3. The Logical Framework

(a) Judgements.
(b) Basic form of rules.
(c) The non-dependent function type and product.
(d) Structural rules. (Omitted 2005).
(e) The dependent function type and product.
(f) Derivations vs. Agda code. (Omitted 2005).
(g) Presuppositions (Omitted 2005).
(h) The full logical framework

This plan might be changed later when the rest of the lecture is developed.
(a) Judgements

- In the $\lambda$-calculus, it is easy to determine the correctly formed types. In dependent type theory the type structure is richer and more complicated.
- Proof steps are required to conclude that something is a type.
Judgements

Therefore we have not only the judgement as in the \( \lambda \)-calculus

\[ a : A \]

but as well a typing judgement \( A \) is a type, written (as we have already seen)

\[ A : \text{Set} \]

Before deriving \( a : A \), where \( A \) is a constant, we first have to show \( A : \text{Set} \).
Equality Judgements

Agda will identify terms which have the same normal form.
E.g. $s := (\lambda x^A.x) \ r$ and $r$ will be identified.

If one needs at some place $r$, one can insert $s$ instead of $r$ and vice versa.

In Agda this is done automatically, the user doesn’t see such equalities.

There is not even a direct command available in Agda, which allows to check whether two terms are equal (this could probably be added easily).
Example

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

$g \ (a :: A)$
  :: $A$
  = $a$

$a' :: A$
  = $g \ a$

$p \ (x :: f \ a)$
  :: $f \ a'$
  = $\{! \ !\}$

```
exampleSimpleEquality2.agda
```

Since $a' = g \ a = a$, we can solve the goal by using $x$. 
Equality Judgements

- When using the simply typed $\lambda$-calculus, we could separate the derivation of $\lambda$-terms, from reductions.
- When using dependent type theory as in Agda, reductions and derivations have to be integrated.
- Traditionally, instead of introducing reductions, one introduces in dependent type theory equalities between terms.
- Written as

$$r = s : A$$

for $r$ and $s$ are equal elements of set $A$. 
Example

The rule expressing that $\pi_0(\langle a, b \rangle) \rightarrow a$ reads in this style as follows:

$$\frac{a : A \quad b : B}{\pi_0(\langle a, b \rangle) = a : A} \quad (\times\text{-Eq}_0)$$

is not directed, so we have as well the rule

$$\frac{a = b : A}{b = a : A} \quad (\text{Sym}_{\text{Elem}})$$

We can therefore derive:

$$\frac{a : A \quad b : B}{\pi_0(\langle a, b \rangle) = a : A} \quad (\times\text{-Eq}_0)$$

$$\frac{a = \pi_0(\langle a, b \rangle) : A}{a = \pi_0(\langle a, b \rangle) : A} \quad (\text{Sym}_{\text{Elem}})$$
Equality of Types

We will have as well equality between types, written as

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

This is something novel in dependent type theory.

In simple type theory, there is only one way of writing a type.
Examples (Equality of Types)

Assume \( f : A \to \text{Set} \).
If \( a = a' : A \), then

\[
f a = f a' : \text{Set}.
\]

We used this in the example above:

There we had

\[
x : f a
\]

and could by \( f a = f a' \) conclude

\[
x : f a'
\]

Above we have defined \( o2 = o \to o \).
As a judgement this reads:

\[
o2 = o \to o : \text{Set}.
\]
Four Judgements

So we have the following 4 types of judgements:

\begin{align*}
A : \text{Set} & \quad \text{“} A \text{ is a type”}. \\
 a : A & \quad \text{“} a \text{ is of type } A \text{”}. \\
A = B : \text{Set} & \quad \text{“} A \text{ and } B \text{ are equal types”}. \\
 a = b : A & \quad \text{“} a \text{ and } b \text{ are equal elements of type } A \text{”}. \\
\end{align*}

In Agda, only $A : \text{Set}$ and $a : A$ are explicit.
Dependent Judgements

- As for the simply typed λ-calculus, in dependent type theory, judgements might depend on a context.

