B2. The Logical Framework

(a) Basic Form of Rules

(b) The non-dependent function type and product.

(c) The dependent function type and product.

(d) Structural rules.
(a) Basic Form of Rules

Four Kinds of Rules

- For each type construction we have usually 4 kinds of rules:
  1. Formation Rules.
  2. Introduction Rules.
  3. Elimination Rules.

- Additionally there are equality versions of the formation, introduction, and elimination rules.
(1) Formation Rules

- The **formation rules** introduce a new types.

- Each type construction has one such rule.

- The **conclusion** of such a rule will have the form:

  \[ C(a_1, \ldots, a_n) : \text{Type} \]

  - where \( C \) is a **type-constructor**,  
  - \( a_1, \ldots, a_n \) are its arguments.
Example 1: The Type of Lists

\[ A : \text{Type} \]
\[ \text{List}(A) : \text{Type} \]

- The **type-constructor** is **List**.
- List(A) is the type of lists of type A.
Example 2: The Type of Natural Numbers

- Formation rule for the type of natural numbers:

  \[ \text{N : Type} \]

  - The \textbf{type-constructor} is \textit{N}.
    \* It has 0 arguments, and we write \textit{N} instead of \textit{N()}.
    \* Note that the formation rule for \textit{N} has 0 premises (the bar is omitted).
  - Later we will see that we can replace in this example \textit{Type} with \textit{Set}.
Example 3: The Non-Dependent Product

- Formation rule for the non-dependent product:

\[
\frac{A : \text{Type} \quad B : \text{Type}}{A \times B : \text{Type}}
\]

- \(A \times B\) stands for \((\times)(A, B)\).
- The \textit{type-constructor} is \((\times)\).
• The formation of a type is usually done by introducing a constant of a certain type.

• **Example 1:**

```
List (A :: Type) :: Type
    = ...
```
• Agda syntax for introducing the **non-dependent product**:

\[(\times) \ (A : \text{Type}) \ (B : \text{Type}) \ :: \text{Type} = \ldots\]

– \ldots\ is an Agda definition of this type (more about this later).
– \(\times\) is Ascii symbol 215 (not the letter x).

• There are as well **predefined versions of the product** (and of the function type) in Agda (see later).
Currying in Agda

- Traditionally one writes in **type theory** type-constructors in **uncurried form**.
  - List alone does not make sense as a term. We have to write \( \text{List}(A) \).

- In Agda type constructors (except those predefined for dep. product and function type) are **always curried**.
  - So List alone is a term and has type (more precisely this is kind and not a type)
    \[
    \text{List} :: \text{Type} \rightarrow \text{Type}
    \]

  \* We write therefore \( \text{List } A \) and not \( \text{List}(A) \).
    - List \( A \) means the application of the function List to \( A \)
Currying in Agda (Cont.)

- \((\times)\) and \((\times)\ N\) are terms and have types (more precisely kinds)

\[
(\times) :: \text{Type} \rightarrow \text{Type} \rightarrow \text{Type} \\
(\times) N :: \text{Type} \rightarrow \text{Type}
\]

- The latter is the operation which takes a \(B\) and returns \(\times N\).

- Agda allows to write \(A \times B\) for \((\times)\ AB\).
(2) Introduction Rules

- The introduction rule introduces elements of a type.
- The conclusion of such a rule will have the form
  \[ C(a_1, \ldots, a_n) : A \]
  where
  - \( A \) is a type introduced by the corresponding formation rule,
  - \( C \) is a constructor or term-constructor,
  - \( a_1, \ldots, a_n \) are terms.

- \( C \) might have zero arguments, then we write \( C \) instead of \( C() \).
The type $\text{NatList}$ of $\text{Lists}$ of type $\text{N}$ has two introduction rules:

- $\text{nil} : \text{NatList}$
- $\frac{n : \text{N}}{\text{cons}(n, l) : \text{NatList}}$
• We generalize the previous example to lists of arbitrary type.

• **Lists** of type \( A \) have two introduction rules:

\[
\begin{align*}
A &: \text{Type} \\
\text{nil} &: \text{List}(A)
\end{align*}
\]

\[
\frac{a : A \quad l : \text{List}(A)}{\text{cons}(a, l) : \text{List}(A)}
\]

• In case of the rule for \( \text{nil} \), we needed the **premise** \( A : \text{Type} \) to guarantee that we can form the type \( \text{List}(A) \).

  – In case of the rule for \( \text{cons} \), this premise is **implicit** in the premise \( a : A \).
Example 2: Natural Numbers.

- The **natural numbers** $\mathbb{N}$ can be considered as being formed from two operations:
  - $0$,
  - $S$ where $S(n) = n + 1$.

- Using these two operations we can form $0$, $S(0) = 1$, $S(1) = 2$, and therefore all natural numbers.
  - So the **constructors** of $\mathbb{N}$ are $0$ and $S$.

- The **introduction rules** of $\mathbb{N}$ are:

\[
\begin{align*}
0 & : \mathbb{N} \\
\frac{n : \mathbb{N}}{S(n) : \mathbb{N}}
\end{align*}
\]
Constructors and Canonical Elements

- **Canonical elements** of a type are those introduced by an introduction rule.

- Canonical elements therefore always start with a *constructor*.

- **Examples:**
  - 0, \( S(2 + 3) \) in case of \( \mathbb{N} \).
    * Here 2 stands for \( S(S(0)) \) and 3 for \( S(S(S(0))) \).
  - nil, \( \text{cons}(1 + 1, \text{concat}(\text{cons}(0, \text{nil}), \text{nil})) \) in case of \( \text{List}(\mathbb{N}) \).
• Terms can usually be reduced further
  – Example:

\[
\text{concat} (\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil})) \rightarrow \text{cons}(2, \text{cons}(3, \text{nil}))
\]

  – \(\rightarrow\) stands for for “reduces to”.

• Further, reductions can be applied to subterms.
  – Example: Using the above reduction we obtain:

\[
\text{cons}(2, \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{cons}(2, \text{cons}(2, \text{cons}(3, \text{nil})))
\]
The reduction rules for addition on \( \mathbb{N} \) are:

1. \( n + 0 \rightarrow n \).
2. \( n + S(m) \rightarrow S(n + m) \).

These (and reductions using them in subterms) are the one-step reductions.

Reductions can be formed by a sequence of one-step-reductions.

Example:

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

is one reduction built from two one-step-reductions.
Canonical Elements and Reductions

- Canonical elements can only be reduced further by reducing the arguments of the constructor.
  - The constructor will always remain in place.
  - For instance $S(3 + 2)$ reduces to $S(S(S(S(S(0))))))$.
  - This reduction can be formed from one-step reductions:

$$
S(3 + 2) = S(S(S(0))) + S(0)
$$

$$
\rightarrow S(S(S(S(S(0)))) + S(0))
$$

$$
\rightarrow S(S(S(S(S(0)))) + 0)
$$

$$
\rightarrow S(S(S(S(S(0))))))
$$

More about this later.
Constructors and Canonical Elements (Cont.)

- The outermost $S$ always remains in place.
  - Once we have determined that we have the "the successor of something", this information will remain as it is when further reducing.
  - It cannot change later to 0.

- Similarly

\[
\text{cons}(1 + 1, \text{concat} (\text{cons}(3, \text{cons}(5, \text{nil})), \text{cons}(6, \text{nil}))) \rightarrow \text{cons}(2, \text{cons}(3, \text{cons}(5, \text{cons}(6, \text{nil})))
\]

- An element starting with cons will never reduce to one starting with nil.
  - Further, the arguments of the constructor reduce independent of each other.
• $2 + 3$ is a **non-canonical element**, and $( + )$ is **not a constructor**. (Note that $2 + 3$ is only a more readable way of writing $( + ) (2, 3)$):

  – $2 + 3 \rightarrow 5 = S(S(S(S(S(0))))).
  
  – When reducing $2 + 3$, the **outermost operator** $(+)$ changes to $S$.

• $\text{concat}(\text{cons}(2, \text{nil}), \text{nil})$ is a **non-canonical element**, and $\text{concat}$ is **not a constructor**:

  – $\text{concat}(\text{cons}(2, \text{nil}), \text{nil}) \rightarrow \text{cons}(2, \text{nil})$.
  
  – Again the **outermost operator** $\text{concat}$ changes to $\text{cons}$.

• **Any element of a type has to reduce to a canonical element**.
Constructors in Agda

- In Agda the constructor $C$ of type $A$ is written as $C@(A)$.
  - If $A$ can be inferred automatically, we can replace the above by $C@$.

- As type-constructors, in Agda constructors are curried: We have

\[
\begin{align*}
nil@(\text{List } N) & \quad :: \quad \text{List } N \\
\text{cons}@(&\text{List } N) & \quad :: \quad (n :: N) \\
 & \quad \rightarrow \quad (l :: \text{List } N) \\
 & \quad \rightarrow \quad \text{List } N
\end{align*}
\]
Since notations like \texttt{nil@(List N)} is usually to cumbersome, it is better to introduce abbreviations:

\[
\begin{align*}
\text{nil} & :: \text{List N} \\
& = \text{nil@} \\
\text{cons} & (n :: N) \\
& (l :: \text{List N}) \\
& :: \text{List N} \\
& = \text{cons@}_n l
\end{align*}
\]

Note that the above introduces \texttt{nil}, \texttt{cons} for \texttt{List N}, \textbf{general case} \texttt{List A} for any type \texttt{A}. (That would require \texttt{A : Type}).
• **Elimination rules** allow to take an element of a type and compute from it an element of another type.

• Example 1: First and second projection of a product:

\[
\begin{align*}
c : A \times B \\
\pi_0(c) : A \\
\pi_1(c) : B
\end{align*}
\]

– Equality rules will express \( \pi_0(\langle a, b \rangle) = a \), \( \pi_1(\langle a, b \rangle) = b \).
Example 2: Addition in $\mathbb{N}$

\[
\frac{n : \mathbb{N} \quad m : \mathbb{N}}{n + m : \mathbb{N}}
\]

- Equality rules will express
  * $n + 0 = n$.
  * $n + S(m) = S(n + m)$.
- Proceeding like this would require one elimination rule from $\mathbb{N}$ we want to define.
- Instead we will introduce one general elimination rule to introduce all functions we expect to be definable, including recursive ones.
Elimination Rules Inverting the Introduction Rules

- Elimination rules invert the introduction rules.

- In case of $A \times B$, the canonical elements are of the form $a : A, b : B$.

- A non-canonical element of type $A \times B$ must reduce to a canonical element.

