\beta s_i
  \]
- For instance in case \( n = 2 \) we need
  \[
  \pi_i^2(\langle \langle r_0, r_1 \rangle, r_2 \rangle) = r_i
  \]
Products with many Components

\[ \pi^2_i((r_0, r_1), r_2)) = r_i \]

We obtain this by defining

\[ \pi^2_0(x) := \pi_0(\pi_0(x)) \]

Then

\[ \pi^2_0((r_0, r_1), r_2)) = \pi_0(\pi_0((r_0, r_1), r_2)) = \pi_0((r_0, r_1)) = r_0 \]

\[ \pi^2_1(x) := \pi_1(\pi_0(x)) \]

Then

\[ \pi^2_1((r_0, r_1), r_2)) = \pi_1(\pi_0((r_0, r_1), r_2)) = \pi_1((r_0, r_1)) = r_1 \]

\[ \pi^2_2((r_0, r_1), r_2)) = \pi_1(\pi_1(\pi_0((r_0, r_1), r_2))) \]

\[ = \pi_1(\pi_1((r_0, r_1), r_2)) = \pi_1((r_0, r_1)) = r_2 \]

\eta\text{-Expansion for Products}

If we have a product \( r : \sigma \times \tau \), then its projections are \( \beta \)-equal to the projections of \( \langle \pi_0(r), \pi_1(r) \rangle \):

\[ \pi_0(\langle \pi_0(r), \pi_1(r) \rangle) =_\beta \pi_0(r), \]

\[ \pi_1(\langle \pi_0(r), \pi_1(r) \rangle) =_\beta \pi_1(r). \]

Therefore, similarly to functions, we would like to have that every term \( r : \sigma \times \tau \) is equal to \( \langle \pi_0(r), \pi_1(r) \rangle \).

\eta\text{-Rule for Products}

The \( \eta \)-rule expresses that subterms \( t : \sigma \times \tau \) can be \( \eta \)-expanded to \( \langle \pi_0(t), \pi_1(t) \rangle \).

However, as for functions, we need to impose some restrictions, in order to avoid circularities:

If \( t \) is of the form \( \langle r_0, r_1 \rangle \), and if we allowed then the reduction \( t \longrightarrow \langle \pi_0(t), \pi_1(t) \rangle \), we would get the following circular reduction:

\[ t \longrightarrow \langle \pi_0(t), \pi_1(t) \rangle \]

\[ \equiv \langle \pi_0(\langle r_0, r_1 \rangle), \pi_1(\langle r_0, r_1 \rangle) \rangle \]

\[ \longrightarrow^* \langle r_0, r_1 \rangle \]

\[ \equiv t \]
\(\eta\)-Rule for Products

- If \( t \) occurs in the form \( \pi_i(t) \), and if we then allowed to expand \( t \), we would get \( \pi_i(t) \rightarrow \pi_i(\langle \pi_0(t), \pi_1(t) \rangle) \) and would get the following circular reduction:

\[
\pi_i(t) \rightarrow \pi_i(\langle \pi_0(t), \pi_1(t) \rangle) \\
\rightarrow_\beta \pi_i(t),
\]

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

\(\eta\)-Expansion for Products

- \(\eta\)-expansion for products is the rule which allows to replace in a typed \(\lambda\)-term \( t \)
  - one subterm \( s : \sigma \times \tau \),
  - which is not of the form \( \langle r_0, r_1 \rangle \),
  - and does not occur in the form \( \pi_0(s) \) or \( \pi_1(s) \)
  by \( \langle \pi_0(s), \pi_1(s) \rangle \).
- \(\eta\)-expansion for the typed \(\lambda\)-calculus with products includes both \(\eta\)-expansion for functions and for pairs.

\(\beta,\eta\)-normal form

The \(\lambda\)-calculus with products, \(\beta\)-reduction and with (or without) \(\eta\)-expansion is confluent and strongly normalising.

We can introduce products as well for the untyped \(\lambda\)-calculus. Then we obtain a confluent (but of course non normalising) reduction system.
\textbf{η-Rule}

- With the η-rule we obtain now that if \( r : \sigma \times \tau \), then \( r =_{\beta, \eta} \langle \pi_0(r), \pi_1(r) \rangle \).

- If \( r : \sigma \times \tau \) is of the form \( \langle r_0, r_1 \rangle \) then we have:
  \[
  r =_{\beta} \langle \pi_0(\langle r_0, r_1 \rangle), \pi_1(\langle r_0, r_1 \rangle) \rangle \\
  \sim^*_{\beta} \langle r_0, r_1 \rangle \\
  \equiv \langle r_0, r_1 \rangle
  \]

- Otherwise \( r \rightarrow_{\eta} \langle \pi_0(r), \pi_1(r) \rangle \).