- So we obtain judgements of the form

\[
x_1 : A_1, \ldots, x_n : A_n \implies A : \text{Set}
\]

\[
x_1 : A_1, \ldots, x_n : A_n \implies A = B : \text{Set}
\]

\[
x_1 : A_1, \ldots, x_n : A_n \implies s : A
\]

\[
x_1 : A_1, \ldots, x_n : A_n \implies s = t : A
\]
Need for Context Rules

\[ x_1 : A_1, \ldots, x_n : A_n \Rightarrow A : \text{Set} \]

\[ \ldots \]

\[ \text{To derive such judgements requires that we know} \]

\[ A_1 : \text{Set} \]

\[ x_1 : A_1 \Rightarrow A_2 : \text{Set} \]

\[ x_1 : A_1, x_2 : A_2 \Rightarrow A_3 : \text{Set} \]

\[ \ldots \]

\[ x_1 : A_1, x_2 : A_2, \ldots, x_{n-1} : A_{n-1} \Rightarrow A_n : \text{Set} \]
Context Rule

- Note that we didn’t require derivations as above in the simply typed $\lambda$-calculus, since it was easy to verify whether something is a valid type.

- In case of dependent types $A : \text{Set}$ requires a derivation.

- It can be as complicated to derive $A : \text{Set}$ as it is to derive a judgement $b : B$:

  One can compute from an unchecked judgement $a : A$ an unchecked type expression $B$ s.t. $a : A$ holds iff $B : \text{Set}$ holds.
Context Rule

In order to organise this in a better way we introduce an additional judgement $\Gamma \Rightarrow \text{Context}$ for “$\Gamma$ is a valid context”.

That $x_1 : A_1, \ldots, x_n : A_n \Rightarrow \text{Context}$ holds means exactly what we had above, i.e.:

\[
\begin{align*}
A_1 & : \text{Set} \\
x_1 : A_1 & \Rightarrow A_2 : \text{Set} \\
x_1 : A_1, x_2 : A_2 & \Rightarrow A_3 : \text{Set} \\
\ldots & \\
x_1 : A_1, x_2 : A_2, \ldots, x_{n-1} : A_{n-1} & \Rightarrow A_n : \text{Set}
\end{align*}
\]
Five Dependent Judgements

We have therefore 5 dependent judgements:

\[ x_1 : A_1, \ldots, x_n : A_n \Rightarrow A : \text{Set} \]
\[ x_1 : A_1, \ldots, x_n : A_n \Rightarrow A = B : \text{Set} \]
\[ x_1 : A_1, \ldots, x_n : A_n \Rightarrow s : A \]
\[ x_1 : A_1, \ldots, x_n : A_n \Rightarrow s = t : A \]
\[ x_1 : A_1, \ldots, x_n : A_n \Rightarrow \text{Context} \]
Example

The assumption rule, which in case of the simply typed \( \lambda \)-calculus read:

\[
\Gamma \Rightarrow x : \sigma \quad \text{(if } x : \sigma \text{ occurs in } \Gamma)\]

reads in dependent type theory as follows (assuming that \( x : A \) occurs in \( \Gamma \)):

\[
\frac{\Gamma \Rightarrow \text{Context} \quad \text{(Ass)}}{\Gamma \Rightarrow x : A}
\]

Similarly we have to deal with the rule introducing constants.
Notations for Judgements, Contexts

\( \theta \) will in the following denote an arbitrary non dependent judgement, i.e. one of the following:

- \( A : \text{Set} \),
- \( A = B : \text{Set} \),
- \( a : A \),
- \( a = b : A \).

\( \Gamma, \Delta \) will usually denote contexts.

We have the same notations as before, i.e.

- \( \Gamma, \Delta \) is the result of concatenating contexts \( \Gamma, \Delta \),
- \( \Gamma, x : A \) is the result of extending the context \( \Gamma \) by \( x : A \),
- \( \emptyset \) is the empty context.
- We write for \( \emptyset \Rightarrow \theta \) usually simply \( \theta \).
In Agda, we have no explicit judgements depending on contexts.

Not needed, since we don’t derive judgements using rules directly.

However, if we have the open judgement

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

Then we can make use of \( x :: B \) for refining the goal.

So we have to solve the goal in context \( x :: B \).

This context can be shown using goal menu Context.

See `exampleShowContext.agda`. 
Jump over the next example.
Example: Derivation of double

(See exampleDoubleString2.agda.)