- Therefore, if $c : A \times B$, the canonical form of $\pi_0(c)$ can be computed as follows;
  
  – Reduce $c$ to a canonical element.
  – This element must be of the form $\langle a, b \rangle$.
  – Reduce $a$ (which is of type $A$) to its canonical form.
• Elimination for built-in types has special notation.

• For user defined types, elimination is realized by **case distinction**.

• Example: Definition of addition in \( \mathbb{N} \):

\[
(+) \quad (n, m :: \mathbb{N}) \\
:: \mathbb{N} \\
= \text{case } m \text{ of} \\
\hspace{1cm} (Z) \rightarrow n; \\
\hspace{1cm} (S m') \rightarrow S (n + m'); \\
\}
\]
• The canonical element for an element, which is the result of an elimination,
can be always computed as follows:
  – Reduce the element to be eliminated to canonical form.
  – Then make one reduction step (Red).
  – The result will be a canonical or non-canonical element of the target type.
    Reduce it to canonical form.

• For instance in case of $A \times B$, (Red) are the reductions
  – $\pi_0(\langle a, b \rangle) \rightarrow a$.
  – $\pi_1(\langle a, b \rangle) \rightarrow b$. 
• In case of (+), (Red) are the reductions
  
  – \( n + 0 \rightarrow n. \)
  
  – \( n + S(m) \rightarrow S(n + m). \)
  
  – Note that the second argument is the argument we are “eliminating”.

• So the computation of \( 0 + (1 + 1) \) is as follows:

\[
0 + (1 + 1) = 0 + (S(0) + S(0)) \rightarrow 0 + S(S(0) + 0) \rightarrow \]

The result is already in canonical form.
Equality rules express (Red) type theoretically.

- They describe what happens, if one first introduces and immediately eliminates it.
Example (Equality Rule)

- Equality rules for $A \times B$:

$$
\frac{a : A \quad b : B}{\pi_0(\langle a, b \rangle) = a : A}
$$

- In the first judgment we can derive $\pi_0(\langle a, b \rangle) : A$ as follows:

$$
\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B}
\frac{\langle a, b \rangle : A \times B}{\pi_0(\langle a, b \rangle) : A}
$$

- So it is derived by first introducing $\langle a, b \rangle$ and then eliminating it.

- The equality rule explains how to reduce that element (namely to $a : A$).
• The second equality rule for $\times$ is similar:

$$\frac{a : A}{\pi_1(\langle a, b \rangle)} = b : B$$
Example 2 (Equality Rule)

- The first equality rule for $+$ is as follows:

\[
\frac{n : \mathbb{N}}{n + 0 = n : \mathbb{N}}
\]

- $n + 0 : \mathbb{N}$ can be derived by first introducing $0 : \mathbb{N}$ and then eliminating it using $+$.
  - (The right side is an axiom, the left side has to be concluded using some derivation.)

\[
\frac{n : \mathbb{N} \quad 0 : \mathbb{N}}{n + 0 : \mathbb{N}}
\]

- The equality rule explain how to reduce $n + 0$. 
Example 3 (Equality Rule)

- The second equality rule for $+$ is as follows:

$$n : N \quad m : N$$

$$n + S(m) = S(n + m) : N$$

- $n + S(m) : N$ can be derived by first introducing $S(m)$, eliminating it using $+$:

$$n : N \quad S(m) : N$$

$$n + S(m) : N$$
Equality Rules in Agda

- Equality Rules in Agda are **implicit**.

- The notation for elimination however indicates already how the reductions take place.

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

- Functions corresponding to elimination are defined by telling how elimination operates.

Equality Versions of Formation-, Introduction- and Elimination Rules

- These express: if we replace the terms in the premises by equal ones, we obtain equal results.

- Example: Equality version of the formation rule for List:

\[
\frac{A = B : Type}{\text{List}(A) = \text{List}(B)}
\]

- Example: Equality version of the formation rule for N (degenerated):

\[
N = N : Type
\]
Example: Equality version of the introduction rules for $\text{List}$ (rule for $\text{nil}$ is degenerated):

$$
\begin{align*}
A &: \text{Type} \\
\text{nil} &= \text{nil} : \text{List}(A)
\end{align*}
$$

$$
\begin{align*}
a = a' &: A \\
l = l' &: \text{List}(A)
\end{align*}
$$

$$
\begin{align*}
\text{cons}(a, l) &= \text{cons}(a', l') : \text{List}(A)
\end{align*}
$$

Example: Equality version of the elimination rule for $(+,\text{N})$,

$$
\begin{align*}
n = n' &: \text{N} \\
m = m' &: \text{N}
\end{align*}
$$

$$
\begin{align*}
n + m &= n' + m' : \text{N}
\end{align*}
$$
Equality Versions of Rules (Cont.)

- The equality versions of the rules in questions can be formed in a straightforward way, once one knows the non-equality version.
  - We will often not mention them.

- In Agda they are implicit (part of the reduction machinery).
(b) The Non-Dependent Function Type and

Rules of the Non-Dependent Product

**Formation Rule**

\[
\frac{A : \text{Type} \quad B : \text{Type}}{A \times B : \text{Type}}
\]

**Introduction Rule**

\[
\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B}
\]

**Elimination Rules**

\[
\begin{align*}
c : A \times B \\
\pi_0(c) : A \\
\pi_1(c) : B
\end{align*}
\]

**Equality Rules**

\[
\begin{align*}
a : A \quad b : B \\
\pi_0(\langle a, b \rangle) = a : A \\
\pi_1(\langle a, b \rangle) = b : B
\end{align*}
\]
The $\eta$-Rule

This rule does not fit into the above schema:

$$c : A \times B \quad \Rightarrow \quad c = \langle \pi_0(c), \pi_1(c) \rangle : A \times B$$

($\eta = \text{greek letter spelled "eta"}$)
• The $\eta$-rule expresses that any element of $A \times B$ is of the form $\langle \text{something}_0, \text{something}_1 \rangle$:

  - If $a : A \times B$ and we have the $\eta$-rule, then this follows with $\text{something}_0 = \pi_0(c)$, $\text{something}_1 = \pi_1(c)$.
  - On the other hand, if we have $c$ of this form, e.g. $c = \langle c_0, c_1 \rangle$ then we get

$$\langle \pi_0(c), \pi_1(c) \rangle = \langle \pi_0(\langle c_0, c_1 \rangle), \pi_1(\langle c_0, c_1 \rangle) \rangle = \langle c_0, c_1 \rangle = c$$

so the conclusion of the $\eta$-rule can be derived without using the $\eta$-rule.
The $\eta$-Rule (Cont.)

- For elements of $A \times B$ introduced by an introduction rule, we don’t need the $\eta$-rule.

- However, if we assume an element of type $A \times B$, e.g. state

  \[
  x : A \times B \Rightarrow x : A \times B
  \]

  (so $x$ is just a variable), we cannot derive that $x = \langle \pi_0(x), \pi_1(x) \rangle$ without making use of the $\eta$-rule.
**Equality Versions of the Rules**

**Equality Version of the Formation Rule**

\[
A = A' : \text{Type} \quad B = B' : \text{Type} \\
A \times B = A' \times B' : \text{Type}
\]

**Equality Version of the Introduction Rule**

\[
a = a' : A \quad b = b' : B \\
\langle a, b \rangle = \langle a', b' \rangle : A \times B
\]

**Equality Versions of the Elimination Rules**

\[
c = c' : A \times B \\
\pi_0(c) = \pi_0(c') : A \\
\pi_1(c) = \pi_1(c') : B
\]
Rules of the Non-Dependent Function Type

**Formation Rule**

\[
\begin{array}{c}
A : \text{Type} \\
B : \text{Type}
\end{array}
\frac{}{A \rightarrow B : \text{Type}}
\]

**Introduction Rule**

\[
\begin{array}{c}
x : A \Rightarrow b : B
\end{array}
\frac{}{\lambda(x : A).b : A \rightarrow B}
\]

**Elimination Rule**

\[
\begin{array}{c}
f : A \rightarrow B \\
a : A
\end{array}
\frac{}{f \ a : B}
\]

**Equality Rule**

\[
\begin{array}{c}
x : A \Rightarrow b : B \\
a : A
\end{array}
\frac{}{(\lambda(x : A).b) \ a = b[x := a] : B}
\]

Here \(b[x := a]\) is the result of substituting in \(b\) every occurrence of variable \(x\) by the term \(a\) (after some renaming of bounded variables).
The reduction corresponding to the equality rule is often called $\beta$-reduction.

- $\beta = \text{greek letter spelled "beta".}$
- As a reduction, it reads:

$$(\lambda(x : A).b) \ a \rightarrow b[x := a]$$
The $\eta$-Rule

Again this rule does not fit into the above schema:

\[
\frac{f : A \rightarrow B}{f = \lambda(x : A).f x : A \rightarrow B}
\]
The $\eta$-Rule (Cont.)

- The $\eta$-rule expresses that any element of $A \rightarrow B$ is of the form $\lambda(x : A).:\text{something}$:
  
  - If $f : A \rightarrow B$ and we have the $\eta$-rule, then this follows with $\text{something} = fx$.
  - On the other hand, if we have $f$ is of this form, e.g. $f = \lambda(x : A)\cdot t$, we get

\[
\begin{align*}
\lambda(x : A)\cdot f x &= \lambda(x : A)\cdot ((\lambda(x : A)\cdot t)x) \\
&= \lambda(x : A)\cdot t[x := x] \\
&= \lambda(x : A)\cdot t \\
&= f
\end{align*}
\]

so the conclusion of the $\eta$-rule can be derived without using the $\eta$-rule.
The $\eta$-Rule (Cont.)

- For elements of $A \rightarrow B$ introduced by an introduction rule, we don't need the $\eta$-rule.

- However, if we assume an element of type $A \rightarrow B$, e.g. state $f : A \rightarrow B \Rightarrow f : A \rightarrow B$

  (so $f$ is just a variable), we cannot derive that $f = \lambda(x : A).fx$ without making use of the $\eta$-rule.
Equality Versions of the Rules

Equality Version of the Formation Rule

\[
A = A' : \text{Type} \quad B = B' : \text{Type} \\
\overline{A \to B = A' \to B' : \text{Type}}
\]

Equality Version of the Introduction Rule

\[
x : A \Rightarrow b = b' : B \\
\overline{\lambda(x : A).b = \lambda(x : A).b' : A \to B}
\]

Equality Version of the Elimination Rule

\[
f = f' : A \to B \quad a = a' : A \\
\overline{f \ a = f' \ a' : B}
\]
(c) The Dependent Function Type and Product

Rules of the Dependent Product

**Formation Rule**

\[ A : \text{Type} \quad x : A \Rightarrow B : \text{Type} \]

\[ (x : A) \times B : \text{Type} \]

**Introduction Rule**

\[ x : A \Rightarrow B : \text{Type} \quad a : A \quad b : B[x := a] \]

\[ \langle a, b \rangle : (x : A) \times B \]

The introduction rule requires an extra premise \( x : A \Rightarrow B : \text{Type} \), which is not implied by the other premises.