Therefore, every element of a product type is of the form \( \langle \text{something}_0, \text{something}_1 \rangle \).

\textbf{j) The Nondep. Product in Agda}

In Agda, there are two ways of defining the product.

- The first one represents the product as a \textit{record type}.
- Assume we have introduced \( A, B :: \text{Set} \).
  Then we can introduce the record type
  \[
  AB :: \text{Set} \\
  = \text{sig}\{a :: A; b :: B\}
  \]

- One can introduce longer record types as well, e.g.
  \[
  \text{sig}\{a :: A; b :: B; c :: C; e :: E\}
  \]

\textbf{The Product as a Record Type}

\( AB = \text{sig}\{a :: A; b :: B\} \)

- Elements of a record type are introduced as follows:
  Assume we have \( a' :: A, b' :: B \).
  Then we can introduce in the above situation
  \[
  ab :: AB \\
  = \text{struct}\{a = a'; b = b'\}
  \]

- Elimination of elements of the record type is record selection.
  Assume \( ab :: AB \).
  Then we have:
  \[
  ab.a :: A \\
  ab.b :: B
  \]

- If \( ab = \text{struct}\{a = a'; b = b'\} \), then we have:
  \[
  ab.a \text{ is equal to } a' \\
  ab.b \text{ is equal to } b'
  \]
Intro

- Usually, we don’t have to spell out
  \[
  \text{struct}\{a = a'; b = b'\}
  \]
  in full.
- Assume we have a goal, the type of which is a sig-type, e.g.
  \[
  AB :: \text{Set} = \text{sig}\{a :: A; b :: B\}
  
  ab :: AB = \{! !\}
  \]

The Product using “data”

- The second version of the product uses the more general data construct for defining so called algebraic types.
- With this construction we are leaving the so called logical framework.
  - \(\lambda\)-terms and the builtin product form the logical framework, the basic types of Agda and of Martin-Löf type theory.
  - The data-construct allows to introduce user-defined types.

\[
\text{Prod} (A :: \text{Set})
\]
\[
(B :: \text{Set})
\]
\[
:: \text{Set}
\]
\[
= \text{data} p (a :: A) (b :: B)
\]

- The “data”-product is introduced as follows:
- Here
  - \(p\) is the constructor of this set.
  - The name (here \(p\)) is up to the user, we could have used any other valid Agda identifier.
- The idea is:
  - The elements of Prod are exactly the terms \(p\ a\ b\) where \(a :: A\) and \(b :: B\).
In order to allow type inference, we cannot write directly $p @ a b$.

$p @ a b$ could be an element of $\Prod A B$ for any $A, B :: \Set$.

Therefore $p$ has to be given a qualifier which indicates, to which set it belongs to:

One writes $p @ (\Prod A B)$

for the $p$ belonging to $\Prod A B$.

$p @_ {\_}$

However, we are not allowed to state

$p @_ :: A \rightarrow B \rightarrow \Prod A B$,

since in this situation $p @_ {\_}$ is not applied to sufficiently many arguments.

We have to write instead

$p @ (\Prod A B) :: A \rightarrow B \rightarrow \Prod A B$.

Case Distinction

In order to decompose an element of $\Prod A B$ in Agda, we can use case distinction.

This is best explained by an example.

We postulate $A, B :: \Set$, and abbreviate $\Prod A B$ as $AB$:

\begin{align*}
\text{postulate } A :: \Set \\
\text{postulate } B :: \Set \\
AB :: \Set \\
&= \Prod A B
\end{align*}

$p @_ {\_}$ is a long notation, which is rarely spelled out in full.

If we apply a $p$ to sufficiently many arguments so that we get an element of a type of the form “data · · ·”, and that type can be derived from the context, then we can write $p @_ {\_}$ instead of $p @ (\Prod A B)$.

For instance, if $a :: A$ and $b :: B$, then we have

$p @_ a b :: \Prod A B$
Case Distinction

postulate $A :: \text{Set}$
postulate $B :: \text{Set}$
$AB :: \text{Set} = \text{Prod} A B$

- Assume we want to define the first projection
  
  $$\text{proj}_0 : AB \rightarrow A,$$
  
s.t.
  
  $$\text{proj}_0 (p@_a b) = a$$

- So we have the goal:
  
  $$\text{proj}_0 \ (ab :: AB) :: A = \{! !\}$$

Case Distinction (Cont.)