- We derive
  \[ \text{double} := \lambda x : \text{String}. \text{concat} \ x \ x : ((x : \text{String}) \to \text{String}) \] in Agda, assuming definitions of \text{String} and \text{concat}.

- We start with
  \[
  \text{double} \ (x :: \text{String}) :: \text{String} \\
  = \{! \ !\}
  \]

- We can insert into the goal \text{concat}:
  \[
  \text{double} \ (x :: \text{String}) :: \text{String} \\
  = \{! \ \text{concat} \ !\}
Example: Derivation of double

When using goal-menu **refine**, we obtain:

\[
\text{double } (x :: \text{String})
\]
\[
:: \text{String}
\]
\[
= \text{concat} \{! \} \{! \}\]

We can check now using goal-menu **Type of Goal** (or **Type of Goal (unfolded)**) that the two new goals require both type **String**.

We can check using goal-menu **Context** that the context of both goals contain \(x :: \text{String}\).
Example: Derivation of double

We insert $x$ into the first goal and refine:

$$\text{double } (x :: \text{String})$$

$$:: \text{String}$$

$$= \text{concat } x \{! !\}$$

Doing the same with the second goal gives:

$$\text{double } (x :: \text{String})$$

$$:: \text{String}$$

$$= \text{concat } x \ x$$

We are done.
double in Type Theory

A derivation of

\[
\text{double} := \lambda x : \text{String}. \text{double} \ x \ x
\]

in Type Theory, assuming global constants

\[
\begin{align*}
\text{String} & : \ 	ext{Set} , \\
\text{concat} & : \ 	ext{String} \to \text{String} \to \text{String} ,
\end{align*}
\]

is as follows:
We first derive \( x : \text{String} \Rightarrow \text{Context} : \)

\[
\frac{
\emptyset : \text{Context} \quad \text{String} : \text{Set}}{x : \text{String} \Rightarrow \text{Context} \quad (\text{Context}_1)}
\]
We derive \( x : \text{String} \Rightarrow x : \text{String} \) using the previous derivation:

\[
\frac{x : \text{String} \Rightarrow \text{Context}}{x : \text{String} \Rightarrow x : \text{String} \quad \text{Ass}}
\]

We derive

\[
x : \text{String} \Rightarrow \text{concat} : \text{String} \rightarrow \text{String} \rightarrow \text{String}
\]

using \( x : \text{String} \Rightarrow \text{Context} \) as follows:

\[
\frac{\text{concat} : \text{String} \rightarrow \text{String} \rightarrow \text{String} \quad x : \text{String} \Rightarrow \text{Context}}{x : \text{String} \Rightarrow \text{concat} : \text{String} \rightarrow \text{String} \rightarrow \text{String} \quad \text{(Weak)}}
\]
double in Type Theory

We derive $x : \text{String} \Rightarrow \text{concat} \ x : \text{String} \rightarrow \text{String}$ using the previous derivations:

\[
\begin{align*}
x : \text{String} \Rightarrow \text{concat} : \text{String} \rightarrow \text{String} \rightarrow \text{String} & \quad x : \text{String} \Rightarrow x : \text{String} \\
\hline
x : \text{String} \Rightarrow \text{concat} \ x : \text{String} \rightarrow \text{String} & \quad (\rightarrow\text{-El})
\end{align*}
\]

The remaining derivation using the above derivations is as follows:

\[
\begin{align*}
x : \text{String} \Rightarrow \text{concat} \ x : \text{String} \rightarrow \text{String} & \quad x : \text{String} \Rightarrow x : \text{String} \\
\hline
x : \text{String} \Rightarrow \text{concat} \ x \ x : \text{String} & \quad (\rightarrow\text{-El}) \\
\hline
\text{double} := \lambda x^{\text{String}} . \text{concat} \ x \ x : \text{String} \rightarrow \text{String} & \quad (\rightarrow\text{-I})
\end{align*}
\]
(b) 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 new types.
- Each type construction has one such rule.
- The conclusion of such a rule will have the form:
  \[ C \ a_1 \ \cdots \ a_n : \text{Set} \]

  where \( C \) is a type-constructor,
  \( a_1, \ldots, a_n \) are its arguments.
  \( n = 0 \) is possible.

