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.
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.
The formation rules introduce new types. Each type constructor 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

- \( \text{List}(A) \) is the type of lists of type \( A \).
- The type-constructor is \( \text{List} \).

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

(C) Anton Setzer 2003 (except for pictures)
Example 2: The Type of Natural Numbers

Formation rule for the type of natural numbers:

\[ N : \text{Type} \]

- Later we will see that we can replace in this example \( \text{Type} \) by \( \text{Set} \).

- Note that the formation rule for \( N \) has 0 premises (therefore the fraction bar is omitted).

- If it has 0 arguments, and we write \( N \) instead of \( N() \).

- The type-constructor is \( N \).

Note: Setzer 2003 (except for pictures)
Example 3: The Non-Dependent Product

- The type-constructor is $(\times)$.
- $A \times B$ stands for $(\forall A' B' (A' \times B') \Rightarrow A \times B)$.

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

Formation rule for the non-dependent product:
The formation of a type is usually done by introducing a constant of a certain type.

Example 1:

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

Formation Rules in Agda
Agda syntax for introducing the non-dependent product:

\[
\text{example 2: } (\times)
\]
Currying in Agda

Traditionally one writes in type theory type constructors in uncurried form. •

We have to write List(A).

- List alone does not make sense as a term.

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)

List :: Type ← Type

- We write therefore List A and not List(A).

· Traditionally one writes in type theory type constructors in uncurried form.

· List A means the application of the function List to A.

· We write therefore List A and not List(A).
• Agda allows to write \( A \times B \) for \( \times \) and returns \( N \times B \).

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

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

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

Currying in Agda (cont.)
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 \( C() \).
Thetype \texttt{NatList} of Lists of type \texttt{N} has two introduction rules:

- **Example 1a**

\[
\text{\texttt{nil} : \texttt{NatList}} \\
\text{\texttt{cons} \ (n, l) : \texttt{NatList}}
\]

\[
\frac{\text{\texttt{l} : \texttt{NatList}}} {\text{\texttt{n} : \texttt{N}}} \\
\text{\texttt{nil} : \texttt{NatList}}
\]
In case of the rule for \texttt{cons}, this premise is implicit in the premise $a : A$.

\begin{align*}
\frac{\text{cons}(a, l)}{(\forall) \text{List} : \text{List}(A)} \quad (\forall) \text{List} : \text{List}(A) \quad A : \text{Type}
\end{align*}

In case of the rule for \texttt{nil}, we needed the premise $A : \text{Type}$ in order to

\begin{align*}
\frac{\text{nil}}{(\forall) \text{List} : \text{List}(A)} \quad (\forall) \text{List} : \text{List}(A) \quad A : \text{Type}
\end{align*}

Lists of type $A$ have two introduction rules:

- We generalize the previous example to lists of arbitrary type.
Example 2: Natural Numbers.

The natural numbers $\mathbb{N}$ can be considered as being formed from two operations:

\[ \begin{align*}
\mathbb{N} & : (u) \\
\mathbb{N} & : u \\
\mathbb{N} & : 0
\end{align*} \]

- The introduction rules of $\mathbb{N}$ are:
  - So the constructors of $\mathbb{N}$ are 0 and $S$.

Using these two operations we can form $0$, $S(0) = 1$, $S(1) = 2$, $S(2) = 3$, and so on...

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

$\mathbb{N}$ can be considered as being formed from two operations:

- $S$ where $S(0) = 1$.
- $0$. 

Constructors and Canonical Elements

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

Examples:

- nil, cons(1, cons(0, nil)) in case of List(N).
- nil, cons(1 + 1, cons(0, nil)) in case of List(N).

Here 2 stands for S(S(0)) and 3 for S(S(S(0))).

- 0, S(2 + 3) in case of N.

Canonical elements therefore always start with a constructor.
Reducions

Terms can usually be reduced further.

Example:

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

Further reductions can be applied to subterms.

Example: Using the above reduction we obtain:

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

stands for, \text{reduces to}
The reduction rules for addition on $\mathbb{N}$ are:

\[
\begin{align*}
(n + 0) & \mapsto n, \\
(n + S(m)) & \mapsto S(n + m).
\end{align*}
\]

One-step reductions are the reductions obtained by applying these rules (and reductions using them in subterms) to the summands. Reductions can be formed by a sequence of one-step reductions. An example:

\[
0 + S(0) \mapsto S(0 + 0) \mapsto S(0).
\]

The reduction rules for addition on $\mathbb{N}$ are:

\begin{center}
Reduction Rules for $\mathbb{N}$
\end{center}
Canonicalelementscanonlybereducedfurtherbyreducingtheconstructor.

Theconstructorwillalwaysremaininplace.

Forinstance

\[
S(3 + 2) = (S(3))S
\]

Thisreductioncanbeformedfromonestepreducitonsasfollows:

Forinstance \((S(3 + 2))\) reduces to \((S(3))S\).

Theconstructorwillalwaysremaininplace.

Canoniclelementscanonlybereducedfurtherbyreducingthearguments.
Independent of each other.

Further, the arguments of the constructor reduce individually and

An element starting with cons will never reduce to one starting with nil.

\[
(\text{cons}(2, \text{cons}(3, \text{cons}(5, \text{nil}))))
\]

Similarly

- It cannot change later to 0.

This information will remain as it is when further reducing it.

- Once we have determined that we have the "successor of something",

- The outermost $S$ always remains in place.

Constructors and Canonical Elements (Cont.)
Any element of a type has to reduce to a canonical element of it.

- $2 + 3$ is a non-canonical element, and $+$ is not a constructor.
- $\text{concat}(\text{cons}(2, \text{nil}), \text{nil})$ is a non-canonical element, and $\text{concat}$ is not a constructor.

$\text{concat}(\text{cons}(2, \text{nil}), \text{nil})$ changes to $\text{cons}(2, \text{nil})$.

- When reducing $2 + 3$, the outermost operator changes to $\text{cons}$.

$(((((0)S)S)S)S)S \leftarrow 2 + 3 = 5$.

Note that $2 + 3$ is only a more readable way of writing $(+)(2', 3')$: $2 + 3$ is a non-canonical element, and $(+)$ is not a constructor.
Constructors in Agda

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

As type-constructors in Agda, constructors are curried:

\[
\begin{align*}
\text{nil} \@ (\text{List}\ N) & \rightarrow \\
(\text{cons} \@ (\text{List}\ N) \@ n) & \rightarrow \\
\text{nil} \@ (\text{List}\ N) & \rightarrow \\
(\text{cons} \@ (\text{List}\ N) \@ n) & \rightarrow
\end{align*}
\]

We have

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

Constructors in Agda
Sincenotationslike
\texttt{nil@(ListN)}
isusuallytocumbersome,itisbetterto
introduceabbreviations:

\begin{verbatim}
\texttt{nil::ListN} = \texttt{nil@}
\texttt{cons(n::N)} \texttt{(l::ListN)}::ListN = \texttt{cons@nl}
\end{verbatim}

Notethattheaboveintroduces\texttt{nil},\texttt{cons}for\texttt{ListN},andnotforthe
\texttt{List\_A}foranytype\texttt{A}.(Thatwouldrequireanextraargument)

\textbf{General case} \texttt{List\_A}foranypertype\texttt{A}.

\texttt{nil} and \texttt{cons}for\texttt{ListN},andnotforthe
\texttt{List\_A}foranytype\texttt{A}.(Thatwouldrequireanextraargument).

\textit{CriticalSystems,CS}
\textbf{411,Lentterm2003,Sec.B2}

\textbf{B2-18}
Elimination rules allow to take an element of a type and compute from it another type.

Example 1: First and second projection of a product:

\[ q = \langle \langle q, a \rangle \rangle_1 = a, \quad \langle q, a \rangle_0 = b. \]

- Equality rules will express \( \langle q, a \rangle_1 = a \)

\[
\begin{align*}
    B : (c) & \quad \frac{}{B \times A : c} \\
    A : (c) & \quad \frac{}{B \times A : c}
\end{align*}
\]
Example 2: Addition in \( \mathbb{N} \)

Recursively ones. Introduce all functions we expect to be definable, including all primitive recursive ones. Instead we will introduce one general elimination rule which allows to proceed like this would require one elimination rule for each function from \( \mathbb{N} \) we want to define.

Equality rules will express

- \( (m + n) + u = m + (n + u) \)
- \( u + 0 = u \)

\[
\frac{N : m + n}{N : m} \quad \frac{N : m}{N : n}
\]
Elimination rules invert the introduction rules.

Elimination rules:

- Reduce $a$ (which is of type $A$) to its canonical form.
- This element must be of the form $\langle a', q \rangle$.
- Reduce $c$ to a canonical element.

Therefore, if $c : A \times B$, the canonical form of $\pi_0(c)$ can be computed as:

follows:

A non-canonical element of type $\pi_0(c)$ must reduce to a canonical element.

Therefore, if $c : A \times B$, the canonical elements are of the form $\langle a', q \rangle$ for $a : A$, $q : B$.

Elimination rules invert the introduction rules.
Example: Definition of addition in \( \mathbb{N} \):

\[
\begin{cases}
\{ \text{\( m + n \)} \} \leftarrow (m, S) \\
\text{\( n \)} \leftarrow (Z) \\
\text{\( \text{case} \ m \text{ of} \rightleftharpoons \) } \\
\text{\( \mathbb{N} \) :} \\
\text{\( \mathbb{N} : m, n \) \( (+) \)} 
\end{cases}
\]

- For user-defined types, elimination is realized by case distinction.
- Elimination for built-in types has special notation.

Elimination in Agda
The canonical element for an element, which is the result of an elimination, can be always computed as follows:

1. Reduce the element to be eliminated to canonical form.
2. Then make one reduction step (Red).
3. The result will be a canonical or non-canonical element of the target type.
4. Reduce it to canonical form.

For instance in case of $A \times B$, (Red) are the reductions:

- $q \leftarrow (\langle q, a \rangle)_{\text{fr}}$
- $a \leftarrow (\langle q, a \rangle)_{\text{fr}}$

Reduce it to canonical form.