- The value of $ab$ in the first goal can be tested as follows:
  
  - Position the cursor in the first goal and choose one of the methods for evaluating expressions, e.g. compute weak head normal form strict.
  - Then type into the mini-buffer $ab$.
  - One gets the answer
    
    $$p@_a b.$$  

Case Distinction (Cont.)

- Alternatively, check, the cursor being in that goal, the context
  
  - (use goal-menu “agda-context”):
  
  - It contains
    
    $$a :: A$$
    
    $$b :: B$$
    
    $$ab :: AB = p@_a b$$

- Note that $a, b$ are only local variables, which don’t exist outside the case distinction.
Case Distinction (Cont.)

- Now we can solve the new goal by inserting \( a \) in it and refining it.
- We obtain the function:

\[
\text{proj} \ 0 \ (ab :: AB) :: A = \text{case } ab \text{ of} \ (p \ a \ b) \rightarrow a
\]

Testing the Defined Function

- We can test our function by using one of the evaluation methods of Agda, e.g.
  - compute weak head normal form strict.
- Postulate first \( a :: A, b :: B \).
- We have to create a dummy goal for this:
  
  \[
  \text{test} :: \text{Set} = \{! !\}
  \]

- Then we move to the new goal and choose compute weak head normal form strict or another evaluation method of Agda,
- and type into the mini-buffer proj\(0 \ (p @ a \ b)\).
- The result shown is \( a \).

The Product in Agda

- Unfortunately, the product based on the record type in Agda does not behave very well.
- This is due to the fact that Agda doesn’t support the \( \eta \)-rule.
- In this setting \( \eta \)-equality would assert that if
  
  \[
  c :: \text{sig}\{a :: A; b :: B\}
  \]

  then
  
  \[
  c = \text{struct}\{a = c.a; b = c.b\}
  \]

Testing the Defined Function

- \( \eta \)-expansion for products is computationally expensive for the same reasons as it was for functions.
- Whether we can expand \( c \) to
  
  \[
  \text{struct}\{a = c.a; b = c.b\}
  \]

  depends on, whether
  
  \[
  c :: \text{sig}\{a :: A; b :: B\}
  \]

- The problem with the missing \( \eta \)-rule can often be avoided using the version of the product based on “data”.

Unfortunately, the product based on the record type in Agda does not behave very well.

- This is due to the fact that Agda doesn’t support the \( \eta \)-rule.
- In this setting \( \eta \)-equality would assert that if
  
  \[
  c :: \text{sig}\{a :: A; b :: B\}
  \]

  then
  
  \[
  c = \text{struct}\{a = c.a; b = c.b\}
  \]

- \( \eta \)-expansion for products is computationally expensive for the same reasons as it was for functions.
- Whether we can expand \( c \) to
  
  \[
  \text{struct}\{a = c.a; b = c.b\}
  \]

  depends on, whether
  
  \[
  c :: \text{sig}\{a :: A; b :: B\}
  \]

- The problem with the missing \( \eta \)-rule can often be avoided using the version of the product based on “data”.
Problem of the Missing $\eta$-Rule

The exact problem of the missing $\eta$-rule is as follows:

- Assume we have
  \[ c :: \text{sig}\{a :: N; b :: N\} \] .

- If we then make case distinction on $c.a : N$, we know in case of $c.a = Z$ (standing for $c$=zero) not that
  \[ c = \text{struct}\{a = Z; b = s.b\} \] .

- This is since we don’t know without the $\eta$-rule, that
  \[ c = \text{struct}\{a = c.a; b = c.b\} \] .

- If we had the $\eta$-rule we would get
  \[ c = \text{struct}\{a = c.a; b = c.b\} = \text{struct}\{a = Z; b = c.b\} \] .

Working Around the Missing $\eta$-Rule

One can often work around the missing $\eta$-rule by replacing types depending on “sig”-types by types depending on its components.

- E.g. assume we have
  \[ \text{Student} = \text{sig}\] 
  name:: String
  gender:: Gender

If we use instead of sig the product based on data, we can make

- case distinction on an element, e.g. $s = p \ a \ b$,
- and if we then make case distinction on $a$, we get in case
  \[ a = Z \] ,
- that
  \[ c = p \ Z \ b . \]
For sig, struct, let, case, and some other constructs to be introduced later (packages, probably some more) there are two versions for determining the scope of the construct:

Using "{" and "}".

The scope is what is enclosed in those brackets.

The selections must be separated by ";".