- Later, we will introduce higher levels Type, Kind, \ldots.

  Then we have formation rules with conclusion
  \[ C \ a_1 \ \cdots \ a_n : \text{Type} \ (\text{or} : \text{Kind}, \text{etc.}) \]
Preliminarily, we will be using type theory without the full logical framework.

For instance, below we will introduce

\[ \text{List } A : \text{Set} \]

for any \( A : \text{Set} \), the set of lists of elements of \( A \).
Logical Framework

Until we have introduced the full logical framework, it doesn’t make sense to talk about \texttt{List} itself, which would have type

$$\texttt{List} : \texttt{Set} \rightarrow \texttt{Set}.$$  

The problem is that \texttt{Set} \rightarrow \texttt{Set} doesn’t make sense without the logical framework.

The full logical framework is conceptually more difficult, that’s why we delay its introduction.

When it is introduced, we can introduce

$$\texttt{List} : \texttt{Set} \rightarrow \texttt{Set}$$

similarly for all other set formation constructors.
Agda has the logical framework built in, so in Agda $\text{List}$ will be a function $\text{Set} \rightarrow \text{Set}$, in Agda notation:

$$
\text{List } (A :: \text{Set})
\quad :: \text{Set}
\quad = \ldots
$$
Example 1: The Set of Lists

\[
\begin{array}{c}
A : \text{Set} \\
\hline
\text{List} \ A : \text{Set} \\
(\text{List-F})
\end{array}
\]

- The **type-constructor** is **List**.
- List \( A \) is the type of lists of type \( A \).
Ex. 2: The Set of Natural Numbers

Formation rule for the type of natural numbers:

\[ \text{N : Set} \quad (\text{N-F}) \]

The **type-constructor** is \textbf{N}.

Note that the formation rule for \textit{N} has 0 premises (therefore the fraction bar is omitted).
Ex. 3: The Non-Dependent Product

Formation rule for the non-dependent product:

\[
\begin{array}{c}
A : \text{Set} \quad B : \text{Set} \\
\hline
A \times B : \text{Set}
\end{array}
\quad (\times\text{-}F)
\]

- \( A \times B \) stands for \((\times)\ A\ B\).
- The type-constructor is \((\times)\).
 Formation Rules in Agda

The formation of a type is usually done by introducing a constant of a certain type.

Example 1:

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

:: Set

= \ldots
Example 2: \((\times)\)

- Agda syntax for introducing the non-dependent product:

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

\(\ldots\) is an Agda definition of this type (more about this later).

\(\times\) is ASCII symbol 215 (not the letter x).

Remember that there is as well a predefined versions of the product in Agda, based on sig.
(2) Introduction Rules

The **introduction rule** introduces elements of a type.

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

\[ C \ a_1 \ \cdots \ a_n : A \]

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 (can be elements of other sets, or sets or types themselves).
The set \texttt{NatList} of lists of natural numbers with formation rule

\[
\text{NatList} : \text{Set} \quad (\text{NatList-F})
\]

has two introduction rules:

\[
\text{nil} : \text{NatList} \quad (\text{N-I}_{\text{nil}})
\]

\[
\frac{n : \text{N} \quad l : \text{NatList}}{\text{cons } n \; l : \text{NatList}} \quad (\text{NatList-I}_{\text{cons}})
\]
We generalise the previous example to lists of arbitrary type.

Lists of type $A$ have two introduction rules:

\[
\frac{A : \text{Set}}{\text{nil} \ A : \text{List} \ A} \quad \text{(List-\text{I}_{\text{nil}})}
\]

\[
\frac{A : \text{Set} \quad a : A \quad l : \text{List} \ A}{\text{cons} \ A \ a \ l : \text{List} \ A} \quad \text{(List-\text{I}_{\text{cons}})}
\]

Note that we need the premise $A : \text{Set}$ in order to guarantee that we can form the set $\text{List} \ A$. 
Conflicting Constructors

- We shouldn’t use the same constructors for different sets. So if we want to use both NatList and List A, we have to choose a notation like natnil instead of nil : NatList, similarly for cons.

- We will usually ignore this distinction, if it doesn’t cause confusion.
Example 2: Natural Numbers.

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

- $0$,
- $S$ where $S \ n$ stands for $n + 1$.