The canonical element for an element, which is the result of an elimination,

(4) Equality Rules
The result is already in canonical form.

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

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

- Note that the second argument is the argument which we are eliminating.

\[(w + u)S \leftarrow (w)S + u \leftarrow u \leftarrow 0 + u\]

In case of \((+)'\) (Red) are the reductions.
Equality rules express (Red) type theoretically.

- They describe what happens if one first introduces an element and then immediately eliminates it.
The equality rule explains how to reduce that element (namely to \( \forall \).

So it is derived by first introducing \( \langle q, \nu \rangle \) and then eliminating it immediately.

\[
\begin{align*}
A : (\langle q, \nu \rangle)^0 \\
B \times A : \langle q, \nu \rangle \\
B : q \quad A :
\end{align*}
\]

In the first judgment we can derive \( \forall \) as follows:

\[
\begin{align*}
A : \forall \nu = (\langle q, \nu \rangle)^0 \\
B : q \quad A :
\end{align*}
\]

Equality rules for \( B \times A \)
The second equality rule is similar:

\[ \frac{\mathcal{B} : q = (\langle q', a \rangle) \not\equiv \mathcal{B} : q}{\mathcal{A} : a} \]

Example (Equality, Rule, Cont.)
The first equality rule for \( + \) is as follows:

\[
\begin{align*}
\text{N} : 0 + u & \\
\text{N} : 0 & \\
\text{N} : u
\end{align*}
\]

The equality rule explains how to reduce \( u + 0 \).

\[
\begin{align*}
\text{N} & : 0 + u \\
\text{N} & : 0 \\
\text{N} & : u
\end{align*}
\]

The right side is an axiom, the left side has to be concluded using some derivation.

- The first equality rule for \( + \) is as follows:

\[
\begin{align*}
\text{N} : u = 0 + u & \\
\text{N} : u
\end{align*}
\]

The first equality rule for \( + \) is as follows:
Example 3 (Equality Rule)

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

\[
\begin{align*}
\text{N} : (w + u)S &= (w)S + u \\
\text{N} : w &\quad \text{N} : u
\end{align*}
\]

: Eliminating it using $+$ and then by

\[
\frac{\text{N} : (w)S + u}{\text{N} : (w)S} \quad \frac{\text{N} : u}{\text{N} : u}
\]

\[
\text{N} : (w)S + u
\]

\[
\text{N} : u
\]
Equality Rules in Agda are implicit. The notation for elimination however indicates already how the reductions take place. Functions corresponding to elimination are defined by telling how elimination operates.

\[
\begin{align*}
&\{ (\_m + u) \_S \leftarrow (\_m \_S) \\
&\_u \leftarrow (\_Z) \\
&\text{case } \_m \text{ of} \\
&\quad N :: \\
&\quad (N :: \_m, \_u) \quad (+)
\end{align*}
\]
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 \( N \) (degenerated):

\[
\begin{align*}
\text{Type} & : N = N \\
\text{Type} & : B = A
\end{align*}
\]

Example: equality version of the formation rule for \( \text{List} \):

\[
\frac{\text{List}(B) = \text{List}(A)}{\text{Type} : B = A}
\]
Example: Equality version of the introduction rule for List (rule for nil is degenerated):

\[
N : \nu + \nu = \nu + \nu \\
N : \nu = \nu \\
N : \nu = \nu
\]

Example: Equality version of the elimination rule for (+), N:

\[
\forall \text{List} : (l', v) \text{cons} = (l, v) \text{cons} \\
\forall \text{List} : l = l \\
\forall : \nu = \nu
\]

\[
\forall \text{List} : \text{nil} = \text{nil} \\
\forall : \text{Type}
\]
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).

Equality Versions of Rules (Cont.)
(b) The Non-Dependent Function Type and Product

**Rules of the Non-Dependent Product**

- **Formation Rule**
  \[
  A : \text{Type} \quad \frac{}{A \times B : \text{Type}}
  \]

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

- **Elimination Rules**
  \[
  c : A \times B \quad \frac{}{\pi_0(c) : A}
  \]

- **Equality Rules**
  \[
  a : A \quad b : B \quad \frac{}{\pi_1((a, b)) = b : B}
  \]

(C) Anton Setzer 2003 (except for pictures)
The $\eta$-Rule

This rule does not fit into the above schema:

\[
\frac{c : A \times B}{c = \langle \pi_0(c), \pi_1(c) \rangle : A \times B}
\]

($\eta =$ greek letter spelled "eta")
The \( \nu \)-rule expresses that any element of \( \mathcal{B} \times \mathcal{A} \) is of the form \( \nu \theta \).

The \( \nu \)-rule (cont.)
Forelements of $A \times B$ introduced by an introduction rule, we don’t need the $\forall$-rule (cont.).

However, if we assume an element of type $A \times B$, e.g. state $x : A \times B$ (so $x$ is just a variable), we cannot derie that $x = \eta^1 \nu x \eta^0 \nu$ without making use of the $\forall$-rule.

$B \times A : x \iff B \times A : x$
Equality Version of the Formation Rule

\[ \text{Type} \quad A = A \]

\[ \text{Type} \quad B = B \]

\[ A \times B : \rho = \sigma \]

Equality Version of the Introduction Rule

\[ \begin{align*}
B : (\rho)T^T & = (\sigma)T^T \\
A : (\rho)T^0 & = (\sigma)T^0
\end{align*} \]

Equality Version of the Elimination Rules

\[ B \times A : \langle q', q \rangle = \langle q', q \rangle \\
B : q = q \quad A : p = p \]

Equality Version of the Formation Rule

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

Equality Version of the Introduction Rule

\[ \text{Type} \quad B = B \]

Equality Version of the Elimination Rule

\[ \text{Type} \quad A = A \]

(c) Anton Setzer 2003 (except for pictures)
the term \( a \) (after some renaming of bounded variables).

Here \([a := x] q\) is the result of substituting in \( q \) every occurrence of variable \( x \) by \( \beta \nvdash:\; [a := x] q = a (q (\forall x) x) \) 

\[ \frac{\forall \alpha \beta \vdash \alpha}{\forall \alpha \beta \vdash \beta \leftarrow \forall \alpha \beta \vdash \forall \alpha} \]

Equality Rule

\[ \frac{\forall \alpha \beta \vdash \beta \leftarrow \alpha}{\forall \alpha \beta \vdash \alpha \leftarrow \forall \alpha \beta \vdash \beta} \]

Elimination Rule

\[ \frac{\forall \alpha \beta \vdash q (\forall x) x}{\forall \alpha \beta \vdash \forall \alpha \beta \vdash \forall \alpha \beta \vdash \forall \alpha \beta \vdash \forall \alpha \beta} \]

Introduction Rule

\[ \frac{\forall \alpha \beta \vdash \forall \alpha \beta \vdash \forall \alpha \beta \vdash \forall \alpha \beta \vdash \forall \alpha \beta \vdash \forall \alpha \beta} \]

Formation Rule
The reduction corresponding to the equality rule is often called $\beta$-reduction.

\[
[q \vdash A \Rightarrow b] \rightarrow (\forall x q(x) \Rightarrow b)
\]

As a reduction, it reads:

- $\beta$ = Greek letter spelled "beta".
- $\beta$-Reduction.
Again this rule does not fit into the above schema:

\[ \frac{\forall x : f \cdot (\forall : x) \chi = f}{\forall x : f} \]

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

\[
\forall (\forall : x) \chi = \forall (\forall : x) \chi \\
[x =: x] \forall (\forall : x) \chi = \forall (\forall : x) \chi \\
(x (\forall (\forall : x) \chi)) \forall (\forall : x) \chi = x \forall (\forall : x) \chi
\]

we get

\[
\forall (\forall : x) \chi = f \quad \text{if we have } f \text{ of this form, e.g.} \quad x f = \text{something}
\]

\[
\forall (\forall : x) \chi \quad \text{if we have the } \forall \text{-rule, then this follows with}
\]

\[
\text{The } \forall \text{-rule expresses that any element of } B \text{ is of the form} \quad B \leftarrow \forall (\forall : x) \chi
\]
For elements of type $A \leftarrow \forall : f$ introduced by an introduction rule, we don’t need the $\forall$-rule. For elements of type $B \leftarrow \forall : f$ introduced by an introduction rule, we don’t need the $\forall$-rule.

However, if we assume an element of type $B$, e.g. state $f : A \not\rightarrow B$ (so $f$ is just a variable), we cannot derive that $\forall x \in F(x) \chi = f$ without making use of the $\forall$-rule.

Therefore, we don’t need the $\forall$-rule.
Equality Version of the Formation Rule

\[ B \vdash \iota, p = q \vdash f = f \]

\[ \forall \vdash \iota, p = q \vdash f = f \]

Equality Version of the Elimination Rule

\[ B \leftarrow \forall \vdash \iota, (\forall \vdash x) \chi = q \vdash (\forall \vdash x) \chi \]

\[ B \vdash \iota, q = q \leftarrow \forall \vdash x \]

Equality Version of the Introduction Rule

\[ \forall \vdash \iota, p \vdash B = B \leftarrow \forall \vdash \iota, p \vdash \forall = \forall \]

Equality Version of the Formation Rule

Equality Versions of the Rules
The introduction rule requires an extra premise which is not implied by the other premises.

**Introduction Rule**

\[
\frac{B \times (\forall x : q) \vdash B : q}{[q =: x]B : q}
\]

**Formation Rule**

**Rules of the Dependent Product** (c) The Dependent Function Type and Product

(©) Anton Setzer 2003 (except for pictures)
In the last introduction rule, an extra premise was required.

```
Extra Premise in the Introduction Rule
```

From which it follows $\forall x : \text{Type} \rightarrow B \times A : \text{Type}$ and $a : \forall x : \text{Type} \rightarrow B \times A : \text{Type}$.

- In case of the non-dependent product, this premise was not necessary:

```
\begin{align*}
(x : \forall x : A \rightarrow B) \\
\forall q : B \times A. (x : A \rightarrow B) \\
\end{align*}
```

- This is required in order to guarantee that we can form the type.