For instance

\[ \text{struct}\{a = a0; b = b0}\} :: \text{sig}\{a :: A; b :: B}\]
So the type of $f$ in Agda is as follows:

\[ AB :: Set = \text{sig}\{a :: A; b :: B\} \]
\[ CD :: Set = \text{sig}\{c :: C; d :: D\} \]
\[ f (a _c :: A \to C) \]
\[ (b _d :: B \to D) \]
\[ (ab :: AB) \]
\[ :: CD \]
\[ = \{! !\} \]

The idea for this function is as follows:

- We first project $ab$ to elements $a : A$, $b : B$.
- Then we apply $a _c$ to $a$ and obtain an element $c : C$, and $b _d$ to $b$ and obtain an element $d : D$.
- Finally we form the pair $\langle b, d \rangle$.

We will use let-expressions in order to compute the intermediate values $a$, $b$, $c'$, $d'$.

Agda Code for the Above

\[ AB :: Set = \text{sig}\{a :: A; b :: B\} \]
\[ CD :: Set = \text{sig}\{c :: C; d :: D\} \]
\[ f (a _c :: A \to C) \]
\[ (b _d :: B \to D) \]
\[ (ab :: AB) \]
\[ :: CD \]
\[ = \text{let} \ a :: A = ab . a \]
\[ b :: B = ab . b \]
\[ c' :: C = a _c . a \]
\[ d' :: D = b _d . b \]
\[ \text{in struct}\{c = c'; d = d'\} \]

See `exampleLetExpressionSig.agda`.
Open Expression

Open abbreviates the unpacking of pairs.

Assume
\[ e :: \text{sig}\{a_1 :: A_1; a_2 :: A_2; \ldots\} \]

Then
\[ \text{open } e \text{ use } x_1 :: A_1 = a_1, \]
\[ x_2 :: A_2 = a_2, \]
\[ \ldots \]
\[ \text{in } \ldots \]

is equivalent to
\[ \text{let } x_1 :: A_1 = e.a_1; \]
\[ x_2 :: A_2 = e.a_2; \]
\[ \ldots \]
\[ \text{in } \ldots \]

So the above example reads
\[
\begin{align*}
f(a_{-c} :: A \rightarrow C) \\
b_{-d} :: B \rightarrow D \\
(ab :: AB) \\
:: CD \\
= \text{open } ab \\
\quad \text{use } a :: A = a, \\
\quad b :: B = b \\
\quad \text{in let } c' :: C = a_{-c} a \\
\quad \quad d' :: D = b_{-d} b \\
\quad \quad \text{in struct}\{c = c'; d = d'\}
\end{align*}
\]

See exampleOpenExpression1.agda.

Open Expression

If we want to use the selectors directly, i.e. assuming
\[ e :: \text{sig}\{a_1 :: A_1; a_2 :: A_2; \ldots\} \]

to use the names \(a_1, a_2, \ldots\), we can write more briefly
\[ \text{open } e \text{ use } a_1, a_2, \ldots \]
\[ \text{in } \ldots \]

So the above example reads more briefly
\[
\begin{align*}
f(a_{-c} :: A \rightarrow C) \\
b_{-d} :: B \rightarrow D \\
(ab :: AB) \\
:: CD \\
= \text{open } ab \text{ use } a, b \\
\quad \text{in let } c' :: C = a_{-c} a \\
\quad \quad \quad d' :: D = b_{-d} b \\
\quad \quad \quad \text{in struct}\{c = c'; d = d'\}
\end{align*}
\]

See exampleOpenExpression2.agda.
Termination Checker and Products

When using products in Agda, it often happens that one writes by mistake black-hole recursive functions.
- Due to the syntax which is easily misunderstood.
- Assume for instance we have defined \( a :: A \) and \( b :: B \), and \( AB = \text{sig}\{a :: A; b :: B\} \).
- If we define
  \[
  ab :: AB = \text{struct}\{a = a; b = b\}
  \]
  we have
  - not defined an element \( ab \) s.t. \( ab.a = a \) and \( ab.b = b \),
  - but defined an element \( ab \) s.t. \( ab.a = ab.a \) and \( ab.b = ab.b \).
- That is a black-hole recursion, and will not pass the termination checker.