• In the last introduction rule, an extra premise $x : A$ was required.

  – This is required in order to guarantee that we can form $(x : A) \times B$.

  – In case of the non-dependent product, this premise was not necessary: $a : A$ and $b : B$ indirectly implies that we know $A : \text{Type}$ and $B : \text{Type}$ from which it follows $A \times B : \text{Type}$.


Elimination Rules

\[
\begin{align*}
c : (x : A) \times B & \quad \frac{c : (x : A) \times B}{\pi_0(c) : A} \\
c : (x : A) \times B & \quad \frac{c : (x : A) \times B}{\pi_1(c) : B[x := \pi_0(c)]}
\end{align*}
\]

Equality Rules

\[
\begin{align*}
x : A \Rightarrow B : \text{Type} & \quad a : A \quad b : B [x := a] \\
& \quad \frac{\pi_0(\langle a, b \rangle) = a : A}{\pi_0(\langle a, b \rangle) = a : A}
\end{align*}
\]

\[
\begin{align*}
x : A \Rightarrow B : \text{Type} & \quad a : A \quad b : B [x := a] \\
& \quad \frac{\pi_1(\langle a, b \rangle) = b : B [x := a]}{\pi_1(\langle a, b \rangle) = b : B [x := a]}
\end{align*}
\]

Note that the last two rules require the extra premise \( x : A =\) (which is not implied by the premises).

We have the following $\eta$-rule:

$$c : (x : A) \times B$$

$$c = \langle \pi_0(c), \pi_1(c) \rangle : (x : A) \times C$$

- Again the $\eta$-rule expresses that every element of $(x : A) \times B$ is of the form $\langle \text{something}_0, \text{something}_1 \rangle$.

- Again the $\eta$-rule cannot be derived if the element in question is a variable.
Equality Versions of the above Rules

Equality Version of the Formation Rule

\[ A = A' : \text{Type} \quad x : A \Rightarrow B = B' : \text{Type} \]
\[ (x : A) \times B = (x : A') \times B' : \text{Type} \]

Equality Version of the Introduction Rule

\[ x : A \Rightarrow B : \text{Type} \quad a = a' : A \quad b = b' : B \]
\[ \langle a, b \rangle = \langle a', b' \rangle : (x : A) \times B \]

Equality Versions of the Elimination Rules

\[ c = c' : (x : A) \times B \]
\[ \pi_0(c) = \pi_0(c') : A \]
\[ \pi_1(c) = \pi_1(c') : B[x := \pi_0(c)] \]
The Non-Dependent Product as an Abbreviation

• The non-dependent product \( A \times B \) can now be seen as an abbreviation for \( (x : A) \times B \) for some fresh variable \( x \).

• Taking \( A \times B \) as an abbreviation, we can see later that the rules for the non-dependent product are special cases of the rules for the dependent product.
The Non-Dependent Product as an Abbreviation

- More precisely we will see:
  - From $A : \text{Type}$ and $B : \text{Type}$ we can derive $x : A \Rightarrow B$.
    * This requires the weakening rule, which will be introduced later.
    Therefore the premises of the formation rule for the non-dependent product imply those of the formation rule for the dependent product.
  - From a derivation of $a : A$ we can derive $A : \text{Type}$ (we need the concept of presupposition for that).
    * Therefore the premises of the introduction rule for the non-dependent product imply those of the dependent product.
    * Similarly for the elimination, equality and $\eta$-rule.
• In Agda, we have the dependent record type.
  – It is essentially a “labelled product”.

• Assume we have introduced already $A :: \text{Type}$, $a :: A \Rightarrow B$.
  Then we can introduce

\[
D :: \text{Type} \\
= \text{sig}\{a :: A; b :: B\}
\]
• If we have \( a' :: A, b' :: B[a := a'] \), then we can introduce

\[
c:: D = \text{struct}\{a = a'; b = b'\} :: D
\]

• One can introduce longer records as well, e.g.

\[
\text{sig}\{a :: A; b :: B; c :: C; e :: E\}
\]
The Dependent Product in Agda (Cont.)

- We can now project any element $d :: D$ as above down to $A$ and $B$:

  $d.a :: A$
  
  $d.b :: B[a := d.a]$

- If $c = \text{struct}\{a = a'; b = b'\}$, then we have:

  $c.a$ is equal to $a'$

  $c.b$ is equal to $b'$
The Dependent Product in Agda (Cont.)

- Unfortunately, the dependent product does not behave very well.
  - This is due to the fact that Agda doesn’t support the $\eta$-rule.
  - In this setting $\eta$-equality asserts that if
    \[
    c :: \text{sig}\{a :: A; b :: B(a)\}
    \]
    then
    \[
    c = \text{struct}\{a = c.a; b = c.b\}
    \]

- In most cases one can avoid this, by using the inductively defined $\Sigma$-type, which will be treated later.
Rules of the Dependent Function Type

Formation Rule

\[ A : \text{Type} \quad x : A \Rightarrow B : \text{Type} \]
\[ (x : A) \rightarrow B : \text{Type} \]

Introduction Rule

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

Elimination Rule

\[ f : (x : A) \rightarrow B \quad a : A \]
\[ f \ a : B[x := a] \]

Equality Rule

\[ y : A \Rightarrow b : B \quad a : A \]
\[ (\lambda(x : A).b) \ a = b[x := a] : B[x := a] \]
The $\eta$-rule has a special status:

\[ f : (x : A) \rightarrow B \]
\[ \frac{f = \lambda(x : A).f x}{f x : (x : A) \rightarrow B} \]

- Again the $\eta$-rule expresses that every element of $(x : A) \rightarrow B$ is of the form $\lambda(x : A).something$.

- Again the $\eta$-rule cannot be derived if the element in question is a variable.
Further terms which differ in the choice of bounded variables are identified:

- E.g. $\lambda(x : A).x$ and $\lambda(y : A).y$ are identified.
- E.g. $\lambda(x : N).x + x$ and $\lambda(y : N).y + y$ are identified.
- A similar rule applies to bounded variables in types.
- Called $\alpha$-equivalence ($\alpha = \text{greek letter spelled alpha}$).
Equality Versions of the above Rules

Equality Version of the Formation Rule

\[ A = A' : \text{Type} \quad x : A \Rightarrow B = B' : \text{Type} \]
\[
(x : A) \rightarrow B = (x : A') \rightarrow B' : \text{Type}
\]

Equality Version of the Introduction Rule

\[ x : A \Rightarrow b = b' : B \]
\[ \lambda(x : A).b = \lambda(x : A).b' : (x : A) \rightarrow B \]

Equality Version of the Elimination Rule

\[ f = f' : (x : A) \rightarrow B \quad a = a' : A \]
\[ f a = f' a' : B[x := a] \]
The non-dependent function type

\[ A \rightarrow B \]

is a special case of the dependent function type

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

where \( B \) does not depend on \( x \).
In Agda one writes \((x::A) \rightarrow C\) for the \textbf{dependent function type} \(A \rightarrow C\) for the \textbf{nondependent function type}.

We write on our slides \(\rightarrow\) instead of \(-\rightarrow\).

There are \textbf{two ways of introducing an element of} \((x::A) \rightarrow C\):

- We can write

\[
f \ (x::A) :: C = \ldots
\]

\* Requires the \(\ldots\) is an element of type \(C\), possibly making use of \(x\).

\* The above introduces

\[
f :: (x::A) \rightarrow C
\]
• Alternatively, one can use the $\lambda$-notation:
  
  – Remember that $\backslash$ is used instead of $\lambda$ in Agda.
    * In our slides we will use $\lambda$.
  – The above can be rewritten as
    
    \[
    f :: (x :: A) \to C = \lambda (x :: A) \to \ldots
    \]
The example is better introduced using the first notation.

However, $\lambda$-notation allows to introduce anonymous 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:

$$\text{map} \ (\lambda(x::N) \to S \ x) \ (\text{cons}\ \text{two} \ (\text{cons}\ \text{three} \ \text{nil}))$$

The result would be

$$\text{cons}\ \text{three} \ (\text{cons}\ \text{four} \ \text{nil})$$
• We can write

\[(n,m::\mathbb{N}) \rightarrow A(n,m)\]

instead of

\[(n::\mathbb{N}) \rightarrow (m::\mathbb{N}) \rightarrow A(n,m)\]

• Similarly we can write

\[\lambda(n,m::\mathbb{N}) \rightarrow \cdots\]

instead of

\[\lambda(n::\mathbb{N}) \rightarrow \lambda(m::\mathbb{N}) \rightarrow \cdots\]
• Similarly we can write

\[
f \ (n,m::N) \\
:: \ N \\
= \ldots
\]

instead of

\[
f \ (n::N) \\
(m::N) \\
:: \ N \\
= \ldots
\]
• **Application** has the same syntax as in the rules above:
  If we have
  \[
  f :: (x :: A) \rightarrow B ,
  \]
  \[
  a :: A ,
  \]
  then we can conclude:
  \[
  fa :: B[x := a]
  \]

• And we have that
  \[
  (\lambda(x :: A) \rightarrow b) a
  \]
  and
  \[
  b[x := a]
  \]
  are identified.
In Agda syntax, the \( \eta \)-rule would state that if

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

then

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

- \( \eta \)-rule is \textit{computationally expensive} and therefore not implemented.
- The lack of the \( \eta \)-rule \textit{causes sometimes problems}.
Example of the Use of Dependent Products

• Let $G$ be the set of genders,

$$G = \{\text{male}, \text{female}\}.$$ 

• Let for $g : G$ the type $\text{Name}s(g)$ be the collection of names of that gender, e.g.

- $\text{Name}s(\text{male}) = \{\text{Tom, Jim}\},$
- $\text{Name}s(\text{female}) = \{\text{Jill, Sara}\}.$
Example of the Use of Dependent Products

- Now the **set of names** is the set of pairs \( \langle g, n \rangle \) s.t. \( g : \text{Gender} \) and \( n : \text{Name}(g) \).