In the last introduction rule, an extra premise was required.
is not implied by the premises).

Note that the last two rules require the extra premise (which is not implied by the premises).

**Equality Rules**

\[
[v =: x]B : q = (\langle q, v \rangle)_{\top}\nu
\]

\[
[v =: x]B : q \quad V : \nu \quad \text{Type } \nu : B \iff V : x
\]

\[
V : \nu = (\langle q, v \rangle)_{\bot}\nu
\]

\[
[v =: x]B : q \quad V : \nu \quad \text{Type } \nu : B \iff V : x
\]

**Elimination Rules**

\[
[(\nu)_{\top}\nu =: x]B : (\nu)_{\top}\nu
\]

\[
B \times (V : x) : \nu
\]

\[
V : (\nu)_{\bot}\nu
\]

\[
B \times (V : x) : \nu
\]
Again the \( \iota \)-rule cannot be derived if the element in question is a variable.

\[
\begin{align*}
C \times (\forall x : B) & : ((\chi) \uparrow \varphi, (\chi)^0 \varphi) = \varphi \\
\Downarrow & \\
B \times (\forall x : C) & : \varphi
\end{align*}
\]

We have the following \( \iota \)-rule:

\textbf{Rules of the Dependent Product (Cont.)}
Equality Version of the Formation Rule

\[
\begin{align*}
(c)^0 \nu &=: x \beta : (\rho)^\top \nu = (c)^\top \nu & \forall : (\rho)^0 \nu = (c)^0 \nu \\
\beta \times (\forall : x) ; \rho &= \epsilon & \beta \times (\forall : x) ; \rho &= \epsilon
\end{align*}
\]

Equality Version of the Elimination Rules

\[
\begin{align*}
\beta \times (\forall : x) ; \langle q , p \rangle &= \langle q , p \rangle & \forall : \rho = \rho & \forall \text{Type} : \beta \Leftarrow \forall : x \\
\end{align*}
\]

Equality Version of the Introduction Rule

\[
\begin{align*}
\forall \text{Type} : \beta \times (\forall : x) &= \beta \times (\forall : x) & \forall \text{Type} : \beta = \beta & \forall \text{Type} : \forall = \forall \\
\end{align*}
\]
The Non-Dependent Product as an Abbreviation

A B

Cannowbeseenasan abbreviation for

\( (\forall x : A) B \times x \)

Taking A 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 A B can now be seen as an abbreviation for

The Non-Dependent Product as an Abbreviation
The Non-Dependent Product as an Abreviation

More precisely we will see:

- Similarly for the elimination, equality and u-rule.
- Product imply those of the dependent product.
- Therefore the premises of the introduction rule for the non-dependent product imply those of the formation rule for the non-dependent product.

\[ \text{From a derivation of } a : A \text{ we can derive } A : \text{Type} \]
\[ \text{We need the concept of presupposition for that.} \]

\[ \text{Therefore the premises of the formation rule for the non-dependent product imply those of the dependent product.} \]

\[ \text{This requires the weakening rule, which will be introduced later.} \]

\[ \text{From } A : \text{Type} \text{ and } B : \text{Type} \text{ we can derive } x : \text{Type}. \]
In Agda, we have the dependent record type.

\{ B :: q :: A :: a :: \forall \forall :: D :: \text{Type} \}

Then we can introduce

Assume we have introduced already \forall \forall :: A :: \text{Type}.

- It is essentially a "labelled product".

- In Agda, we have the dependent record type.
If we have $a::A$, $b::B[a := a]$, then we can introduce

```
\text{sig}\ a\ q\ c\ e :\ \mathbf{E}
{\text{A}\ q\ c\ e :\ \mathbf{B}}
\{\ q = q' ; \ p = p' \} \text{strict}$
```

\text{D} :: \{ q = q' ; p = p' \}$

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

```
\text{sig}\ f :: A ; b :: B ; c :: C ; e :: E
```

If we have $c :: A$, $q :: B[a := a]$, then we can introduce

```
The Dependent Product in Agda (Cont.)
```
We can now project any element of $\mathcal{Q}$ as above down to $\mathcal{A}$ and $\mathcal{B}$.

\[
\begin{align*}
\forall \mathcal{A}, \mathcal{B} :& \quad \mathcal{Q} \ni q, \mathcal{P} \ni p \Rightarrow q = q' \land p = p' \\
\forall \mathcal{A} :& \quad \mathcal{A} \ni q, \mathcal{P} \ni p \Rightarrow q \circ q = q' \circ p
\end{align*}
\]
Unfortunately, the dependent product does not behave very well.

The Dependent Product in Agda (Cont.)

In this setting \( \mathfrak{u} \)-equality asserts that if

\[
\{q \in \mathfrak{u} \mid \forall a. \exists c. q = c \} \quad \text{then}
\]

\[
\{ (a) \in \mathfrak{u} \mid \forall a. \exists c. a \in c \} \\
\]

which will be treated later.

In most cases one can avoid this, by using the inductively defined \( \mathfrak{u} \)-type,

\[
\{ q = c. a \mid q \in \mathfrak{u} \} \\
\]

In this setting \( \mathfrak{u} \)-equality asserts that if

This is due to the fact that Agda doesn’t support the \( \mathfrak{u} \)-rule.

Unfortuantely, the dependent product does not behave very well.

\[
\text{\textcopyright} Antton Setzer 2003 (except for pictures) \]
Rules of the Dependent Function Type

\[
\begin{align*}
\frac{[v := x]B : [v := x]q = \nu (\forall x : \text{Type})}{\forall \nu : B : q} \iff \forall \nu : \text{Type}
\end{align*}
\]

Equality Rule

\[
\frac{[v := x]B : \nu f}{\forall \nu : B \leftarrow (\forall x : \text{Type}) : f}
\]

Elimination Rule

\[
\frac{B \leftarrow (\forall x : q (\forall x : \text{Type}) \nu)}{B : q} \iff \forall \nu : \text{Type}
\]

Introduction Rule

\[
\frac{\text{Type} : B \leftarrow (\forall x : \text{Type})}{\text{Type} : B \leftarrow \forall x : \text{Type} : \text{Type}}
\]

Formation Rule
The \( \exists \)-Rule

\[ f : (x : A) \not B \]

\[ f = (x : A) : (x : A) \not B \]

Again the \( \exists \)-Rule expresses that every element of \( B \) is of the form \( (\forall : x)(\forall : x) \) something.

\[ (\forall : x) : x f \cdot (\forall : x) \chi = f \]

\[ B \leftarrow (\forall : x) : f \]

\[ \exists \text{-Rule} \]

The \( \exists \)-Rule has a special status:
Further terms which differ in the choice of bounded variables are identified:

- Called $\alpha$-equivalence ($\alpha = \text{greek letter spelled alpha}$).
- A similar rule applies to bounded variables in types.
  - $\forall : \alpha(x : A) \equiv x + x$.
  - $\exists : \alpha(x : A) \equiv x + x$.
  - $\forall : \alpha(x : N) \equiv x + x$.
  - $\exists : \alpha(x : N) \equiv x + x$.

$\alpha$-Equivalence
Equality Version of the Elimination Rule

\[
\frac{[v := x]B : p f = v f}{\forall : p = v \quad B \leftarrow (\forall : x) : f = f}
\]

Equality Version of the Introduction Rule

\[
B \leftarrow (\forall : x) : q \cdot (\forall : x) \chi = q \cdot (\forall : x) \chi
\]

Equality Version of the Formation Rule

\[
\frac{\text{Type} : \chi}{\text{Type} : \chi \leftarrow \chi \leftarrow A}
\]

Equality Versions of the above Rules

(C) Anton Setzer 2003 (except for pictures)
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 $x$.

The non-dependent function type is the relationship between non-dependent and dependent function type.
In Agda one writes \((\mathbf{x}::\mathbf{A}) \to \mathbf{C}\) for the nondependent function type.

\[\mathbf{I} :: \mathbf{C} \leftarrow (\mathbf{x}::\mathbf{A})\]

The above introduces

\[\cdots = \mathbf{C} :: \mathbf{I}\]

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

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

- We can write

\[\mathbf{f} :: \mathbf{I}::\mathbf{C}\]

\[\text{Requires the \ldots is an element of type } \mathbf{C}, \text{ possibly making use of } \mathbf{x}.
\]

\[\text{The above introduces} \]

\[\text{\ldots} \]

\[\text{\ldots} \]

\[\text{\ldots} \]

\[\text{\ldots} \]
Alternatively, one can use the \( \forall \)-notation:

\[
\forall x : A \Rightarrow C \Rightarrow (x : A) \Rightarrow C
\]

\[\cdots \leftarrow \forall x : A \Rightarrow C \]
The example is better introduced using the first notation.

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:

\[
\text{map} \ (\lambda (x:N) \to \text{const two (cons three nill)}) \ (\text{const two (cons three nill)})
\]

The result would be:

\[
\text{const three (cons four nill)}
\]
\[\cdots \leftarrow (N::\forall) \leftarrow (N::\forall) \]

instead of

\[\cdots \leftarrow (N::\forall) \]

Similarly we can write

\[(\forall,\forall) \leftarrow (N::\forall) \leftarrow (N::\forall)\]

instead of

\[(\forall,\forall) \leftarrow (N::\forall,\forall)\]

We can write

Abbreviations

(c) Anton Setzer 2003 (except for pictures)
Similarly we can write

```
\cdots =
N ::
(N : w)
(N : u) \not\in
```

instead of

```
\cdots =
N ::
(N : w, u) \not\in
```

Abbreviations (cont.)
The Dependent Function Type in Agda (Cont.)