Concrete Products

When using the data-construct, it is often more convenient to introduce concrete products in a more direct way.
- **Example:** Assume we have defined
  - a set `Gender` of genders,
  - a set `Name` of names.
  - The set of `persons`, given by a gender and a name, can then be defined as

\[
\text{Person} :: \text{Set} = \text{data}\ \text{person} (g :: \text{Gender})(n :: \text{Name})
\]

Concrete Products

Not surprisingly, for **elimination** we use
- **case distinction**, e.g.:

\[
\begin{align*}
\text{gender} \ (p :: \text{Person}) \\
:: \text{Gender} \\
= \text{case } p \text{ of} \\
\ (\text{person } g \ n) \rightarrow g
\end{align*}
\]

Constructive Meaning of \( \land \)

\( A \land B \) is true, if \( A \) is true and \( B \) is true.
- Therefore a proof \( p : A \land B \) consists
  - of a proof \( a : A \)
  - and a proof \( b : B \).
- So such a proof is a pair \( \langle a, b \rangle \) s.t. \( a :: A \) and \( b :: B \).
- Therefore \( A \land B \) is just the product \( A \times B \) of \( A \) and \( B \).
- We can identify \( A \land B \) with \( A \times B \).
Conjunction in Agda

- Conjunction is represented as a product.
- There are two products in Agda, therefore as well two ways of representing conjunction:
  - One using the logical framework product:
    \[
    \text{AND1} \quad (A, B :: \text{Set})
    \]
    \[
    :: \text{Set}
    \]
    \[
    = \text{sig}
    \]
    \[
    \quad \quad \text{and1} :: A
    \]
    \[
    \quad \quad \text{and2} :: B
    \]

Conjunction in Agda

- One can write as well \( \land \) for one of the versions of conjunction, and use it infix.
- We write on slides \( \land \) for it, and get therefore:
  \[
  (\land) \quad (A, B :: \text{Set})
  \]
  \[
  :: \text{Set}
  \]
  \[
  = \text{data and}(a :: A)(b :: B)
  \]
  \[
  AB :: \text{Set}
  \]
  \[
  = A \land B
  \]

Example (Conjunction)

- We prove \((A \land B) \rightarrow (B \land A)\) (see exampleproofpropllogic6.agda):
  \[
  \text{lemma2a} \quad (ab :: \text{AND1} A B)
  \]
  \[
  :: \text{AND1} B A
  \]
  \[
  = \text{struct}
  \]
  \[
  \quad \quad \text{and1} = ab.\text{and2}
  \]
  \[
  \quad \quad \text{and2} = ab.\text{and1}
  \]
  \[
  \text{lemma2b} \quad (ab :: \text{AND2} A B)
  \]
  \[
  :: \text{AND2} B A
  \]
  \[
  = \text{case} \; ab \; \text{of}
  \]
  \[
  \quad \quad (\text{and} \; a \; b) \rightarrow \text{and@} \; b \; a
  \]
Conjunction with more Conjuncts

- If one has a conjunction with more than two conjuncts, e.g. \( A \land B \land C \), one can always express it using the binary \( \land \):
  - As \( (A \land B) \land C \) or \( A \land (B \land C) \).
- But it is often more convenient to use a ternary version of conjunction (using one of the two versions of the product).
- Similarly one can introduce conjunctions of 4 or more conjuncts.

\[ \text{CONJUNCTION WITH MORE CONJUNCTS} \]

\[ \text{AND3a} \quad (A, B, C :: \text{Set}) \]
\[ :: \text{Set} \]
\[ = \text{sig} \]
\[ \quad \text{and1} :: A \]
\[ \quad \text{and2} :: B \]
\[ \quad \text{and3} :: C \]

\[ \text{AND3b} \quad (A, B, C :: \text{Set}) \]
\[ :: \text{Set} \]
\[ = \text{data and3} (a :: A) (b :: B) (c :: C) \]

See `exampleproofprologic5.agda`

(k) The \( \lambda \)-Calc. and Term Rewriting

- One can combine the \( \lambda \)-calculus with term writing.
- This means that we have apart from the rules of the typed or untyped \( \lambda \)-calculus additional rules like \( x + 0 \longrightarrow x \).
  - Then we obtain for instance
    \[ \lambda y. \lambda z. y + 0 \longrightarrow \lambda y. \lambda z. y \]
- More details are given on the following slides but will not be treated in this lecture.
  Jump over rest of this section.
\textbf{\textit{\textlambda} -Calculus and Term Rewriting}

Then $s \rightarrow t$, if
\begin{itemize}
  \item $s$ $\beta$ -reduces (or $\eta$ -expands, if one allows the $\eta$ -rule) to $t$
  \item or there exists an instantiation $s' \rightarrow t'$ s.t. $s'$ is a subterm of $s$ and $t$ is the result of replacing this subterm in $s$ by $t'$.
\end{itemize}
$s'$ is called as usual a \textit{redex} of $s$.

\textbf{What does Subterm Mean?}

When referring to ordinary term rewriting rules, then for a term $t$ to have subterm $s$ meant essentially that there is a term $t'$ in which a new variable $x$ occurs exactly once, and $t = t'[x := s]$.

Replacing this subterm by $s'$ means that we replace $t$ by $t'[x := s']$.