- This type is written as

\[
(g : G) \times \text{Names}(g)
\]
The “Names”-Example in Agda

• Although we haven't introduced yet a notation for algebraic data types, the following is readable for those familiar with Haskell:

\[
G :: \text{Type} \\
= \text{data male | female}
\]

\[
\text{Names} \ (g :: G) :: \text{Type} \\
= \text{case } g \text{ of } \{ \\
\quad \text{(male)} \rightarrow \text{data Tom | Jim}; \\
\quad \text{(female)} \rightarrow \text{data Jill | Sara}; \\
\}
\]

\[
\text{AllNames} :: \text{Type} \\
= \text{sig}\{ \ g :: G; \\
\quad n :: \text{Names } g \}. \\
\]
Example of the Dependent Function Type

• Define

\[
\text{select} : (g : G) \rightarrow \text{Names}(g)
\]

\[
\text{select(male)} = \text{Tom}
\]

\[
\text{select(female)} = \text{Jill}
\]

• select selects for every gender a name.

• select male will be an element of \( B(\text{male}) = B(x)[x := \text{male}] \).

• It wouldn’t make sense to say select male : \( B(x) \).
As before, here is the code for the select example, which should be readable for those familiar with Haskell:

\[
\begin{align*}
G & :: \text{Type} = \text{data male} \mid \text{female} \\
\text{Names} \ (g :: G) & :: \text{Type} \\
& = \text{case } g \text{ of } \\
& \quad (\text{male}) \to \text{data Tom} \mid \text{Jim}; \\
& \quad (\text{female}) \to \text{data Jill} \mid \text{Sara}; \\
\text{select} & :: (g :: G) \to \text{Names } g \\
& = \lambda(g :: G) \to \text{case } g \text{ of } \\
& \quad (\text{male}) \to \text{Tom@} \\
& \quad (\text{female}) \to \text{Jill@}
\end{align*}
\]
Common Contexts

- The convention is that all rules can as well be weakened by a common context.
- This means that when introducing a rule
  \[
  \frac{\Gamma_1 \Rightarrow \theta_1 \quad \cdots \quad \Gamma_n \Rightarrow \theta_n}{\Gamma \Rightarrow \theta}
  \]
  we implicitly introduce as well the following rules (for any choice of \(n, x_n, A_n\)):
  \[
  \frac{x_1 : A_1, \ldots, x_n : A_n, \Gamma_1 \Rightarrow \theta_1 \quad \cdots \quad x_1 : A_1, \ldots, x_n}{x_1 : A_1, \ldots, x_n : A_n, \Gamma \Rightarrow \theta}
  \]
Example

- For instance, the formation rule of $\times$:

\[
A : \text{Type} \quad x : A \Rightarrow B : \text{Type} \\
\frac{}{(x : A) \times B : \text{Type}}
\]

can be weakened as follows:

\[
x_1 : A_1, \ldots, x_n : A_n \Rightarrow A : \text{Type} \quad x_1 : A_1, \ldots, x_n : A_n, x : A \\
\frac{}{x_1 : A_1, \ldots, x_n : A_n \Rightarrow (x : A) \times B : \text{Type}}
\]
Example (Cont.)

- Consider the sample derivation (assuming $A : \text{Type}$):

  $\frac{x : A, y : A \Rightarrow y : A}{x : A \Rightarrow \lambda(y : A) \rightarrow y : A \rightarrow A}$

  $\frac{x : A \Rightarrow \lambda(y : A) \rightarrow y : A \rightarrow A}{\lambda(x : A) \rightarrow \lambda(y : A) \rightarrow y : A \rightarrow A}$

- The first rule used is the rule for $\lambda$-introduction, weakened by the context $x : A$.

- The second rule used is the rule for $\lambda$-introduction without any weakening.
The only side condition is that, if the rule introduces a new variable, it must not occur in the context.

For instance in the weakend form of the $\eta$-rule:

$$
\frac{x_1 : A_1, \ldots, x_n : A_n \Rightarrow f : A \rightarrow B}{x_1 : A_1, \ldots, x_n : A_n \Rightarrow f = \lambda(x : A).fx : A \rightarrow B}
$$

$x$ must be different from $x_1, \ldots, x_n$.

- This is since in this lecture we want that variables bound in any context are different.
  
  $f\,x$ above is implicitly in the context $x_1 : A_1, \ldots, x_n : A_n$.  

Weakening of Axioms

- If we have an axiom, we need to be sure that the context, we weakened with, is well-formed.

- This requires the context judgment $x_1 : A_1, \ldots, x_n : A_n \Rightarrow \text{Context}$
  - Will be discussed later.
Weakening of Axioms

• For the moment we mention how the formation rule for $N : \text{Type}$ can be weakened:

$$ x_1 : A_1, \ldots , x_n : A_n \Rightarrow \text{Context} \Rightarrow x_1 : A_1, \ldots , x_n : A_n \Rightarrow N : \text{Type} $$

• More about this later.
Let expressions in Agda

- Agda allows to introduce temporary variables, using “let-expressions”.
- 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 constants $a_1, \ldots, a_n$ of types $A_1, \ldots, A_n$ as $s_1, \ldots, s_n$ respectively, and can then use them.

- $s_i$ can refer to $a_i$ (might be result in non-termination; this will be discussed below).
Let expressions in Agda (Cont.)

- If we are in a goal, we can use the command `agda-let (Make let expression)`.
  - We have to write down the variables, separated by a blank.
  - Agda will construct a template of the form:

    ```latex
    \text{let } a_1 \ :: \ \{! \ !\} \\
    \hspace{1cm} = \{! \ !\} \\
    a_2 \ :: \ \{! \ !\} \\
    \hspace{2cm} = \{! \ !\} \\
    \hspace{3cm} \vdots \\
    a_n \ :: \ \{! \ !\} \\
    \hspace{4cm} = \{! \ !\} \\
    \text{in } \{! \ !\}
    ```

Example of Let expressions

Here follows a simple concrete example, which computes \((n + n) \times (n + n)\) for natural numbers \(n, m :: \mathbb{N}\).

\[
\begin{align*}
f \ (n :: \mathbb{N}) \\
& :: \mathbb{N} \\
& = \text{let } m :: \mathbb{N} \\
& \quad = n + n \\
& \quad \text{in } m \times m
\end{align*}
\]
Derivations and the Corresponding Agda Code

- In this subsection we look at the relationship between Agda code and the corresponding derivations.
  - We consider various examples.
    - First we will go through the development of the Agda code.
    - Then we will look at, how the corresponding derivations are developed, following each step in the development of the Agda code.
Example 1

- We want to derive in Agda

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

(See example file `exampleIdentity.agda`)

- **Step 1:**
  - We need to introduce the type \( A \) first.
  - Since we want to have the definition for an arbitrary type \( A \), we postulate (i.e., assume) one type \( A \):

\[ \text{postulate } A :: \text{Type} \]
Example 1 (Cont.)

- **Step 2:** We state our goal:

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

- Agda is an intensional-sensitive language. The complete definition of \( f \) must be intended otherwise it will be regarded as a new definition.
Example 1 (Cont.)

- **Step 3:**
  - We want to derive an element of function type $A \rightarrow A$.
  - Elements of the function type $A \rightarrow A$ are introduced by using $\lambda$-terms.
  - If introduced as a $\lambda$-term, the term in question will be of the form $\lambda(a :: A) \rightarrow \text{something}$.
  - Agda has a command `agda-intro (Intro)` which does this step automatically.
    * Has to be executed while the cursor is inside one goal.
  - After executing it we get:
    
    $$f :: A \rightarrow A = \lambda(h :: \{! !\}) \rightarrow \{! !\}$$

    (The precise Agda code uses \ instead of $\lambda$, and $\rightarrow$ instead of $\rightarrow$.)
Example 1 (Cont.)

- **Step 4:**
  - The first goal, the type of the variable \( h \) can be solved automatically.
  - Use `agda-solve` (Solve)
  - We obtain:
    \[
    f :: A \rightarrow A \\
    = \lambda (h :: A) \rightarrow \{! !\}
    \]
Example 1 (Cont.)

- **Step 4 (Cont)**
  - It is a good idea to rename the variable to something, for instance to a. This can be done by simple editing.
  - We can always edit the current code.
  - If one wants to edit parts involving goals, one first has to execute:
    \[
    \text{agda-restart ( (Re)Start Agda)}
    \]
    Then one is in a mode where the goals are converted to ordinary symbols and can edit everything.
  - At the end of any editing one should execute:
    \[
    \text{agda-load-buffer (Load Buffer)}
    \]
    Otherwise the changes will not be known by Agda.
  - We obtain:
    \[
    f :: A \rightarrow A = \lambda (a :: A) \rightarrow \{! !\}
    \]
Example 1 (Cont.)

- **Step 5:**

  - In order for \( \lambda(a :: A) \rightarrow \{! !\} \) to be of type \( A \rightarrow A \), \( \{! !\} \) must be of type \( A \).
  - Then this \( \lambda \)-term computes an element of type \( A \) depending on some \( a \) of type \( A \), which means it is a function of type \( A \rightarrow A \).
  - So the type of the goal is \( A \).
  - This can be inspected by using the menu `agda-goalType-of-meta-reduced` (Type of goal (unfolded)), which shows the type of the current goal.
  - Has to be executed while the cursor is inside one goal.
  - It shows \( A \).

• **Step 5 (Cont.)**

  - We can inspect the context.
  - The context contains everything we can use when solving our goal. It contains:
    * $A :: \text{Type}$.
    * $f :: A \rightarrow A$.
      
      See next slide.
    * $a :: A$.
      
      Since we are defining $a$ an element of type $A$ depending on $a :: A$, we can use $a$. 

Termination Check

- On the last slide we had \( f :: A \rightarrow A \) in the context.

- This appears, since the type checker allows to define functions recursively, independently of whether the recursion terminates or not.

- For the type checker a definition \( b :: A = b \) would be legal, although evaluating \( b \) doesn’t terminate (black hole recursion).
• Agda has a command `agda-term-check-buffer` (Check Termination), which checks whether recursive definitions are done properly.

• One should use this command at the end of a session to avoid black hole recursion.

• If the termination check succeeds, all programs checked will terminate.

• If the termination check fail, it might still be the case that all programs terminate.
  (One cannot write a universal termination checker, since the problem is undecidable).
Example 1 (Cont.)