- **Application** has the same syntax as in the rules above:

If we have
\[
\begin{align*}
& f :: (x :: A) \rightarrow B, \\
& a :: A, \\
& a :: B[x := a]
\end{align*}
\]

then we can conclude:
\[
\begin{align*}
& f a :: B[x := a] \\
& (\lambda (x :: A) \rightarrow b) a \\
& b[x := a]
\end{align*}
\]

And we have that
- **are identified.**
In Agda syntax, the \( \text{u}-\text{rule} \) would state that if

\[
\text{\( \forall x :: A \) \( \text{\( \exists \)} x \)}
\]

then

\[
\text{\( \exists (x :: A) B \)}
\]
Let $G$ be the set of genders, $G = \{ \text{male}; \text{female} \}$. 

- $\text{Names(\text{female})} = \{ \text{Sara}, \text{Jill} \}$
- $\text{Names(\text{male})} = \{ \text{Tom}, \text{Jim} \}$

E.g.

Let for $g$ : $G$ the type $\text{Names}(g)$ be the collection of names of that gender.

$\{ \text{male}; \text{female} \} = \{ \text{Tom}, \text{Jim}; \text{Sara} \}$

Example of the Use of Dependent Products

Let $G$ be the set of genders.

(c) Anton Setzer 2003 (except for pictures)
Example of the Use of Dependent Products (cont.)

Now the set of names is the set of pairs \( \langle n, \vec{g} \rangle \) s.t. \( \vec{g} \) is a Gender and

This type is written as

\[
(\vec{g} : \text{Names}(\vec{g})) \times (G : \vec{b})
\]
Although we haven’t introduced yet a notation for algebraic data types, the following is readable for those familiar with Haskell:

The “Names”-Example in Agda

```agda
{\text{u \ Names G :: \ \{} \n  \text{g :: G, } \text{c :: C' } \text{) \text{sig} =}
  \text{Type :: \text{AllNames}}

\text{}\{ \text{female} \leftarrow \text{data Jill} | \text{Sara};
  \text{male} \leftarrow \text{data Tom} | \text{Jim}; \}
\text{case G of}
  \text{Type ::}
  \text{AllNames (G :: G)}
  \text{female | male =}
  \text{Type ::}
  \text{G}
```

The following example illustrates how to define a type `Names` representing the names of male and female individuals. We define two data constructors: `male` and `female`, each associated with specific names, and a type constructor `AllNames` that takes a gender and a name as arguments. The type `G` is used to select the appropriate data constructor based on the gender.
Example of the Dependent Function Type

\[
\text{De}\vphantom{\alpha}\text{f
\hspace{1em}}
\text{select}(g : G) \rightarrow \text{Names}(g)
\]

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

It wouldn't make sense to say \( \text{select male} : B(\text{male}) \rightarrow B(x) \).

\[
[x](x)B = (\text{make of } B(x)) : \text{male}
\]

Select male will be an element of \( B(x) \).

It wouldn't make sense to say \( \text{select male} : B(x) \).

Select selects for every gender a name:

\[
\begin{align*}
\text{select(\text{male})} & = J\hspace{1em}\\
\text{select(\text{female})} & = T\hspace{1em}\\
(\text{Names}(\text{male}) \leftarrow (g : b)) : & = \hspace{1em} \text{select}
\end{align*}
\]

Define
As before, here is the code for the select example, which should be readable.

The `select`-Name Example in Agda

```
{ { female \to \text{Jim} \at \text{data Tom} \at \text{male} } \at (\text{male}) } \at \text{male} = \text{Name} = (G :: b) :: \text{select}

{ { female \to \text{Jill} \at \text{data Sara} \at \text{female} } \at (\text{female}) } \at \text{female} = \text{Name} = (G :: b) :: \text{select}

\text{G} \at \text{female} \at \text{data male} = \text{G} \at \text{female} \at \text{data male} = \text{G}
```

For those familiar with Haskell:

- As before, here is the code for the select example, which should be readable.
The convention is that all rules can as well be weakened by a common context.

This means that when introducing a rule

\[ \theta \leftarrow \bigcup_{\substack{u \mid \forall x : u \forall : \forall x \cdots \forall \forall : \forall x}} \]

\[ u \theta \leftarrow \bigcup_{\substack{u \mid \forall x : u \forall : \forall x \cdots \forall \forall : \forall x}} \bigcup_{\substack{1 \mid \forall x \cdots \forall x : \forall x}} \]

(for any choice of \( u, x \)).

We implicitly introduce as well the following rules:

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

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

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

\[ x_1 : A_1; \ldots; x_n : A_n \]
Example
Consider the sample derivation (assuming Type : Type):

\[
\begin{align*}
\Gamma & \vdash \text{\textit{x}} : \text{\textit{A}} \\
\Gamma & \vdash \text{\textit{y}} : \text{\textit{A}} \\
\Gamma & \vdash \text{\textit{x}} : \text{\textit{A}} \\
\Gamma & \vdash \text{\textit{y}} : \text{\textit{A}} \\
\end{align*}
\]

The second rule used is the rule for \( \land \)-introduction without any weakening.

The first rule used is the rule for \( \land \)-introduction weakened by the context \( \text{\textit{x}} : \text{\textit{A}} \).

The second rule used is the rule for \( \land \)-introduction weakened by the context.
Bound Variables in Common Contexts

The only side condition is that, if the rule introduces a new variable, it must not occur in the context.

For instance in the weakened form of the $\forall$-rule:

\[
B \leftarrow \forall : x \ f \cdot (\forall : x) \ \chi = f \leftarrow u \forall : u x, \ldots, u x \ \forall : x
\]

- This is since in this lecture we want that variables bound in any context above is implicitly in the context $x f$ * are different.
- $x$ must be different from $x 1, \ldots, x n$.

(B) Anton Setzer 2003 (except for pictures)
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 $x^n : A^n \uprod \cdots \uprod x^1 : A^1$, \(\vdash\) Context.

\[ \text{Context} \]

\[ \text{Will be discussed later.} \]

\[ \text{If we have an axiom, we need to be sure that the context, we weakened with,} \]

Weakening of Axioms

\[ \text{will be discussed later.} \]
Weakening of Axioms

For the moment, we mention how the formation rule for \( \mathbb{N} \) can be weakened:

\[
\text{Context} \quad \frac{\forall x : \mathbb{N}, \ldots, \forall x_i : \mathbb{N}}{\forall \mathbb{N} : \text{Type}}
\]

More about this later.
Let expressions in Agda allow to introduce temporary variables using "let-expressions".

\[
\begin{align*}
\text{in} & \quad t \\
\up s & = \\
\forall \quad u \quad \exists \quad u \quad \vdash \\
\ldots & \\
\exists \quad s & = \\
\forall \quad \exists \quad s & = \\
\forall \quad \forall \quad s & = \\
\text{let} \quad a \quad \forall \quad \exists \quad s & = \\
\end{align*}
\]
Let expressions in Agda (Cont.)

This means that we introduce new constants $a_1, \ldots, a_n$ of types $A_1, \ldots, A_n$ respectively, and can then use them.

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

Let expressions in Agda (Cont.)
Let expressions in Agda (Cont.)

If we are in a goal, we can use the command

\begin{verbatim}
let expression in Agda (Cont.)
\end{verbatim}

- Agda will construct a template of the form:
- We have to write down the variables, separated by a blank.

\begin{verbatim}
{\textit{i}} \textit{i} = \\
{\textit{i}} \textit{i} :: \textit{ud} \\
\ldots \\
{\textit{i}} \textit{i} = \\
{\textit{i}} \textit{i} :: \textit{ud} \\
{\textit{i}} \textit{i} = \\
{\textit{i}} \textit{i} :: \textit{ud} \\
\end{verbatim}

(C) Anton Setzer 2003, except for pictures
Example of let expressions

Here follows a simple concrete example, which computes \((n + n)\) for natural numbers \(n\).

\[
\begin{align*}
\text{let } m :: N = & \quad (N :: u) \ f \\
\text{in } m * m = & \quad n + u = \\
N :: m = & \quad \text{let } m = \\
N :: & \quad \text{for } (n + n) \ast (n + n)
\end{align*}
\]
In this subsection we look at the relationship between Agda code and the corresponding derivations. We consider various examples.

Following each step in the development of the Agda code, then we will look at how the corresponding derivations are developed.

First we will go through the development of the Agda code.

In this subsection we look at the relationship between Agda code and the corresponding derivations.
Example 1

We want to derive in Agda

\( \forall (a : A) . a : A \leftarrow A \)

Step 1:

See example file exampleIdentity.agda

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

postulate (i.e. assume) one type \( A \):

-- Since we want to have the definition for an arbitrary type \( A \), we
-- We need to introduce the type \( A \) first.

\[ \forall (a : A) . a : A \text{.} \]
Step 2: We state our goal:

\[
\{ i \mid i \} = V \leftarrow V : \neg f
\]

Example I (Cont.)
Step 3:

We want to derive an element of function type $A$. We have:

$$
\{ i \ i \} \leftarrow (\{ i \ i \ :: \ y \} \, y \ = \\
A \leftarrow A :: f
$$

After executing it we get:

- Has to be executed while the cursor is inside one goal.
- Aghda has a command `agda-intro(Intro)` which does this step automatically.

The precise Agda code uses `\` instead of `$` and `\textless` instead of `$$.
Step 4:

The first goal, the type of the variable \( h \) can be solved automatically.

We obtain:

Use \texttt{agda-solve (solve)}

- \{ i \} \leftarrow (\forall : \eta) \forall = \forall \leftarrow \forall :: f
Step 4 (Cont.)
{ i \ i } \leftarrow (\forall \ a)(\forall = \ V \leftarrow V :: f

- We obtain:

Otherwise the changes will not be known by Agda.

agda-load-buffer (load Buffer)

At the end of any editing one should execute:
and can edit everything.

Then one is in a mode where the goals are converted to ordinary symbols

agda-restart (Re)start Agda

If one wants to edit parts involving goals, one first has to execute:

If we can always edit the current code.

This can be done by simple editing.

It is a good idea to rename the variable to something, for instance to a:

Example 1 (Cont.)

Example 1 (Cont.)