\textbf{\textit{\textlambda} -Calculus and Term Rewriting}

Assume for instance the rule
\[
\text{double} \rightarrow \lambda x. x + x
\]
Then we have
\[
(\lambda f. \lambda x. f (f x)) \text{double} \\
\rightarrow \lambda x. \text{double} (\text{double } x) \\
\rightarrow \lambda x. \text{double} ((\lambda x. x + x) x) \\
\rightarrow \lambda x. \text{double} (x + x) \\
\rightarrow \lambda x. (\lambda x. x + x) (x + x) \\
\rightarrow \lambda x. (x + x) + (x + x)
\]

When referring to $\textit{\textlambda}$ -terms, this is no longer the case:
\begin{itemize}
  \item Assume for instance the rewrite rule $x + 0 \rightarrow \text{Rule } x$.
  \item $\lambda x. x + 0$ has subterm $x + 0$, but there is no term $t$ s.t. $\lambda x. x + 0 = t[y := x + 0]$.
  \item If we substitute for instance in $\lambda x. y y$ by $x + 0$ we obtain $\lambda z. x + 0$.
\end{itemize}
The reason is that when matching a rewrite rule, free variables in the instantiation of the rule used might become bound.

So we can apply $x + 0 \rightarrow \text{Rule } x$ to $\lambda x. x + 0$ and have therefore $\lambda x. x + 0 \rightarrow \lambda x. x$.

Replacing a subterm by another subterm is to be understood verbally.
Higher Order Rewrite Systems

- The full definition of so called higher order term rewriting systems imposes more restrictions on the reduction rules.
- For our purposes the naive interpretation just presented suffices.

Jump over next part.

Reduction to Closed Terms

- One can always replace term rewriting rules for the $\lambda$-calculus by one in which for all rules $s \rightarrow \text{Rule} t$ we have that $s, t$ are closed.
- This can be done in such a way that equality (modulo the rewriting rules, $\beta$ and possibly $\eta$) in both systems coincide:
- Assume a rule $s \rightarrow \text{Rule} t$
- and let $x_1, \ldots, x_n$ be the free variables in $s$.
- Then replace this rule by $\lambda x_1, \ldots, x_n. s \rightarrow \text{Rule}' \lambda x_1, \ldots, x_n. t$.

Proof

- We write in the following $\vec{x}$ for $x_1, \ldots, x_n$.
- Assume a term $r$ reduces using this rule in the original system to a term $u$:
- Then $r$ contains a subterm of the form $s'$ where $s'$ is the result of substituting in $s x_i$ by some terms $t_i$.
- Let $t'$ be the result of substituting in $t x_i$ by $t_i$. Then $u$ is the result of replacing $s'$ in $r$ by $t'$.
- Let then $r'$ be the result of replacing $s'$ by $(\lambda \vec{x}. s) t_1 \cdots t_n$, and $u'$ be the result of replacing in $s s'$ by $(\lambda \vec{x}. t) t_1 \cdots t_n$.
- Then we have $r =_\beta r' \rightarrow \text{Rule}' u' =_\beta u$, so the reduction can be simulated in the second system.

On the other hand, if $r \rightarrow u$ by using in the second system the rule $\lambda \vec{x}. s \rightarrow \text{Rule}' \lambda \vec{x}. t$, then $r \rightarrow u$ in the previous system by using the rule $s \rightarrow \text{Rule} t$:
- $r$ contains a subterm equal to $\lambda \vec{x}. s$ and $u$ is the result of substituting this subterm in $r$ by $\lambda \vec{x}. t$.
- But then $r$ contains the subterm $s$ and $t$ is the result of substituting this subterm in $r$ by $t$. 

Proof
Example

We can replace the rewriting rules
\[
\begin{align*}
x + 0 & \rightarrow x \\
x + S\ y & \rightarrow S\ (x + y)
\end{align*}
\]
by
\[
\begin{align*}
\lambda x. x + 0 & \rightarrow \lambda x. x \\
\lambda x, y. x + S\ y & \rightarrow \lambda x, y. S\ (x + y)
\end{align*}
\]
That
\[
S\ (0 + S\ 0) \rightarrow S\ (S\ (0 + 0)) \rightarrow S\ (S\ 0)
\]
becomes in the new system
\[
\begin{align*}
S\ (0 + S\ 0) &= S\ ((\lambda x, y. x + S\ y)\ 0\ 0) \\
&\rightarrow S\ ((\lambda x, y. S\ (x + y))\ 0\ 0) = S\ (S\ (0 + 0)) \\
&= S\ (S\ ((\lambda x. x + 0)\ 0)) \rightarrow S\ (S\ ((\lambda x. x)\ 0)) = S\ (S\ 0)
\end{align*}
\]

Extended Typed $\lambda$-Calculus

Finally, we can combine the typed $\lambda$-calculus (with or without products, with or without $\eta$-expansion) with term rewriting rules.