- **Step 5 (Cont.)**
  - Now everything with *result type* $A$ (i.e. which has at the arrow $A$) can be used in order to solve the goal.
    - $f$ would result in black-hole recursion.
    - So we take $a$.
  - We type in $a$ into the goal and then use the command `agda-refine (Refine)`
  - We obtain:
    $$f :: A \rightarrow A = \lambda(a :: A) \rightarrow a$$
    and are done.
Example 1, Using Rules

- In **Agda step 1** we postulated $A :: \text{Type}$. This corresponds in the rule system, that we can assume $A : \text{Type}$, i.e. can write this down without any derivation.

- In **Agda step 2** we stated our goal:

\[ f :: A \rightarrow A = \{! \, !\} \]

In terms of rules this means that we want to derive something of type $A \rightarrow A$. We write for this something $d_0$ and get as conclusion of our derivation:

\[ d_0 : A \rightarrow A \]
Example 1, Using Rules (Cont.)

- In Agda step 3 and 4 we replaced \{! !\} by \(\lambda (a :: A) \rightarrow\):

\[
f :: A \rightarrow A \\
= \lambda (a :: A) \rightarrow \{! !\}
\]

In terms of rules this means that we replace \(d_0\) by \(\lambda (a)\).

\[
a : A \Rightarrow d_1 : A \\
\frac{\lambda (a : A) . d_1 : A \rightarrow A}
\]

Example 1, Using Rules (Cont.)

- In Agda step 5 we replaced \( \{! !\} \) in \( \lambda(a :: A) \rightarrow \{! !\} \) by

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

In terms of rules this means that we replace \( d_1 \) by \( a \).

\( a : A \Rightarrow a : A \) follows by an assumption rule:

\[
\frac{a : A \Rightarrow a : A}{\lambda(a : A).a : A \rightarrow A}
\]

- The assumption rule will be discussed later.
  - Essentially it allows to derive if \( x : B \) occurs in the context it holds.
Example 2

- We consider a derivation of

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

(See example [exampleSampleDerivation2.agda](exampleSampleDerivation2.agda)).

- **Step 1:**
  - We postulate \( A \):
    \[
    \text{postulate } A :: \text{Type}
    \]
  - We state our goal:
    \[
    f :: ((A \rightarrow A) \rightarrow A) \rightarrow A
    
    = \{! !\}
    \]
Example 2 (Cont.)

- **Step 2:**
  - The type of the goal is a function type, and we can use `agda-intro (Intro)`.
  - We obtain
    \[
    f:: ((A \to A) \to A) \to A
    = \lambda(h :: \{! !\}) \to \{! !\}
    \]
  - Using `agda-solve (Solve)` we obtain:
    \[
    f:: ((A \to A) \to A) \to A
    = \lambda(h :: (A \to A) \to A) \to \{! !\}
    \]
• **Step 2 (Cont.):**
  
  – We rename the variable $h$ to $x$ and use `agda-load-buffer` so that Agda realizes this change:

$$f :: ((A \rightarrow A) \rightarrow A) \rightarrow A$$

$$= \lambda(x :: (A \rightarrow A) \rightarrow A) \rightarrow \{! !\}$$
• **Step 3:**

- The type of the new goal is $A$, which is the result type we are defining.
- The context contains $f$ (for recursive definitions), $A$ and $x$.
- $x$ is a function of result type $A$. Applying it to its argument would result the type of the goal in question.
- Therefore we type into the goal $x$ and use `agda-refine`.
  
  * Agda will then apply $x$ to as many goals as needed in order to obtain an element of the desired type.
  * In our case it is one (of type $A \to A$).

We obtain

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

$$= \lambda (x :: (A \to A) \to A) \to x \{! !\}$$
Example 2 (Cont.)

- Step 4:
  - The type of the new goal is $A \to A$.
    * This is since $x :: (A \to A) \to A$ needs to be applied to an element of type $A \to A$ in order to obtain an element of type $A$.
    * We try \texttt{agda-intro (Intro)} and obtain:
      
      \[
      f :: ((A \to A) \to A) \to A \\
      = \lambda (x :: (A \to A) \to A) \to x (\lambda (h :: \{! !\}))
      \]

Example 2 (Cont.)

• **Step 4 (Cont)**

  * Using *agda-solve* (*Solve*) we obtain:

    \[
    f :: ((A \rightarrow A) \rightarrow A) \rightarrow A \\
    = \lambda(x :: (A \rightarrow A) \rightarrow A) \rightarrow x(\lambda(h :: A) \rightarrow A)
    \]

    We wanted to define an element of \(A \rightarrow A\), so the domain will be \(A\).
Example 2 (Cont.)

**Step 4 (Cont.)**

- We rename $h$ by $a$, reload the buffer, and obtain:

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

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

Step 5

- The new goal has type $A$.
  * The complete expression $\lambda (a :: A) \rightarrow \{! !\}$ should have type $A$, so $\{! !\}$ must have type $A$.
- The context contains $A :: \text{Type}$, $f$, $x$ and $a$. We can use both $x$ and $a$ here.
  * There is usually more than one solution for proceeding.
    This means that we sometimes have to backtrack and try a different solution.
- We try $a :: A$. After inserting it and using \texttt{agda-refine (Refine)} we obtain the following and are done.

\[
\begin{align*}
f &: ((A \rightarrow A) \rightarrow A) \rightarrow A \\
  &= \lambda (x :: (A \rightarrow A) \rightarrow A) \rightarrow x (\lambda (a :: A) \rightarrow a)
\end{align*}
\]
Example 2, Using Rules

- Postulating $A :: \text{Type}$ corresponds to assuming $A : \text{Type}$ in the rules without deriving it.

- Stating the goal means that we have as last line of the derivation:

\[
d_0 : ((A \to A) \to A) \to A
\]
Example 2, Using Rules

• The next step in the Agda-derivation was to replace the goal
  \( \lambda(x :: (A \rightarrow A) \rightarrow A) \rightarrow \{! !\} \).

• This corresponds to replacing \( d_0 \) by \( \lambda(x : (A \rightarrow A) \rightarrow A) \).
  In the last step an introduction rule:

\[
\begin{align*}
  x : (A \rightarrow A) \rightarrow A & \Rightarrow d_1 : A \\
  \lambda(x : (A \rightarrow A) \rightarrow A).d_1 : ((A \rightarrow A) \rightarrow A) & \Rightarrow d_2 : ((A \rightarrow A) \rightarrow A)
\end{align*}
\]
Example 2, Using Rules

- The next step in the Agda-derivation used refine. \{! !\} was replaced by \(x \{! !\}\).

- This corresponds to replacing \(d_1\) by \(xd_2\), and using one elimination rule in order to derive it:

\[
\begin{align*}
x : (A \to A) \to A &\Rightarrow x : (A \to A) \to A \\
x : (A \to A) \Rightarrow x d_2 : A \\
\lambda(x : (A \to A) \to A).x d_2 : ((A \to A) \to A)
\end{align*}
\]

- The left top judgement can be derived by an assumption (more about this later).
Example 2, Using Rules

- We then used intro on the goal which was then replaced by $\lambda(x : A).d_3$.

- This corresponds to replacing $d_2$ by $\lambda(x : A).d_3$ which can be introduced by an introduction rule:

\[
\begin{align*}
\text{If } & x : (A \rightarrow A) \rightarrow A, a : A, d_3 : A \\
\text{then } & x : (A \rightarrow A) \rightarrow A \\
\text{implies } & x : (A \rightarrow A) \rightarrow A, \lambda(a : A).d_3 : A \\
\text{implies } & \lambda(x : (A \rightarrow A) \rightarrow A).x (\lambda(a : A).d_3) : ((A \rightarrow A) \rightarrow A)
\end{align*}
\]
Example 2, Using Rules

- Finally we used refine with \( a \), which replaced the goal by \( a \).

- This corresponds to replacing \( d_3 \) by \( a \).

\[
\begin{align*}
  x : (A \to A) \to A, a : A \\
  x : (A \to A) \to A \Rightarrow x : (A \to A) \to A \\
  x : (A \to A) \to A \Rightarrow x (\lambda(a : A).a) : A \\
  \lambda(x : (A \to A) \to A) . x (\lambda(a : A).a) : ((A \to A) \to A)
\end{align*}
\]

The right hand derivation can again be derived by an assume rule (more about this later).
Example 3

- We derive an element of type

\[ A \rightarrow B \rightarrow AB \]

where \( AB \) is the product of \( A \) and \( B \).

(See exampleProductIntro.agda).
Example 3 (Cont.)

• **Step 1:**

  – We postulate types $A$, $B$:

    
    \[
    \text{postulate } A :: \text{Type} \\
    \text{postulate } B :: \text{Type}
    \]

  – We introduce the product of $A$, $B$:

    * This will be a record with element $a : A$, $b : B$.

    \[
    AB :: \text{Type} \\
    = \text{sig}\{a :: A; b :: B\}
    \]
Example 3 (Cont.)

- **Step 2:**
  - Our goal is:

\[
f :: A \to B \to AB
\]

\[
= \{! !\}
\]
Example 3 (Cont.)

• **Step 3:**
  - We use intro.
    * An element of $A \to B \to AB$ will be of the form
      \[
      \lambda(a' :: A) \to \lambda(b' :: B) \to \{! !\}
      \]
    which is introduced by two introduction steps.
    * Agda will immediately carry out both of them.
    * We choose to use $a'$ instead of $a$, $b'$ instead of $b$, since 
      labels of $AB$. 

Example 3 (Cont.)

• Step 3 (Cont)
  – After applying intro we get

\[ f :: A \rightarrow B \rightarrow AB = \lambda (h :: \{! !\}) \rightarrow \lambda (h' :: \{! !\}) \rightarrow \{! \} \]

  – After applying agda-solve and renaming of variables we get

\[ f :: A \rightarrow B \rightarrow AB = \lambda (a' :: A) \rightarrow \lambda (b' :: B) \rightarrow \{! !\} \]
Example 3 (Cont.)

- **Step 4:**
  - The new goal is of type $AB$ which is a record type. An element of it can be introduced by an introduction.
  - Elements of type $AB$ introduced by the introduction principle will have the form
    \[
    \text{struct}\{a = \{! !\}; \quad b = \{! !\}; \} \]
  - When using intro we get:
    \[
    f :: A \to B \to AB \\
    = \lambda (a' :: A) \to \lambda (b' :: B) \to \text{struct}\{a = \{! !\}; b = \{! !\}; \}
    \]
Example 3 (Cont.)