Step 5:

\[
\{ i \mid A \} \leftarrow \{ i \mid A \} \leftarrow \{ \forall \ A : (A :: a) \mid A \}
\]

- In order for \( \forall \ A : (A :: a) \mid A \),

\[ \text{This can be inspected by using the menu} \]

\text{agda-goal-type-of-meta-reduced (Type of goal (unfolded))},

\[ \text{which shows the type of the current goal.} \]

\[ \text{which shows type A of the current goal.} \]

\[ \text{Has to be executed while the cursor is inside one goal.} \]

\[ \text{so the type of the goal is A.} \]

\[ \text{A \leftarrow A, which means it is a function of type A \rightarrow A.} \]

\[ \text{Then this A-term computes an element of type A depending on some a} \]

\[ \text{must be of type A.} \]

\[ \text{It shows A.} \]
Step 5 (Cont.)

- We can inspect the context.

- The context contains everything we can use when solving our goal. It contains:
  - A :: Type
  - f :: A → A
  - a :: A

Since we are defining an element of type A depending on a :: A, we can use a.

See next slide:

\[
A \leftarrow A :: f \quad \ast \quad A :: \text{Type} :: \ast
\]
On the last slide we had $\forall \in \text{ the context}$. 

Termination Check

For the type checker a definition $q :: \forall q = \forall$ would be legal, although evaluating $q$ doesn't terminate (black hole recursion).

This appears, since the type checker allows to define functions recursively independently of whether the recursion terminates or not.
Termination Check (Cont.)

Agda has a command \texttt{agda-term-check-buer} (Check Termination), which checks whether recursive definitions are done properly. One cannot write a universal termination checker, since the Turing Halting Problem is undecidable.

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

If the termination check fails, it might still be the case that all programs terminate.

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

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

(On one cannot write a universal termination checker, since the Turing Halting Problem is undecidable).
and are done.

\[
v \leftarrow (\forall :: a)(\forall = \forall \leftarrow \forall :: f)
\]

• Example 1 (Cont.)

(C) Anton Setzer 2003 (except for pictures)
In Agda step 1 we postulated \( A : \text{Type} \).

In Agda step 2 we stated our goal:

\[
V \leftarrow V : {}^0p
\]

We write for this something of type \( {}^0p \) and get as conclusion of our derivation:

\[
V \leftarrow V : {}^nA
\]

In terms of rules this means that we want to derive something of type \( {}^nA \).

{\{i \ i\} =
V \leftarrow V :: f
\]

This corresponds in the rule system, that we can assume \( A : \text{Type} \), i.e. can write this down without any derivation.

Example 1, Using Rules
Example 1, Using Rules (Cont.)

In Agda step 3 and 4 we replaced \(:\{i \quad i\} \leftarrow (\forall :: v)\chi\) by \:\{i \quad i\} which is derived by an introduction rule.

\[
\begin{align*}
\forall & \leftarrow \forall : \mathcal{I} p (\forall : v)\chi \\
\forall : \mathcal{I} p & \leftarrow \forall : v \\
\end{align*}
\]

In terms of rules this means that we replace \(p^0\) by \(\mathcal{I} p (\forall : v)\chi\) which is derived by an introduction rule.
Example 1, Using Rules (Cont.)

In Agda step 5 we replace \{i \ i\} \to (\forall :: v) \chi \text{ in } \{i \ i\} by \chi \text{ by } a.

\[
\begin{align*}
\forall &\to (\forall :: v) \chi = \\
V &\leftarrow V :: \phi
\end{align*}
\]

\text{The assumption rule will be discussed later.}

- Essentially it allows to derive \( \forall x \colon B \) occurs in the context that \( x \colon B \) holds.

\[
\begin{array}{c}
\forall \vdash \forall \colon (\forall :: v) \chi \\
\forall \vdash \forall \equiv \forall
\end{array}
\]

\text{In terms of rules this means that we replace } \frac{\forall a}{(\forall :: v) \chi} \text{ by } a.
Example 2

We consider a derivation of

\( \{ i \ i \} = V \leftarrow (V \leftarrow (V \leftarrow V)) :: f \)

Step 1:

We state our goal:

- Postulate \( A :: \text{Type} \)

- We postulate \( A :: \text{Type} \)

Step 1:

See example `exampleExamples/Example2.agda`

\[ V \leftarrow (V \leftarrow (V \leftarrow V)) :: (a \leftarrow (a :: (a :: (a :: x \cdot x \cdot x) \leftarrow (a \leftarrow (a \leftarrow a)) :: x)) \cdot x) \cdot x \ dereivation \\
\]

We consider a derivation of
Step 2:

Using \texttt{agda-solve} (\texttt{Solve}) we obtain:

\[
\{ i \; i \} \leftarrow (V \leftarrow (V \leftarrow V) :: y) \; y =
V \leftarrow (V \leftarrow (V \leftarrow V)) :: f
\]

We obtain:

- The type of the goal is a function type, and we can use \texttt{agda-intro (Intro)}:

\[
\{ i \; i \} \leftarrow (\{ i \; i \} :: y) \; y =
V \leftarrow (V \leftarrow (V \leftarrow V)) :: f
\]

Example 2 (Cont.)
Step 2 (Cont.):

\[
\{i \ i\} \leftarrow (\forall \leftarrow (\forall \leftarrow \forall) \:: x) \forall = \\
\forall \leftarrow (\forall \leftarrow (\forall \leftarrow \forall)) \:: f
\]

so that Agda realizes this change:

We rename the variable \( y \) to \( x \) and use Agda-load-buffer (load buffer)
Step 3:

We obtain

\[ \{i \ i\} x \leftarrow (\forall \ (\forall \ x) : x) \quad \forall \ (\forall \ (\forall \ x) : x) : f \]

\[ \text{In our case it is one (of type } \forall x \rightarrow \forall ) \text{.} \]

\[ \text{Agda will then apply } x \text{ to as many goals as needed in order to obtain an element of the desired type.} \]

\[ \text{Therefore we type into the goal } x \text{ and use } \text{agda-rename (Reform).} \]

\[ \text{The context contains } f \text{ (for recursive definitions), } x \text{ and } x. \]

\[ \text{The type of the new goal is } \forall, \text{ which is the result type of the function we are defining.} \]

Example 2 (cont.)
Step 4:

The type of the new goal is \( \forall. \)

- The type of the new goal is \( \forall. \)

- \( f \) :: \( \forall \left( \forall \left( \forall \left( \forall \right) : x \right) \gamma = \forall \left( \forall \left( \forall \right) \right) \cdot f \)

**Example 2 (Cont.)**

We try \texttt{agda-intro} \texttt{Intro} \texttt{Intro} and obtain:

- \( \forall \) in order to obtain an element of type \( \forall. \)
- Since \( x \) needs to be applied to an element of type \( \forall. \)

\( \{ i \ i \} \leftarrow (\{ i \ i \} :: \eta) \gamma \ x \leftarrow (\forall \left( \forall \left( \forall \right) :: x \right) \gamma = \forall \left( \forall \left( \forall \right) \right) : f \)
Using \texttt{agda-solve} (solve) \ we \ obtain:\n
\begin{align*}
\{i \gets (\forall \mathbf{y})(\exists x) & (V \leftarrow (V \leftarrow (V \leftarrow V)) \leftarrow \mathbf{x}) \mathbf{y} = \\varepsilon \}
\end{align*}

\textbf{Step 4 (Cont.)}
\[
\{i \cdot i\} \leftarrow (\forall : v) x \leftarrow (\forall \leftarrow (\forall \leftarrow \forall) :: x)\gamma = \\
\forall \leftarrow (\forall \leftarrow (\forall \leftarrow \forall)) :: f
\]

- We rename \( y \) by \( a \), reload the buffer, and obtain:

Step 4 (Cont.)
Step 5

Thenewgoalhastype

\[ A \]

The complete expression

\((a :: A) \backslash f \backslash g\)
should havetype

\[ A \]

so

\(f \backslash g\)
must havetype

\[ A \]

Thecontextcontains

\(A :: Type\), \(f\), \(x\) and \(a\).

ThereisausuallymorethanonesolutionforproceedinginAgda.

We can use both \(x\) and \(a\) here.

The context contains \(A :: \Lambda\) \(y\) \(f\), \(x\) and \(a\).

so

\(\{ i \mid i \}\) should havetype \(A\).

The complete expression \(\Lambda(x :: a)(A :: A)\)
should havetype \(A\).

We try \(a :: A\). After inserting it and using agda-refine (Refine) we obtain

\(x \leftarrow (A :: a)(A) \backslash x \leftarrow (A \leftarrow (A \leftarrow A) :: x) \Lambda =\)

\(A \leftarrow (A \leftarrow (A \leftarrow A)) \Lambda \Lambda f\)

the following and are done:

- We try \(a :: A\). After inserting it and using agda-refine (Refine) we obtain

This means that we sometimes have to backtrack and try a different solution.

- There is usually more than one solution for proceeding in Agda.

Example 2 (Cont.)
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:

$$A \leftarrow (A \leftarrow (A \leftarrow A)) : \beta$$
Example 2, Using Rules

\[ V \leftarrow (V \leftarrow (V \leftarrow V)) : \text{\( \vdash \)} p \cdot (V \leftarrow (V \leftarrow V) : x) \\vdash \]
\[ V : \text{\( \vdash \)} p \leftarrow V \leftarrow (V \leftarrow V) : x \]

- This corresponds to replacing \( d \) by \( \{ i \ \vdash \ (V \leftarrow (V \leftarrow V) : x) \} \)
- The next step in the Agrda-derivation was to replace the goal by

```
Example 2, Using Rules
```

Example 2, Using Rules

The left top judgement can be derived by an assumption rule (more about this later).

\[
\frac{A \leftarrow (A \leftarrow (A \leftarrow A)) : \exists \, x \cdot (A \leftarrow (A \leftarrow A) : x) \leftarrow \exists \, x \cdot (A \leftarrow (A \leftarrow A) : x) \leftarrow (A \leftarrow A) : x \leftarrow (A \leftarrow A) : x}{A \leftarrow A : \exists \, x \cdot (A \leftarrow (A \leftarrow A) : x} \leftarrow (A \leftarrow A) : x \leftarrow (A \leftarrow A) : x
\]

This corresponds to replacing \( \exists \, x \cdot (A \leftarrow (A \leftarrow A) : x) \) by \( x \). This was replaced by \( \exists \, x \cdot (A \leftarrow (A \leftarrow A) : x) \) by \( \exists \, x \cdot (A \leftarrow (A \leftarrow A) : x) \leftarrow (A \leftarrow A) : x \leftarrow (A \leftarrow A) : x \) in the Agda-derivation used refinement.