Essentially this means that we have additional constants with types and reduction rules for them.

The details (which are given on the following slides) will not be treated in the lecture itself.

Example

Assuming $(+) : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$ and writing as usual $r + s$ for $(+) r s$ we have the following derivation of $\lambda x^{\text{nat}}. x + x : \text{nat} \rightarrow \text{nat}$:

\[
\begin{align*}
\text{Abs} & : x : \text{nat} \Rightarrow (+) x x : \text{nat} \\
\text{Ap} & : x : \text{nat} \Rightarrow (+) x x : \text{nat} \rightarrow \text{nat}
\end{align*}
\]

The left most leaf in this derivation follows by the rule for the constant $(+)$. 

For introducing the new rewrite rules, we have to make the following modifications:

- We assign a type to each additional constant.
- The set of typed $\lambda$-terms is then introduced by the same rules as before, but we have as additional rule:
- If $c$ is a constant of type $\sigma$, then we have $\Gamma \Rightarrow c : \sigma$
Example

Then we have

\[(\lambda x^{\text{nat}}. f (f x)) \text{double} \]
\[\rightarrow \lambda x^{\text{nat}}. \text{double} (\text{double} x)\]
\[\rightarrow \lambda x^{\text{nat}}. \text{double} ((\lambda x^{\text{nat}}. x + x) x)\]
\[\rightarrow \lambda x^{\text{nat}}. \text{double} (x + x)\]
\[\rightarrow \lambda x^{\text{nat}}. (\lambda x^{\text{nat}}. x + x) (x + x)\]
\[\rightarrow \lambda x^{\text{nat}}. (x + x) + (x + x)\]

Extended Typed $\lambda$-Calculus

- Instantiations of a rule $\Gamma \Rightarrow s \longrightarrow_{\text{Rule } t : \sigma}$ are now obtained by replacing variables $x$ of type $\tau$ by terms $r : \tau$ (possibly depending on some context $\Delta$).
- Reductions w.r.t. the rules are obtained by replacing subterms $r : \sigma$, which coincide with the left hand side of an instantiation of a rule $r \longrightarrow r' : \sigma$ by the right hand side $r'$.

Example

Assume

- ground type $\text{nat}$,
- constants $(+) : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$ (written infix, i.e. $r + s$ for $(+ r s)$),
- and $\text{double} : \text{nat} \rightarrow \text{nat}$.
- and the reduction rule
  $\text{double} \longrightarrow (\lambda x^{\text{nat}}. x + x) : \text{nat} \rightarrow \text{nat}$.

Extended Typed $\lambda$-Calculus

- Reduction rules should now be of the form
  $\Gamma \Rightarrow s \longrightarrow_{\text{Rule } t : \sigma}$ (instead of $s \longrightarrow_{\text{Rule } t}$) where we have $\Gamma \Rightarrow s : \sigma$ and $\Gamma \Rightarrow t : \sigma$.
  - As before, $s$ shouldn’t be a variable, and all variables in $t$ should occur in $s$.
  - Best guaranteed by demanding that all variables in $\Gamma$ occur free in $s$.
  - One usually omits $\Gamma$, $\sigma$, if it is clear from the context.
- Very often, the reduction rules will be of the form
  $c \longrightarrow_{\text{Rule } t : \sigma}$ where $c$ is a constant and therefore $t$ a closed term.
Example

Then we have

\[
(\lambda f^{\text{nat} \to \text{nat}}. \lambda x^{\text{nat}}. f (f x)) \, \text{double}
\]
\[
\to \lambda x^{\text{nat}}. \text{double} \, (\text{double} \, x)
\]
\[
\to \lambda x^{\text{nat}}. \text{double} \, ((\lambda x. x + x) \, x)
\]
\[
\to \lambda x^{\text{nat}}. \text{double} \, (x + x)
\]
\[
\to \lambda x^{\text{nat}}. (\lambda x. x + x) \, (x + x)
\]
\[
\to \lambda x^{\text{nat}}. (x + x) + (x + x)
\]

(l) Currying

In the \(\lambda\)-calculus with products, there are two versions of a function \(f\) taking two integers and returning an integer:

- \(f_1 : (\text{int} \times \text{int}) \to \text{int}\)
- \(f_2 : \text{int} \to \text{int} \to \text{int}\).