- **Step 5:**
  - The first goal has as context:
    * $A, B : \text{Type}$,
    * $AB : \text{Type}$,
    * $f : A \to B \to AB$,
    * $a' : A$,
    * $b' : B$,
    * $a : A$,
    * $b : B$.
  - $a : A, b : B$ are the projections of the record we are defining, which might be used recursively.
  - Using $a$ and $b$ would in our example result in non-termination.
Example 3 (Cont.)

Step 5 (Cont)

- We insert $a$, use refine and solve the first goal:

$$f :: A \to B \to AB$$
$$= \lambda(a' :: A) \to \lambda(b' :: B) \to \text{struct}\{a = a', b = \{!\}\}$$
Example 3 (Cont.)

- **Step 6:**
  
  Similarly we can solve the second one:

\[
f :: A \rightarrow B \rightarrow AB = \lambda(a' :: A) \rightarrow \lambda(b' :: B) \rightarrow \text{struct}\{a = a', b = b'\}
\]
Example 3, Using Rules

• The definition of $AB$ means that $AB$ abbreviates $A \times B$, which can be derived as follows
  (using assumptions $A : \text{Type}$, $B : \text{Type}$):

  $\begin{array}{c}
  A : \text{Type} \\
  B : \text{Type} \\
  \hline
  A \times B : \text{Type}
  \end{array}$

• We won’t use this however, since it is required for the assumption rules, the treatment of which will be delayed until later.
Example 3, Using Rules (Cont.)

- Stating the goal corresponds to having as last line of the derivation:

\[ d_0 : A \rightarrow B \rightarrow (A \times B) \]

- Using intro means that we replace \( d_0 \) by \( \lambda(a' : A).\lambda(b' : B).d_1 \) introduced by two introduction rules:

\[
\frac{a' : A, b' : B \Rightarrow d_1 : A \times B}{a' : A \Rightarrow \lambda(b' : B).d_1 : B \rightarrow (A \times B)} \]

\[
\lambda(a' : A).\lambda(b' : B).d_1 : A \rightarrow B \rightarrow (A \times B) \]
Example 3, Using Rules (Cont.)

• Using intro again means that we replace $d_1$ by $\langle d_2, d_3 \rangle$, introduced by an introduction rule:

$$
\begin{align*}
\frac{a': A, b': B \Rightarrow d_2 : A \quad a': A, b': B \Rightarrow d_3 : B}{a': A, b': B \Rightarrow \langle d_2, d_3 \rangle : A \times B} \\
\frac{a': A \Rightarrow \lambda(b': B).\langle d_2, d_3 \rangle : B \Rightarrow (A \times B)}{\lambda(a': A).\lambda(b': B).\langle d_2, d_3 \rangle : A \Rightarrow B \Rightarrow (A \times B)}
\end{align*}
$$

Example 3, Using Rules (Cont.)

- Solving the goals by refining them with $a'$, $b'$ means that we replace $d_2$ by $b$, $d_3$ by $c$:

$$
\frac{a' : A, b' : B \Rightarrow a' : A \quad a' : A, b' : B \Rightarrow b'}{a' : A, b' : B \Rightarrow \langle a', b' \rangle : A \times B}
$$

$$
\frac{a' : A \Rightarrow \lambda(b' : B).\langle a', b' \rangle : A \times B}{\lambda(a' : A).\lambda(b' : B).\langle a', b' \rangle : A \to B \to (A \times B)}
$$

- The premises require an assumption rule (which will use $\langle a, b \rangle : A \times B$), see later for details.
Example 4

- We derive an element of type

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

where \(BC\) is the product of \(B\) and \(C\). (See exampleProductElim.agda)
• **Step 1:**
  - We postulate types $A$, $B$, $C$:
    
    \[
    \begin{align*}
    &\text{postulate } A :: \text{Type} \\
    &\text{postulate } B :: \text{Type} \\
    &\text{postulate } C :: \text{Type}
    \end{align*}
    \]
  
  - We introduce the product of $B$, $C$:
    
    \[
    BC :: \text{Type} \\
    = \text{sig}\{b :: B; c :: C\}
    \]
Example 4 (Cont.)

- **Step 2:**
  - Our goal is:

\[ f :: (A \rightarrow BC) \rightarrow A \rightarrow B = \{! !\} \]
• **Step 3:**
  
  – We use intro and get (after using agda-solve and renaming):

\[
 f :: (A \to BC) \to A \to B \\
 = \lambda (x :: A \to BC) \to \lambda (a :: A) \to \{! \}
\]
• **Step 4:**
  
  – The context has no element with result type $B$ (except for $f$, which results in a circular definition).
  – However, $x$ has function type with result type $BC$, which can be projected to $B$.
  – We introduce first an element of type $BC$ by a let-expression, and then derive from it the desired element of type $B$:
  – Using **agda-let (Make let expression)** with variable $bc$,

\[
 f :: (A \to BC') \to A \to B \\
 = \lambda (x :: A \to BC') \to \lambda (a :: A) \to \text{let } bc \\
\text{in } \{! !\}
\]

**Step 5:**

- We insert as type of variable $bc$ the type $BC$ (using refine)

\[
f :: (A \rightarrow BC) \rightarrow A \rightarrow B
= \lambda(x :: A \rightarrow BC) \rightarrow \lambda(a :: A) \rightarrow \text{let } bc
\]

\[
in \{! !\}
\]
Example 4 (Cont.)

- **Step 6:**
  - For solving the first goal (definition of $bc$) we can refine $x$, which has as result type $BC$.

\[
f :: (A \to BC) \to A \to B \\
= \lambda(x :: A \to BC) \to \lambda(a :: A) \to \text{let } bc :: BC \equiv x \{! !\} \text{ in } \{! !\}
\]
Step 7:

The new goal can be solved by refining it with variable $a$:

$$f :: (A \rightarrow BC') \rightarrow A \rightarrow B$$

$$= \lambda(x :: A \rightarrow BC') \rightarrow \lambda(a :: A) \rightarrow \text{let } bc$$

$$\text{in } \{! \}$$
Example 4 (Cont.)

- **Step 8:**
  - Currently, Agda doesn’t have any direct support for refining a term to an element of type $B$.
  - We have to do this by hand, insert $bc.b$, choose refine and obtain:

$$f :: (A \rightarrow BC) \rightarrow A \rightarrow B$$
$$= \lambda(x :: A \rightarrow BC) \rightarrow \lambda(a :: A) \rightarrow \text{let } bc \text{ in } bc.b$$
In our rule calculus we don’t introduce a let construction (we could add this).

In order to get close to the derivations, we omit in the Agda derivation the let expression, and replace in the body of it $bc$ by its definition ($xa$).

We get

\[
\begin{align*}
f &: (A \to BC) \to A \to B \\
es &= \lambda(x :: A \to BC) \to \lambda(a :: A) \to (xa)
\end{align*}
\]
Example 4, Using Rules

- Using rules we start with our goal

\[ d_0 : (A \rightarrow (B \times C)) \rightarrow A \rightarrow B \]
The intro step amounts to replacing $d_0$ by

$$\lambda(x : (A \rightarrow (B \times C))).\lambda(a : A).d_1$$

introduced by two applications of an introduction rule:

$$x : A \rightarrow (B \times C), a : A \Rightarrow d_1 : A$$

$$\frac{x : A \rightarrow (B \times C) \Rightarrow \lambda(a : A).d_1 : A \rightarrow 1}{\lambda(x : A \rightarrow (B \times C)).\lambda(a : A).d_1 : (A \rightarrow (B \times C))}$$
Example 4, Using Rules (Cont.)

- In Agda, we then replace the goal corresponding to $d_1$ by $(xa).b$.

- In our rule calculus, this reads $\pi_0(x a)$.

- This can be introduced by two applications of elimination rules:

\[
\begin{align*}
\frac{x : A \rightarrow (B \times C), a : A \Rightarrow x : A \rightarrow (B \times C)}{x : A \rightarrow (B \times C), a : A \Rightarrow xa : B \times C} \\
\frac{x : A \rightarrow (B \times C), a : A \Rightarrow \pi_0(xa) : B}{x : A \rightarrow (B \times C), a : A \Rightarrow \lambda(a : A).\pi_0(xa) : (A \rightarrow (B \times C))}
\end{align*}
\]

- The two initial judgements can be introduced by assumption rules.
• $\Gamma, \Delta$ denote contexts.
  So $\Gamma, \Delta$ stand for expressions like
  \[
  x : A, y : B, z : C
  \]

• The context $\Gamma, \Gamma'$ is the result of concatenating $\Gamma, \Gamma'$:
  
  - E.g. if $\Gamma = x : A, y : B$ and $\Gamma' = z : C$, then $\Gamma, \Gamma' = x$
    
  * In fact $\Gamma'$ is usually not really a context, but a “context piece” – it might depend on $\Gamma$. 

Notations for Contexts (Cont.)

• Similarly, for $\Gamma = x : A, y : B$, the expression $\Gamma, z : D$ stands for $x : A, y : B, z : D$.

• $\emptyset$ is the empty context (no variables are bound in it).

  A non-dependent judgement $\theta$ (e.g. $A : \text{Type}$) is an abbreviation for $\emptyset \Rightarrow \theta$. 
Sometimes we need as assumptions of an axiom the assertion
“Γ is a valid context”.

- If Γ is $x : A, y : B, z : C$ this would mean
  * $A : \text{Type}$.
  * $x : A \Rightarrow B : \text{Type}$.
  * $x : A, y : B \Rightarrow C : \text{Type}$.
- We form a **new judgement**

\[ \Gamma \Rightarrow \text{Context} \]

expressing “Γ is a valid context”.