The next step in the Agda-derivation used refinement:
Example 2, Using Rules

We then used into on the goal which was then replaced by

\{ i \mapsto i \} \leftarrow (A :: x) \vDash

This corresponds to replacing \( \exists p' \)(A : v) \vDash by \( \exists p \)(A : x) \vDash which can be introduced by

\[
\frac{V \leftarrow (V \leftarrow (V \leftarrow V)) : (\exists p' (V : v) V) x \cdot (V \leftarrow (V \leftarrow V) : x) V}{V : \exists p \leftarrow V : v, V \leftarrow (V \leftarrow V) : x}
\]

an introduction rule:
Example 2, Using Rules

Finally we used refinement, which replaced the goal by a.

This corresponds to replacing $d_3$ by $a$.

The right-hand derivation can again be derived by an assumption rule (more about this later).

\[
\frac{V \leftarrow (V \leftarrow (V \leftarrow V)) : (\nu \cdot (V : V)V) x \cdot (V \leftarrow (V \leftarrow V) : x)V}{V : (\nu \cdot (V : V)V) x \leftarrow V \leftarrow (V \leftarrow V) : x}
\]

\[
\frac{V \leftarrow V : \nu \cdot (V : V)V \leftarrow V \leftarrow (V \leftarrow V) : x}{V \leftarrow (V \leftarrow V) : x \leftarrow V \leftarrow (V \leftarrow V) : x}
\]

\[
\frac{V \leftarrow V : \nu \cdot (V : V)V \leftarrow (V \leftarrow V) : x}{V \leftarrow (V \leftarrow V) : x}
\]

(C) Anton Sezer 2003 (except for pictures)
We derive an element of type $A \rightarrow B$ where $AB$ is the product of $A$ and $B$.

(See exampleProductIntro.agda).
Step 1:

We postulate types $A$, $B$:

$$\begin{align*}
\{ B :: q, \\
\forall A :: \text{sig} \} & = \\
AB :: \text{Type}
\end{align*}$$

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

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

$$\begin{align*}
\text{postulate } B :: \text{Type} \\
\text{postulate } A :: \text{Type}
\end{align*}$$

- We postulate types $A$, $B$:
Step 2: Our goal is:

$$\{ i \mid i \} = \{ \overline{B} \overline{A} \} \leftarrow B \overline{A} \leftarrow A \overline{B} :: f$$

Example 3 (Cont.)
Step 3: 

We use intro. A nelement of $A \times B$ will be of the form $\{ (a, b) \mid a \in A, b \in B \}$.

- We use intro.

Critical Systems, CS 411, Lenterm 2003, Sec. B2
\begin{align*}
\{i, i\} & \leftarrow (B :: A)\chi \leftarrow (V :: p)\chi = \\
AB & \leftarrow B \leftarrow V :: f
\end{align*}

After applying algebraic solve and renaming of variables we get

\begin{align*}
\{i, i\} & \leftarrow (\{i, i\} :: q)\chi \leftarrow (\{i, i\} :: q)\chi = \\
AB & \leftarrow B \leftarrow V :: f
\end{align*}

After applying into we get

Step 3 (Cont.)
Example 3 (Cont.)

Step 4:

The new goal is of type \( AB \) which is a record type.

Elements of type \( AB \) introduced by the introduction principle will have the form.

\[
\{ \{ i \ i \} = q \\
\{ i \ i \} = v \} \text{ struct} \leftarrow (B :: q) \chi \leftarrow (A :: v) \chi = AB \leftarrow B \leftarrow A :: f
\]

When using intro we get:

\[
\{ \{ i \ i \} = q \\
\{ i \ i \} = v \} \text{ struct}
\]
Step 5:

- Using $a$ and $q$ would in our example result in non-termination.
  - $q$ and $p$ would be used recursively.

$\begin{align*}
  A & : \text{Type} \\
  B & : \text{Type} \\
  f & : A \to B \\
  a_0 & : A \\
  b_0 & : B
\end{align*}$

- The first goal has as context:

Example 3 (Cont.)
{ ; { i \vdash i } = q \\
; \vdash p = p } \quad \text{struct} \leftarrow ( B \vdash p ) \chi \leftarrow ( \forall \vdash p ) \chi = \chi \\
AB \leftarrow B \leftarrow A \vdash f

Step 5 (cont.)

Example 3 (cont.)
Similarly, we can solve the second one:

Step 6:
The treatment of which will be delayed until later.

We won’t use this however, since it is required for the assumption rules only.

\[
\frac{\text{Type} : A 	imes B \text{ Type}}{A \text{ Type}, B \text{ Type}}
\]

which can be derived as follows:

The definition of $AB$ means that $AB$ abbreviates $A \times B$.

Example 3, Using Rules
Example 3, Using Rules (Cont.)

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

\[
(\exists x : A) \leftarrow (A \land B)
\]

Using into means that we replace \(d_0\) by \(d_1\) which is introduced by two introduction rules:

\[
\begin{align*}
(\exists x : A) &\leftarrow (A \land B) \\
&\leftarrow A : \forall x (B \land x) \\
\end{align*}
\]

\[
\begin{align*}
(\exists x : A) &\leftarrow (A \land B) \\
&\leftarrow B : \forall x (A \land x) \\
\end{align*}
\]

Critical Systems, CS441, Lent term 2003, Sec. B2
Using intro again means that we replace \( d \) by \( \langle p, q \rangle \), which can be introduced by an introduction rule:

\[
\begin{align*}
(B \times A) & \leftarrow B \leftarrow A : \langle \exists p, q \rangle \cdot (B : A) \chi \\
(B \times A) & \leftarrow B : \langle \exists p, q \rangle \cdot (B : A) \chi \leftarrow A : p \\
B \times A & \leftarrow B : A, A : q, A : p \\
B & \leftarrow B : A, A : q, A : p
\end{align*}
\]
Example 3, Using Rules (Cont.)

The premises require an assumption rule (which will use the derivation of $A \times B$), see later for details.

\[
\begin{align*}
(B \times V) & \leftarrow B \leftarrow V : \langle \rho, \pi \rangle \cdot (B : \rho) \forall (V : \pi) \\
(B \times V) & \leftarrow B : \langle \rho, \pi \rangle \cdot (B : \rho) \forall \leftarrow V : \pi \\
B & \leftarrow B : \rho \forall V : \pi \\
V : \pi & \leftarrow B : \rho \forall V : \pi
\end{align*}
\]

\[d_3 \text{ by } c:\]

Solving the goals by refining them with $\rho$, means that we replace $\rho$ by $\rho'$.
We derive an element of type $A \leftarrow B \leftarrow A \leftarrow B (A \leftarrow B C)$.

where $BC$ is the product of $B$ and $C$. (See example ProductElim.agda).
Step 1:

We postulate types $A$, $B$, $C$:

$\{ C :: C, B :: B, A :: A \}$

$BC :: TYPe$

$SIG \ f b :: B; c :: C\}$

We introduce the product of $B$, $C$:

postulate $C :: TYPe$

postulate $B :: TYPe$

postulate $A :: TYPe$

$- We\ postulate\ types\ A, B, C$.
Example 4 (Cont.)

Step 2:

- Our goal is:

\[
\{ i, i \} = B \leftarrow A \leftarrow (B \leftarrow AB \leftarrow A) :: f
\]
\{i \mid i\} \leftarrow (\forall :: v)(\forall \leftarrow (\forall :: x))\chi = (B \leftarrow A \leftarrow (B \leftarrow A)) :: f

Step 3:

Example 4 (Cont.)
Step 4:

The context has no element with result type \( B \) (except of \( f \) which results in a circular definition).

- However, \( x \) has function type with result type \( BC \), which can be projected in a let-expression.
- We introduce first an element of type \( BC \) by a let-expression, and then derive from it the desired element of type \( B \).
- Using \texttt{agda-let} (Make let expression) with variable \( bc \) we obtain:

\[
\begin{align*}
\{i \ i\} & \quad \text{in} \\
\{i \ i\} & = \\
\{i \ i\} :: \ & \text{let } \alpha \leftarrow (\forall : \alpha \forall x : \forall \beta \forall \gamma \exists z : \forall (\forall \beta \forall \gamma \beta) \gamma \alpha \beta \gamma \cdot f

[\text{Example 4 (Cont.}]
\]
Example 4 (Cont.)

Step 5:

We insert a type of variable $bc$ of type $BC$ (using refine) and obtain:

$$\{ i \mid i \} \text{ in } \{ i \mid i \} = \{ i \mid i \} \text{ in } \{ i \mid i \}$$

$$f :: (A \rightarrow BC) \rightarrow A \rightarrow BC = \lambda \text{ let } bc :: BC = f \rightarrow g \in f \rightarrow g$$
Step 6:

For solving the first goal (definition of \( \Phi \)) we can refine \( x \), which has as result type \( BC \).

\[
\{i \ i\} x = \{i \ i\} \in \mathcal{BC} \quad \text{let} \quad \lambda \in (\forall \ a : \mathcal{BC} \rightarrow \mathcal{V} : \mathcal{V}) \lambda = \begin{align*}
\mathcal{B} & \leftarrow \mathcal{V} \leftarrow (\mathcal{BC} \leftarrow \mathcal{V} : \mathcal{V} : \mathcal{B}) : \mathcal{f}
\end{align*}
\]

Example 4 (Cont.)
Step 7: The new goal can be solved by refining it with variable a:

Example 4 (Cont.)
Step 8:

Currently, Agda doesn’t have any direct support for refining `c` to an element of type `B`. We have to do this by hand, insert `bc`, choose refining and obtain:

```
\begin{verbatim}
in p.c.
q x =

bc :: c \rightarrow (a :: A ! BC) \rightarrow (x :: A) \rightarrow \forall \forall :: x) \forall =
B \leftarrow A \leftarrow (\forall \forall :: (B) \leftarrow (A) :: f
\end{verbatim}
```

Example 4 (Cont.)
In our rule calculus we don't introduce a let construction (we could add this).