We say

- that \(f_1\) is in **Uncurried** form,
- and \(f_2\) is in **Curried** form.

The name “Curry” honours Haskell Curry.

The application of these two functions to arguments \(x\) and \(y\) is written

- \(f_1(x, y)\),
- \(f_2 \, x \, y\).

Curried/Uncurried Functions

The above generalises to functions with arbitrarily (but finitely) many arguments of different type.

- The **Curried version** of a function \(f\) with arguments of types \(\sigma_0, \ldots, \sigma_n\) and result type \(\rho\) is of type

\[
\sigma_0 \to \cdots \to \sigma_n \to \rho .
\]

- Its **Uncurried version** has type

\[
(\sigma_0 \times \cdots \times \sigma_n) \to \rho .
\]

Haskell Brooks Curry (1900 - 1982)
Uncurrying

From a Curried function we can obtain an Uncurried function.

This is called **Uncurrying**.

**Example:**

Assume

\[ f : \text{int} \to \text{int} \to \text{int} . \]

Then

\[ \lambda x. \lambda y. f \ (x) \ (y) : (\text{int} \times \text{int}) \to \text{int} \]

is the **UnCurried** form of \( f \).

---

Currying

From a Uncurried function we can obtain an Curried function.

This is called **Currying**.

**Example:**

Assume

\[ f : (\text{int} \times \text{int}) \to \text{int} . \]

Then

\[ \lambda x. \lambda y. f \ (x) \ (y) : (\text{int} \times \text{int}) \to \text{int} \]

is the **Curried** form of \( f \).

---

Uncurrying

We can obtain from the Curried form \( f_{\text{Curry}} \) of a function its Uncurried form \( f_{\text{Uncurry}} \) by

\[ f_{\text{Uncurry}} = \lambda x. f_{\text{Curry}} \ (\pi_0^n (x)) \cdots \pi_n^n (x) \]

where \( \pi_i^n : (\sigma_0 \times \cdots \times \sigma_n) \to \sigma_i \) are the projections.

One can as well define a \( \lambda \)-term

\[ \text{Uncurry} : (\sigma_0 \to \cdots \to \sigma_n \to \rho) \to (\sigma_0 \times \cdots \times \sigma_n) \to \rho \]

\[ \text{Uncurry} := \lambda f, x. f \ (\pi_0^n (x)) \cdots \pi_n^n (x) \]

s.t. \( \text{Uncurry} \ f_{\text{Curry}} \beta \ f_{\text{Uncurry}} \).

This transformation is called **Uncurrying**.

---

Currying

We can obtain from the Uncurried form \( f_{\text{Uncurry}} \) of a function its Curried form \( f_{\text{Curry}} \) by

\[ f_{\text{Curry}} = \lambda x_0, \ldots, x_n. f_{\text{Uncurry}} \ (x_0, \ldots, x_n) \]

Again we can define

\[ \text{Curry} : ((\sigma_0 \times \cdots \times \sigma_n) \to \rho) \to \sigma_0 \to \cdots \to \sigma_n \to \rho \]

\[ \text{Curry} := \lambda f, x_0, \ldots, x_n. f \ (x_0, \ldots, x_n) \]

s.t. \( \text{Curry} \ f_{\text{Uncurry}} \beta \ f_{\text{Curry}} \).

This transformation is called **Currying**.

It is an easy exercise to show \( \text{Curry} \ (\text{Uncurry} \ f) = \beta, \eta \ f \) and \( \text{Uncurry} \ (\text{Curry} \ f) = \beta, \eta \ f \).
(Un)Currying in Programming

The Uncurried form of a function corresponds to the form functions are presented usually outside functional programming.

In functional programming one often prefers the Curried form.

This allows to apply a functional partially to its arguments.

E.g. if we take (+) as usual in Curried form, then (+) 3 : int → int is the function taking \( x \) and returning \( x + 3 \).

For instance map ((+) 3) [1, 2, 3] = [4, 5, 6]: If we apply the function increasing every \( x \) by 3 to the list [1, 2, 3], we obtain the result of incrementing each list element by 3, i.e. [4, 5, 6].

(U)Currying in Programming

One often avoids in functional programming (and as well in Agda) the formation of products (or record types).

Especially for intermediate calculations.

The packing and unpacking of products makes programming often harder.

E.g. instead of defining a function \( f : \sigma \rightarrow (\rho \times \tau) \) it is often better to form two functions \( f_1 : \sigma \rightarrow \rho \) and \( f_2 : \sigma \rightarrow \tau \), (which are often defined simultaneously).

Only, when delivering the final program, the use of products is often better, because the result is more compact.