- This judgement **does not occur explicitly in Agda**.
Context Rules

The empty context

\[ \emptyset \Rightarrow \text{Context} \]

Extending a context

\[ \Gamma \Rightarrow A : \text{Type} \]

\[ \Gamma, x : A \Rightarrow \text{Context} \]

(where in the last rule \( x \) must not occur in \( \Gamma \)).
Example Derivation (Context Rules)

- We assume the following formation rule for the type of natural numbers:

\[
N : \text{Type}
\]

- With this rule, following the convention on slide B2-62, we have introduced the rules

\[
\frac{\Gamma \Rightarrow \text{Context}}{\Gamma \Rightarrow N : \text{Type}}
\]
Example Derivation (Context Rules)

- The following derives $x : N, y : N, z : N \Rightarrow \text{Context}$
  (Note that $N : \text{Type}$ is the same as $\emptyset \Rightarrow N : \text{Type}$):

  $\begin{align*}
  N : \text{Type} \\
  \frac{x : N \Rightarrow \text{Context}}{x : N \Rightarrow N : \text{Type}} \\
  \frac{x : N, y : N \Rightarrow \text{Context}}{x : N, y : N \Rightarrow N : \text{Type}} \\
  \frac{x : N, y : N, z : N \Rightarrow \text{Context}}{}
  \end{align*}$
**Assumption Rule**

\[
\begin{align*}
& \Gamma, x : A, \Gamma' \Rightarrow \text{Context} \\
\hline
& \Gamma, x : A, \Gamma' \Rightarrow x : A
\end{align*}
\]
Example Derivation (Assumption Rule)

- We extend the derivation of B2-124 to a derivation of $x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow y : \mathbb{N}$:

$$\frac{x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow \text{Context}}{x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow y : \mathbb{N}}$$

- Similarly we can derive $x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow z : \mathbb{N}$:

$$\frac{x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow \text{Context}}{x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow z : \mathbb{N}}$$
Example Derivation (Assumption Rule; Cont)

- The full derivation of first judgement on the previous slide is as follows:

\[
\begin{align*}
N &: \text{Type} \\
\frac{\, x : N \\ \Rightarrow \text{Context} \,}{x : N \Rightarrow N : \text{Type}} \\
\frac{\, x : N, y : N \\ \Rightarrow \text{Context} \,}{x : N, y : N \Rightarrow N : \text{Type}} \\
\frac{\, x : N, y : N, z : N \Rightarrow \text{Context} \,}{x : N, y : N, z : N \Rightarrow y : N}
\end{align*}
\]
• When we define a function:

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

we can make use of \( a :: A \) when solving the goal \( \{! !\} \).

– This is an application of the assumption rule:
  When solving \( \{! !\} \) we essentially define under the assumption \( a :: A \) an element \( \{! !\} :: B \).
• The above corresponds to a derivation

\[
\frac{a : A \Rightarrow \{! !\} : B}{\lambda(a : A).\{! !\} : A \rightarrow B}
\]

• If \(B\) is equal to \(A\) we can use the assumption rule directly

\[
\frac{a : A \Rightarrow a : B}{\lambda(a : A).a : A \rightarrow B}
\]

in order to solve this goal.
Assumption Rule in Agda (Cont.)

- More generally we might in the derivation of $a : A \Rightarrow \{\}$ anywhere use of $a : A$, as long as this is in the context.

\[
\begin{align*}
\vdots \\
\frac{a : A \Rightarrow a : A}{\vdots} \\
\frac{a : A \Rightarrow s : B}{\lambda(a : A).s : A \rightarrow B}
\end{align*}
\]
• Similarly, when solving the goal

\[ f :\ A \rightarrow B = \lambda (a :: A) \rightarrow \{! !\} \]

in \{! !\} we can make use of \( a :: A \).

– In fact when solving the above, we implicitly use the rule

\[
\frac{a : A \Rightarrow \{! !\} : B}{\lambda (a : A).\{! !\} : A \rightarrow B}
\]

So we have to solve

\[ a : A \Rightarrow \{! !\} : B \]

in order to derive

\[ \lambda (a : A).\{! !\} : A \rightarrow B \]
Weakening Rule

\[
\frac{\Gamma, \Gamma' \Rightarrow \theta}{\Gamma, \Delta, \Gamma' \Rightarrow \text{Context}} \quad \frac{\Gamma, \Delta, \Gamma' \Rightarrow \theta}{\Gamma, \Delta, \Gamma' \Rightarrow \theta}
\]

- \(\theta\) stands for an arbitrary non-dependent judgement.

- This rule allows to add an additional context piece (\(\Delta\)) to a judgement.
  - The judgement \(\Gamma, \Gamma' \Rightarrow \theta\) is weakened by \(\Delta\).
Weakening Rule (Cont.)

• Remark: One can in fact show that the Thinning rule can be weakly derived.
  – **Weakly derived** means: whenever the assumptions of the rule can be derived in the complete set of rules we provide, then as well the conclusion.
  – However, this can’t be derived from the premise the conclusion.

• An exception is when we *additionally assume some judgements* for instance $A : \text{Type}$ (corresponding to “postulate” in Agda).
  – Then $\Gamma \Rightarrow A : \text{Type}$ doesn’t follow without the weakening rule.
Example Derivation (Weakening Rule)

- The following derives the first premise in Example 3 (slide B2-105) from assumptions $A : \text{Type}$, $b : \text{Type}$:

$$
\begin{align*}
A : \text{Type} \\
B : \text{Type} & \quad \frac{a' : A \Rightarrow \text{Context}}{\quad a' : A} \\
& \quad \frac{a' : A \Rightarrow B : \text{Type}}{\quad a' : A, b' : B \Rightarrow \text{Context}} \\
& \quad \frac{\quad a' : A, b' : B \Rightarrow a' : A}{\text{Context}}
\end{align*}
$$
Example Derivation 2 (Weakening Rule)

- The following derives the first premise in Example 4 (slide B2-118) from assumptions $A : \text{Type}$, $B : \text{Type}$, $C : \text{Type}$:

$$
\frac{
A : \text{Type} \quad B : \text{Type} \quad C : \text{Type}
}{
A \rightarrow (B \times C) : \text{Type}
}$$

$$
\frac{
B \times C : \text{Type}
}{
A \rightarrow (B \times C) : \text{Type}
}$$

$$
\frac{
A : \text{Type} \quad x : A \rightarrow (B \times C) \Rightarrow \text{Context}
}{
x : A \rightarrow (B \times C) \Rightarrow A : \text{Type}
}$$

$$
\frac{
x : A \rightarrow (B \times C) \Rightarrow A : \text{Type}
}{
x : A \rightarrow (B \times C) \Rightarrow \text{Context}
}$$

$$
\frac{
x : A \rightarrow (B \times C) \Rightarrow \text{Context}
}{
x : A \rightarrow (B \times C), a : A \Rightarrow x : A \rightarrow (B \times C)
}$$
General Equality Rules

Reflexivity

\[
\frac{A : \text{Type}}{A = A : \text{Type}}
\]

\[
\frac{a : A}{a = a : A}
\]

(Reflexivity can be weakly derived, except for additional assumptions.

Symmetry

\[
\frac{A = B : \text{Type}}{B = A : \text{Type}}
\]

\[
\frac{a = b : A}{b = a : A}
\]
General Equality Rules (Cont.)

Transitivity

\[
A = B : \text{Type} \quad B = C : \text{Type} \\
\overline{A = C : \text{Type}}
\]

\[
a = b : A \quad b = c : A \\
\overline{a = c : A}
\]

Transfer

\[
a : A \quad A = B : \text{Type} \\
\overline{a : B}
\]

\[
a = b : A \quad A = B : \text{Type} \\
\overline{a = b : B}
\]
Example Derivation (General Equality Rules)

\[
\begin{align*}
N : \text{Type} & \\
y : N & \Rightarrow \text{Context} \\
y : N & \Rightarrow N : \text{Type} \\
y : N, x : N & \Rightarrow x : N \\
y : N, x : N & \Rightarrow (\lambda(x : N).x) y : N \\
y : N & \Rightarrow (\lambda(x : N).x) y : N \\
y : N & \Rightarrow y + 0 = (\lambda(x : N).x) y : N \\
y : N & \Rightarrow y + 0 = (\lambda(x : N).x) y : N \\
y : N & \Rightarrow y + 0 = (\lambda(x : N).x) y : N \\
\lambda(y : N).y + 0 & = \lambda(y : N).(\lambda(x : N).x) y : N \rightarrow N
\end{align*}
\]
• In the previous derivation, the most complicated step was:

\[
\frac{y : N, x : N \Rightarrow x : N \quad y : N \Rightarrow y : N}{y : N \Rightarrow (\lambda(x : N).x) y = y : N}
\]

• This is an example of the equality rule for the non-dependent function type (B2-34):

\[
\frac{x : A \Rightarrow b : B \quad a : A}{(\lambda(x : A).b) a = b[x := a] : B}
\]

- This was weakened by an additional context \( y : N \).
- Then we have the substitutions:
  * \( A = B = N \).
  * \( b = x \).
  * \( a = y \).
  * Therefore \( b[x := a] = y \).
Example Derivation (General Equality Rules; Cont.)

- Note that from the premises of that rule

\[
\begin{align*}
  y : N, x : N & \Rightarrow x : N \\
  y : N & \Rightarrow y : N \\
  y : N & \Rightarrow (\lambda(x : N).x) \ y = y : N
\end{align*}
\]

we can derive using the introduction and elimination rule

\[
y : N \Rightarrow (\lambda(x : N).x) \ y : N
\]
as follows:

\[
\begin{align*}
  y : N, x : N & \Rightarrow x : N \\
  y : N & \Rightarrow \lambda(x : N).x : N \rightarrow N \\
  y : N & \Rightarrow y : N \\
  y : N & \Rightarrow (\lambda(x : N).x) \ y : N
\end{align*}
\]

Example Derivation (General Equality Rules; Cont.)

• The equality rule expresses how the function $\lambda(x : N).x$ applied to $y$ is evaluated as follows:
  - We evaluate the body of the function $(x)$ by setting for $x$ the argument of the function $(y)$.
  - This is the same as substituting in the body for $x$ the argument of the function, i.e. $y$.

• This explains how the detour above of first introducing and then eliminating an expression can be reduced (namely to $y$ or in general to $b\ [x := a]$).
The following rules can be weakly derived:

**Substitution 1**

\[
\frac{\Gamma, x : A, \Gamma' \Rightarrow \theta \quad \Gamma \Rightarrow a : A}{\Gamma, \Gamma'[x := a] \Rightarrow \theta[x := a]}
\]

(\(\Gamma'[x := a]\) is the result of substituting in \(\Gamma'\) all occurrences of \(x\) by \(a\)).