We get

\[ q \cdot (\forall x) \leftarrow (\forall x :: a)y \leftarrow (\forall x :: a)y = \right. \]
\[ B \leftarrow A \leftarrow (\forall x :: a)y :: f \]

In order to get close to the derivations, we omit in the Agda derivation the let expression, and replace it by its definition (\( x :: a \)).
Using rules we start with our goal

\[ B \leftarrow \forall \leftarrow ((\forall C \times B) \leftarrow \forall) :^0 \rho \]

Example 4, Using Rules
Example 4, Using Rules (Cont.)

The intro step amounts to replacing $d_0$ by $\vdash p \cdot (\forall : v) \chi \cdot ((\forall \times B) \leftarrow \forall : x) \chi$

Introduces by two applications of an introduction rule:

$\vdash p \cdot (\forall : v) \chi \cdot ((\forall \times B) \leftarrow \forall : x) \chi$

• The intro step amounts to replacing $d_0$ by $\vdash p \cdot (\forall : v) \chi \cdot ((\forall \times B) \leftarrow \forall : x) \chi$
In Agda, we then replace the goal corresponding to $p$ by $(\forall x)q$.

Example 4, Using Rules (Cont.)

The two initial judgements can be introduced by assumption rules.

\[
\begin{align*}
B & \leftarrow V \leftarrow \left((C \times B) \leftarrow V \right) : (\forall x)^0 \forall \cdot (V : x) \forall \cdot ((C \times B) \leftarrow V : x) \\
B & \leftarrow V : (\forall x)^0 \forall \cdot (V : x) \forall \cdot ((C \times B) \leftarrow V : x) \\
C \times B & \leftarrow V : (\forall x) (C \times B) \leftarrow V : x
\end{align*}
\]

This can be introduced by two applications of elimination rules.

In our rule calculus, this reads $\forall x^0 (\forall x).$.

In Agda, we then replace the goal corresponding to $p$ by $(\forall x)q$.

(c) Anton Setzer 2003 (except for pictures)
Notations for Contexts

(d) Structural Rules

So $\land$, $\land$, denote contexts.
Similarly, for \( x = \lambda y : B \), the expression \( \frac{\forall y : B, \ x = \lambda y : B \cdot x}{\theta} \) stands for the context \( \theta \in \emptyset \).

\[ \theta \in \emptyset \]

\[ \theta \subseteq \emptyset \]

Notations for Contexts (cont.)
Sometimes we need assumptions or axioms to form a new judgement.

We can express this as a valid context:

\[ I \overset{\text{Context}}{\mapsto} I \]

This judgement does not occur explicitly in Agda.

- If \( I \) is a valid context

\[ \text{This would mean \( I \) is a valid context.} \]

Additional Judgement \( I \overset{\text{Context}}{\mapsto} I \)
Context Rules

Extending a context

The empty context

(Where in the last rule \( x \) must not occur in \( \Gamma \)).

\[
\frac{}{\text{Type}}
\]

\[
\Gamma \vdash A \quad \Gamma, x : A \vdash \text{Type}
\]

\[
\text{Context} \leftarrow \emptyset
\]
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 as well introduced the rules

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

\[
\text{Context} \quad \leftarrow \quad N : z , \quad N : \bar{y} , \quad N : x \\
\text{Type} \quad \leftarrow \quad N \quad \leftarrow \quad N \quad \leftarrow \quad N : \bar{y} , \quad N : x \\
\text{Context} \quad \leftarrow \quad N \quad \leftarrow \quad N \quad \leftarrow \quad N : \bar{y} , \quad N : x \\
\text{Type} \quad \leftarrow \quad N \\
\text{Type} \quad \leftarrow \quad \emptyset \\
\text{Note that } N : \text{Type} \text{ is the same as } N : \text{Type} \\
\text{The following derives } x : N , \quad \bar{y} , \quad N : z , \quad N : \bar{y} , \quad N : x \
\]

(C) Anton Setzer 2003 (except for pictures)
\[ \frac{\Gamma, x : A \vdash A'}{\Gamma, x : A, I' \vdash \text{context}} \]
We extend the derivation of B2-124 to a derivation of

\[ \text{Context} \Rightarrow N : x \]

Similarly we can derive \( x \) •

\[ N : \mathcal{H} \Rightarrow N : z', N : \mathcal{H}, N : x \]

\[ \text{Context} \Rightarrow N : z', N : \mathcal{H}, N : x \]

\[ N : \mathcal{H} \Rightarrow N : z', N : \mathcal{H}, N : x \]

\[ \text{Context} \Rightarrow N : z', N : \mathcal{H}, N : x \]

Example Derivation (Assumption Rule) •

(C) Anton Setzer 2003 (except for pictures)
The full derivation of first judgment on the previous slide is as follows:

Example Derivation (Assumption Rule: Cont)
When we define a function:

\[ f(a::A) :: B = f \,!!\, g \]

- This is an application of the assumption rule:

\[ \{i \mid i\} \]

we can make use of \( a :: A \) when solving the goal

\[ \{i \mid i\} = \begin{array}{l}
B :: \\
\exists (a :: A)
\end{array} \]

- When solving goal

\[ \text{Assumption Rule in Agda} \]
The above corresponds to a derivation in Agda (Cont.).
More generally we might in the derivation of $\forall : a$ make

$B : \exists (\forall : a) \forall$

$\vdash B : s \iff \forall : a$

$\vdash B : s \iff \forall : a$

$\vdots$

$\vdash \forall : a \iff \forall : a$

$\vdots$

Assumption Rule in Agda (cont.)
Similarly, when solving the goal

\[
B \leftarrow \forall : \{ i \; i \} \cdot (\forall : a) \forall
\]

\[
B : \{ i \; i \} \leftarrow \forall : a
\]

In order to derive

\[
B : \{ i \; i \} \leftarrow \forall : a
\]

So we have to solve

\[
B \leftarrow \forall : \{ i \; i \} \cdot (\forall : a) \forall
\]

\[
B : \{ i \; i \} \leftarrow \forall : a
\]

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

\[
\{ i \; i \} \leftarrow (\forall \; a) \forall
\]

\[
\forall \; a \; B \leftarrow \forall : a
\]

In case we can make use of \( a :: a \forall \).

Similarly, when solving the goal

**Assumption Rule in Agda (cont.)**
The judgement $I', I'$ is weakened by $\land$.

- The judgement $I', I'$ is weakened by $\land$.

This rule allows to add an additional context piece ($\land$) to the context of a judgement.

This rule allows to add an additional context piece ($\land$) to the context of a judgement.

$\theta \iff I', I' \land \theta \iff I', I' \land \theta \iff I', I'$

Weakening Rule
Weakening Rule (Cont.)

Remark: One can in fact show that the thinning rule can be weakly derived.

- Then \( \Gamma \vdash A : \text{Type} \) doesn't follow without the weakening rule.

\[
\text{Weakly derived means: whenever the assumptions of the rule can be derived in the complete set of rules we provide, then as well the conclusion.}
\]

\[
\text{An exception is when we additionally assume some judgments for instance.}
\]

\[
\text{An exception is when we additionally assume some judgments for instance.}
\]

\[
\text{A \( \vdash \text{Type} \) (corresponding to "postulate" in Agda).}
\]

\[
\text{doesn't follow without the weakening rule.}
\]
The following derives the first premise in Example 3 (slide B2-105) from assumptions Type A, Type B, Type A, Type A, Type A:

\[
\frac{\forall \alpha : \text{Type} \quad \alpha : \text{Type} \\ \exists \beta : \text{Type} \quad \beta : \text{Type} \quad \alpha : \text{Type}}{\forall \beta : \text{Type} \quad \beta : \text{Type}}
\]

Example Derivation (Weakening Rule)
Example Derivation 2 (Weakening Rule)

The following derives the first premise in Example 4 (slide B2-118) from assumptions:

\[
\begin{align*}
\forall x : \text{Type} & \quad \forall x : (\text{Type} \times \text{Type}) & \quad \forall x : \text{Type} \\
\text{Context} & \quad \left(\forall x : (\text{Type} \times \text{Type})\right) & \quad \forall x : \text{Type} \\
\text{Type} & \quad : (\text{Type} \times \text{Type}) & \quad : \text{Type} \\
\text{Type} & \quad : \text{Type} \\
\text{Type} & \quad : \text{Type} \\
\text{Type} & \quad : \text{Type}
\end{align*}
\]
General Equality Rules

Relexivity

\[ V : a = q \]
\[ \forall : q = a \]

Symmetry

\[ V : A = B \]
\[ \forall : B = A \]

(Relexivity can be weakly derived, except for additional assumptions.)

\[ V : a = a \]
\[ \forall : a \]

\[ \forall : A = A \]

Reflexivity

(C) Anton Setzer 2003 (except for pictures)
Transitivity

\[
\frac{B : q = a}{\text{Type}} \quad \frac{\text{Type} : B = \mathcal{V}}{\mathcal{V} : q = a}
\]

\[
\frac{B : a}{\text{Type}} \quad \frac{\text{Type} : B = \mathcal{V}}{\mathcal{V} : a}
\]

Transitivität

\[
\frac{\mathcal{V} : c = a}{\text{Type}} \quad \frac{\text{Type} : \mathcal{V} = \mathcal{V}}{\mathcal{V} : q = a}
\]

\[
\frac{\text{Type} : \mathcal{V} = \mathcal{V}}{\text{Type}} \quad \frac{\mathcal{V} : c = B}{B : B = \mathcal{V}} \quad \frac{\mathcal{V} : q = a}{\text{Type}}
\]