**Substitution 2**

\[
\frac{\Gamma, x : A, \Gamma' \Rightarrow B : \text{Type} \quad \Gamma \Rightarrow a = a' : A}{\Gamma, \Gamma'[x := a] \Rightarrow B[x := a] = B[x := a'] : \text{Type}}
\]

**Substitution 3**

\[
\frac{\Gamma, x : A, \Gamma' \Rightarrow b : B \quad \Gamma \Rightarrow a = a' : A}{\Gamma, \Gamma'[x := a] \Rightarrow b[x := a] = b[x := a'] : B[x := a]}
\]
Example Derivation (Substitution)

\[
\begin{align*}
\cdots & \quad x : \mathbb{N}, y : \mathbb{N} \Rightarrow x : \mathbb{N} \quad \cdots \\
& \quad x : \mathbb{N}, y : \mathbb{N} \Rightarrow y : \mathbb{N} \\
& \quad x : \mathbb{N}, y : \mathbb{N} \Rightarrow x + y : \mathbb{N} \\
& \quad y : \mathbb{N} \Rightarrow 0 + y : \mathbb{N} \\
& \quad \lambda(y : \mathbb{N}).0 + y : \mathbb{N} \Rightarrow \mathbb{N}
\end{align*}
\]

Example Derivation 2 (Substitution)

\[
\begin{align*}
N : \text{Type} \\
\text{Con} \\
\text{S(z)} \\
\text{S(z) + 0} = \text{S(z) + y} \\
\lambda(y : N). (\text{S(z) + 0} + y) = \lambda(y : N). (\text{S(z) + y})
\end{align*}
\]
Presuppositions

- In order to derive $x : A, y : B \Rightarrow C : \text{Type}$ we need to show:
  - $A : \text{Type}$.
  - $x : A \Rightarrow B : \text{Type}$

- So the judgement $x : A, y : B \Rightarrow C : \text{Type}$ implicitly contains the judgements

\[
A : \text{Type}, \quad x : A \Rightarrow B : \text{Type}.
\]
• $A : \text{Type and } x : A \Rightarrow B : \text{Type are presuppositions of the judgement }\ x : A, y : B \Rightarrow C : \text{Type}$. 

• The next slide shows the presuppositions of judgements.
<table>
<thead>
<tr>
<th>Judgement</th>
<th>Presuppositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, x : A \Rightarrow \text{Context}$</td>
<td>$\Gamma \Rightarrow A : \text{Type}$.</td>
</tr>
<tr>
<td>$\Gamma \Rightarrow A : \text{Type}$</td>
<td>$\Gamma \Rightarrow \text{Context}$</td>
</tr>
<tr>
<td>$\Gamma \Rightarrow A = B : \text{Type}$</td>
<td>$\Gamma \Rightarrow A : \text{Type}$, $\Gamma \Rightarrow B : \text{Type}$.</td>
</tr>
<tr>
<td>$\Gamma \Rightarrow a : A$</td>
<td>$\Gamma \Rightarrow A : \text{Type}$.</td>
</tr>
</tbody>
</table>
## Presuppositions

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Presuppositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \Rightarrow a = b : A$</td>
<td>$\Gamma \Rightarrow a : A$,</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \Rightarrow b : A$.</td>
</tr>
<tr>
<td>$\Gamma \Rightarrow (x : A) \times B : \text{Type}$</td>
<td>$\Gamma, x : A \Rightarrow B : \text{Type}$.</td>
</tr>
<tr>
<td>$\Gamma \Rightarrow (x : A) \rightarrow B : \text{Type}$</td>
<td>$\Gamma, x : A \Rightarrow B : \text{Type}$.</td>
</tr>
</tbody>
</table>
Furthermore, *presuppositions of presuppositions* of 
\[ \Gamma \Rightarrow \theta \]
are as well *presuppositions* of 
\[ \Gamma \Rightarrow \theta \].
Example of Presuppositions

- \( x : A, y : B \Rightarrow a = b : (z : C) \times D \) presupposes:
  - \( \emptyset \Rightarrow \) Context,
  - \( A \) : Type,
  - \( x : A \Rightarrow \) Context,
  - \( x : A \Rightarrow B : Type, \)
  - \( x : A, y : B \Rightarrow \) Context,
  - \( x : A, y : B \Rightarrow C : Type, \)
  - \( x : A, y : B, z : C \Rightarrow \) Context,
  - \( x : A, y : B, z : C \Rightarrow D : Type, \)
  - \( x : A, y : B \Rightarrow (z : C') \times D : Type, \)
  - \( x : A, y : B \Rightarrow a : (z : C) \times D, \)
  - \( x : A, y : B \Rightarrow b : (z : C') \times D. \)
Remark on $A \rightarrow B$, $A \times B$

- Note that $A \rightarrow B$ is an **abbreviation** for $(x : A) \rightarrow B$ for some fresh $x$.

- Similarly $A \times B$ is an **abbreviation** for $(x : A) \times B$ for some fresh $x$.

- Therefore the presupposition of $A \rightarrow B : \text{Type}$ (which abbreviates $\emptyset \Rightarrow A \rightarrow B : \text{Type}$) are:

  - $\emptyset \Rightarrow \text{Context},$
  - $A : \text{Type},$
  - $x : A \Rightarrow \text{Context},$
  - $x : A \Rightarrow B : \text{Type}.$
• We would like to add operations on types, such as

\[ \text{prod : Type} \to \text{Type} \to \text{Type} \]

which should take two types and form the product of it.

• The problem is that for this we need

\[ \text{Type} \to \text{Type} \to \text{Type} : \text{Type} \]

and our rules allow this only if we had

\[ \text{Type} : \text{Type} \]
• Adding \( \text{Type} : \text{Type} \)

as a rule results however in an \textit{inconsistent theory}:

– using this rule \textit{we can prove everything}, especially false formulas.

The corresponding paradox is called \textit{Girard’s paradox}.

Per Martin-Löf

The Founder of Martin-Löf Type Theory.

The main theoretician behind Agda (which was implemented by his wife, of whom I have no picture).
• Instead we introduce a **new type**

\[
\text{Set} : \text{Type}
\]

**Set** is the **type of sets**.

– A **set** is a **small type**.
• We add rules asserting that if $A : \text{Set}$ then $A : \text{Type}$.
  
  – Since $\text{Set} : \text{Type}$ we get
  
  $$\text{Set} \rightarrow \text{Set} \rightarrow \text{Set} : \text{Type}$$

  and we can assign to $\text{prod}$ above the type

  $$\text{prod} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$$

  (Of course $\text{prod}$ needs to be defined – this will be done.)
– However, we cannot use \texttt{prod} in order to form the product of two sets, ie. we cannot introduce

\[
\text{prod Set Set : Set ,}
\]

since \text{Set : Set} does not hold.

* That would result in the same inconsistency as \text{Type : Type}.
Formation Rule for Set

Set : Type

Every Set is a Type

\[
\frac{A : \text{Set}}{A : \text{Type}}
\]

Closure of Set under the dependent product

\[
\frac{A : \text{Set} \quad x : A \Rightarrow B : \text{Set}}{(x : A) \times B : \text{Set}}
\]

Closure of Set under the dependent function type

\[
\frac{A : \text{Set} \quad x : A \Rightarrow B : \text{Set}}{(x : A) \rightarrow B : \text{Set}}
\]
Equality Versions of the Above Rules

Formation Rule for Set

\[ \text{Set} = \text{Set} : \text{Type} \]

Every Set is a Type

\[
\frac{A = B : \text{Set}}{A = B : \text{Type}}
\]

Closure of Set under the dependent product

\[
\frac{A = A' : \text{Set}}{(x : A) \times B = (x : A') \times B' : \text{Set}}
\]

Closure of Set under the dependent function type

\[
\frac{A = A' : \text{Set}}{(x : A) \to B = (x : A') \to B' : \text{Set}}
\]
We can now introduce $\text{prod} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$:

First we derive $X : \text{Set}, Y : \text{Set} \Rightarrow X : \text{Set}$:

$$
\begin{align*}
\text{Set} : \text{Type} \\
X : \text{Set} \Rightarrow \text{Context} \\
X : \text{Set} \Rightarrow \text{Set} : \text{Type} \\
X : \text{Set}, Y : \text{Set} \Rightarrow \text{Context} \\
X : \text{Set}, Y : \text{Set} \Rightarrow X : \text{Set}
\end{align*}
$$
Example: prod (Cont.)

Using this we can derive

\[ X : \text{Set}, Y : \text{Set}, x : X \Rightarrow Y : \text{Set} \]

as follows:

\[
\begin{align*}
X : \text{Set}, Y : \text{Set} & \Rightarrow X : \text{Set} \\
X : \text{Set}, Y : \text{Set} & \Rightarrow X : \text{Type} \\
X : \text{Set}, Y : \text{Set}, x : X & \Rightarrow \text{Context} \\
X : \text{Set}, Y : \text{Set}, x : X & \Rightarrow Y : \text{Set}
\end{align*}
\]
Example: prod (Cont.)

Now we can derive our desired judgement:

\[
\begin{align*}
X : \text{Set}, Y : \text{Set} & \Rightarrow X : \text{Set} & X : \text{Set}, Y : \text{Set}, x : X \\
X : \text{Set}, Y : \text{Set} & \Rightarrow (x : X) \times Y : \text{Set} \\
X : \text{Set} & \Rightarrow \lambda(Y : \text{Set}).(x : X) \times Y : \text{Set} \rightarrow \text{Set} \\
\lambda(X, Y : \text{Set}).(x : X) \times Y : \text{Set} & \rightarrow \text{Set} \rightarrow \text{Set}
\end{align*}
\]

So define

\[
\text{prod} := \lambda(X, Y : \text{Set}).(x : X) \times Y
\]
Hierarchies of Types

- If one wants to form \( \text{prod} : \text{Type} \to \text{Type} \to \text{Type} \), one needs to have a further level \( \text{Kind} \), s.t. \( \text{Type} : \text{Kind} \).
  - Then \( \text{Type} \to \text{Type} \to \text{Type} : \text{Kind} \).
Rules for Type as a Kind

**Type is a Kind**

\[\text{Type} : \text{Kind}\]

**Every Type is a Kind**

\[A : \text{Type} \quad \overline{\quad A : \text{Kind}}\]

**Closure of Kind under the dependent product**

\[A : \text{Kind} \quad x : A \Rightarrow B : \text{Kind} \quad \overline{(x : A) \times B : \text{Kind}}\]

**Closure of Kind under the dependent function type**

\[A : \text{Kind} \quad x : A \Rightarrow B : \text{Kind} \quad \overline{(x : A) \rightarrow B : \text{Kind}}\]

Plus **equality versions** of the above rules.
Hierarchies of Types (Cont.)

• This can be iterated further, forming
  \( \text{Type} = \text{Type}_1, \text{Kind} = \text{Type}_2, \text{Type}_3, \text{Type}_4 \cdots \)

• Agda has a hierarchy of types built in, written as \#1 (which is \text{Type}), \#2 (in the rule calculus called \text{Kind}), \#3 etc.

• So we have
  – Set : Type,
  – Set : \#2, Type : \#2,
  – Set : \#3, Type : \#3, \#2 : \#3,
  – Set : \#4, Type : \#4, \#2 : \#4, \#3 : \#4,
  – etc.