General Equality Rules (Cont.)
Example Derivation (General Equality Rules)

\[
\begin{align*}
N & \leftarrow N : \textcolor{red}{\delta (x \cdot (N : x) \cdot (x \cdot (N : x)) = 0 + \delta \cdot (N : \delta))} \\
N : \delta & \leftarrow \textcolor{red}{N : \delta} \\
\text{Context} & \leftarrow \textcolor{red}{N : \delta} \\
\text{Type} & \leftarrow \textcolor{red}{N : \delta}
\end{align*}
\]

\[
\begin{align*}
\text{Context} & \leftarrow \textcolor{red}{N : \delta} \\
\text{Type} & \leftarrow \textcolor{red}{N : \delta}
\end{align*}
\]
In the previous derivation, the most complicated step was:

\[ y : N \quad ; \quad x : N \quad \]

\[ y : N \]

This is an example of the equality rule for the non-dependent function type

\[ \text{(B2-34)} \]

\[ \exists \]
Example Derivation (General Equality Rules; Cont.)

\[
\frac{\text{N} : f_i (x \cdot (\text{N} : x) \gamma) \Leftarrow \text{N} : f_i}{\text{N} : f_i \Leftarrow \text{N} : f_i}
\]

\[
\frac{\text{N} \Leftarrow \text{N} : x \cdot (\text{N} : x) \gamma \Leftarrow \text{N} : f_i}{\text{N} : x \Leftarrow \text{N} : x \cdot \text{N} : f_i}
\]

Note that from the premises of that rule as follows:

\[
\frac{\text{N} : f_i (x \cdot (\text{N} : x) \gamma) \Leftarrow \text{N} : f_i}{\text{N} : f_i \Leftarrow \text{N} : f_i}
\]

\[
\frac{\text{N} : f_i = f_i (x \cdot (\text{N} : x) \gamma) \Leftarrow \text{N} : f_i}{\text{N} : f_i \Leftarrow \text{N} : f_i}
\]

\[
\frac{\text{N} : f_i \Leftarrow \text{N} : f_i}{\text{N} : x \Leftarrow \text{N} : x \cdot \text{N} : f_i}
\]
The equality rule expresses how the function $x : \mathbb{N}$ is evaluated as follows:

1. We evaluate the body of the function $f$ by setting the argument of the function, i.e., $f(x)$.
2. This is the same as substituting in the body for $x$ the argument of the function, i.e., $f(x)$.
3. This explains how the detour above of first introducing and then eliminating an expression can be reduced (namely to or in general to $f(y)$).

This equality rule expresses how the function $\forall x : \mathbb{N} \cdot x$ is applied to $y$. 

Example Derivation (General Equality Rules: Cont.)
The following rules can be weakly derived:

**Substitution Rules**

**Substitution 1**

\[ [p =: x] q = [p =: x] q \leftarrow [p =: x], \mathcal{I}, \mathcal{I} \]

\[ V : p = q \leftarrow \mathcal{I} \]

\[ B : q \leftarrow \mathcal{I}, V : x, \mathcal{I} \]

**Substitution 2**

\[ [p =: x] B = [p =: x] B \leftarrow [p =: x], \mathcal{I}, \mathcal{I} \]

\[ V : p = q \leftarrow \mathcal{I} \]

\[ \exists \mathcal{I} : B \leftarrow \mathcal{I}, V : x, \mathcal{I} \]

**Substitution 3**

\[ [p =: x] \theta \leftarrow [p =: x], \mathcal{I}, \mathcal{I} \]

\[ V : \theta \leftarrow \mathcal{I} \]

\[ \exists \mathcal{I} : V : x, \mathcal{I} \]
Example Derivation (Substitution)

\[
\begin{align*}
N & \leftarrow N : \tilde{f} + 0 (N : \tilde{f}) \\
N : \tilde{f} + 0 & \leftarrow N : \tilde{f} \\
N : 0 & \leftarrow \quad N : \tilde{f} + x \leftarrow N : \tilde{f} , N : x \\
N : \tilde{f} & \leftarrow N : \tilde{f} , N : x \\
N : x & \leftarrow N : \tilde{f} , N : x \\
\ldots & \\
\end{align*}
\]
Example Derivation (Substitution)
In order to derive $x : A', y : B : \text{Type}$ we need to show:

So the judgement implicitly contains the judgements

\[
\forall : B \iff \forall : x
\]

•
The next slide shows the presuppositions of judgments.

\[
\text{Presuppositions of the judgment: } A : \text{Type}, \forall x, y : B \iff \forall A : \text{Type} \text{ and } x : A \implies B.
\]
<table>
<thead>
<tr>
<th>Presuppositions</th>
<th>Judgment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I : A \leftarrow I$</td>
<td>$A : a \leftarrow I$</td>
</tr>
<tr>
<td>$I : B \leftarrow I$</td>
<td>$A : B = V \leftarrow I$</td>
</tr>
<tr>
<td>$I : A : Type$</td>
<td>$I : A : Type$</td>
</tr>
<tr>
<td>$x : A : Type$</td>
<td>$x : A : Type$</td>
</tr>
</tbody>
</table>

Furthermore, presuppositions of presuppositions of

\[ \theta \iff I \]

are as well presuppositions of

\[ \theta \iff I \]

Furthermore, presuppositions of presuppositions of
Example of Presuppositions:

\[ D \times (\mathcal{C} : z) : q \Leftarrow B : \forall x : A \]

\[ D \times (\mathcal{C} : z) : q \Leftarrow B : \forall x : A \]

\[ D \times (\mathcal{C} : z) : q \Leftarrow B : \forall x : A \]

\[ D \times (\mathcal{C} : z) : q \Leftarrow B : \forall x : A \]

\[ D \times (\mathcal{C} : z) : q = q \Leftarrow B : \forall x : A \]

\[ D \times (\mathcal{C} : z) : q = q \Leftarrow B : \forall x : A \]
Remark on \( A \rightarrow B \times A \rightarrow B \)  

<table>
<thead>
<tr>
<th>( x )</th>
<th>( A \rightarrow B )</th>
<th>( A \rightarrow B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \rightarrow A \rightarrow B )</td>
<td>Context,( \rightarrow A \rightarrow B )</td>
<td>Type,( \rightarrow A \rightarrow B )</td>
</tr>
</tbody>
</table>

\( A \rightarrow B \) is an abbreviation for \( (x : A) \rightarrow (x : A) \rightarrow B \) for some fresh \( x \).

Therefore the presupposition of \( A \rightarrow B : Type \) are:

- \( A \rightarrow B : Type \) (which abbreviates ;)
- \( A \rightarrow B : Type \)

Similarly \( A \rightarrow B \times A \rightarrow B \) is an abbreviation for some fresh \( x \).

Note that \( A \rightarrow B \rightarrow (A : A) \rightarrow B \) for some fresh \( x \).
We would like to add operations on types, such as

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

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

The problem is that for this we need

\[ \text{Type} \rightarrow \text{Type} \rightarrow \text{Type} \]

and our rules allow this only if we had

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

Critical Systems, CS
411, Lent term 2003, Sec. B2
The corresponding paradox is called Girard’s paradox. Using this rule we can prove everything, especially false formulas. As a rule results however in an inconsistent theory.

$$
\text{Type : Type Set}
$$
The Founder of Martin-Löf Type Theory.

Per Martin-Löf
The main theoretician behind Agda (which was implemented by his wife, or whom I have no picture).

Thierry Coquand
Instead we introduce a new type.

Set : \text{Type}

Set is the type of sets.

- A set is a small type.
We add rules asserting that if A : Set then A : Type.

(C) Anton Setzer 2003 (except for pictures)
However, we cannot use prod in order to form the product of two sets, i.e., we cannot introduce sets.

That would result in the same inconsistency as Type : Type.

Since Set : Set does not hold,

prod Set Set : Set

*
Every set is a type

Set : Type

Closure of set under the dependent function type

\[
\forall : A \to (A : B) \\
\forall : A \to (\forall : x) \\
\forall : B \to (\forall : x) \\
\forall : A \to (\forall : x) \\
\forall : B \to (\forall : x)
\]

Formation rule for set

Rules for set


B2-152
Equality Versions of the Above Rules

Formation Rule for Set

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

Every Set is a Type

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

Closure of Set under the dependent product

\[ \text{Set} : \text{B} \times \text{(\forall x)} = \text{B} \times \text{(\forall x)} \]

Closure of Set under the dependent function type

\[ \text{Set} : \text{B} \rightarrow \text{(\forall x)} = \text{B} \rightarrow \text{(\forall x)} \]
First we derive $X : \text{Set}$, $\forall X : \text{Set} : X$:

We can now introduce $\text{prod} : \text{Set} : \text{Set}$.
Using this we can derive

Example: prod (cont.)
Example: prod (Cont.)

Now we can derive our desired judgement:

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

\[ \lambda(X, Y : \text{Set}, x : X) \times Y \Rightarrow (x : X) \times Y : \text{Set} \Rightarrow \text{Set} \]

So define

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

(C) Anton Setzer 2003 (except for pictures)
Hierarchies of Types

If one wants to form prod : Type → Type → Type : Kind, one needs to have a further level Kind, s.t. Type : Kind.

- Then Type ← Type ← Type : Kind.
Rules for Type as a Kind

Every Type is a Kind

Type: Kind

Closure of Kind under the dependent function type

\[ \text{Kind : } A \rightarrow B \rightarrow C \Rightarrow A \rightarrow (B \rightarrow C) \]

Closure of Kind under the dependent product

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

Closure of Kind under the dependent product

\[ \text{Kind : } A \times (B \times C) \Rightarrow (A \times B) \times C \]

Plus equality versions of the above rules.

\[ \text{Kind : } A \leftarrow (A \times B) \Rightarrow A \]

\[ \text{Kind : } A \leftarrow (B \times C) \Rightarrow B \]

\[ \text{Kind : } A \leftarrow (A \rightarrow B) \Rightarrow A \rightarrow B \]
This can be iterated further, forming hierarchies of types.

\[ \text{Hierarchies of Types (Cont.)} \]