Using these two operations we can form $0$, $S \ 0 = 1$, $S \ 1 = 2$, $\ldots$ and therefore all natural numbers.

So the **constructors** of $\mathbb{N}$ are $0$ and $S$.

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

\[
\begin{align*}
0 : \mathbb{N} & \quad \text{(N-I}_0) \\
\frac{n : \mathbb{N}}{S \ n : \mathbb{N}} & \quad \text{(N-I}_S)
\end{align*}
\]
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, cons \((1 + 1)\) (concat (cons 0 nil) nil) in case of \( \text{NatList} \).
Non-Canonical Elements

Terms can usually be reduced further

Example:

\[ 2 + 3 = 2 + S\ 2 \rightarrow S\ (2 + 2) \, . \]

The underlying reduction system is essentially a term rewriting system combined with the \( \lambda \)-calculus.

Therefore we can apply reductions to subterms.

A term is a non-canonical element of a type, if it reduces to a canonical element of that type.

Each element of a type (depending on the empty context) in dependent type theory will either be a canonical or a non-canonical element of that type.

Consequence of the normalisation theorem.
Non-Canonical Elements

- E.g. $2 + 3$ is a non-canonical element of $\mathbb{N}$, since $S(2 + 2)$ is a canonical element of $\mathbb{N}$.

- However, we have

$$x : \mathbb{N} \Rightarrow x : \mathbb{N}$$

and $x$ doesn’t reduce to a canonical element of $\mathbb{N}$.

- However, if we substitute for $x$ any closed element of $\mathbb{N}$, we get a canonical or non-canonical element of $\mathbb{N}$. 
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 as in dependent type theory with the logical framework, i.e. we have

\[
\begin{align*}
nil@(\text{List } N) & :: \text{ List } N \\
\text{cons}@\!(\text{List } N) & :: (n :: N) \\
& \rightarrow (l :: \text{List } N) \\
& \rightarrow \text{List } N
\end{align*}
\]
Constructors in Agda

Since notations like \( \text{nil@}(\text{List } N) \) are usually too 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 \( \text{nil, cons for List } N \), and not for the general case \( \text{List } A \) for any type \( A \). (That would require an extra argument \( A : \text{Set} \).)
(3) Elimination Rules

- **Elimination rules** allow to take an element of a type and **compute from it an element of another type**.

- Example 1: The introduction rule for the non-dependent product is

  \[
  \frac{a : A \quad b : B}{\langle a, b \rangle : A \times B} \quad (\times \text{-I})
  \]

  The elimination rules are the first and second projections:

  \[
  \frac{c : A \times B}{\pi_0(c) : A} \quad (\times \text{-El}_0) \quad \frac{c : A \times B}{\pi_1(c) : B} \quad (\times \text{-El}_1)
  \]

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

\[
\frac{n : \mathbb{N} \quad m : \mathbb{N}}{n + m : \mathbb{N}} \quad \text{(N-El$_+$)}
\]

Equality rules will express

- $n + 0 = n$.
- $n + \mathsf{S} \; m = \mathsf{S} \; (n + m)$.

The equality rules show that $n$ is only a parameter, we are eliminating the second argument $m$.

Proceeding like this would require one elimination rule for each function from $\mathbb{N}$ we want to define.

Instead we will later introduce one generic elimination rule, which will allow to introduce all functions we expect to be definable, including all primitive-recursive ones.
Elimination in Agda

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

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

\[
(+) \quad (n, m :: \mathbb{N}) \\
:: \mathbb{N} \\
= \text{case } m \text{ of} \\
\quad (Z) \to n \\
\quad (S \; m') \to S \; (n + m')
\]
(4) Equality Rules

Equality rules will express what happens when we first introduce an element and then eliminate it.

For instance if we first introduce \(0 : N\) and then eliminate it by using \((N-\text{El}_+)\) we obtain \(n + 0\).

Now \(n + 0\) should reduce to \(n\).

Since in dependent type theory we don’t derive reductions but equalities, which is the transitive, symmetric and reflexive closure of \(\rightarrow\), we obtain \(n + 0 = n : N\) instead.

The equality rule expresses this:

\[
\frac{n : N}{n + 0 = n : N} \quad (N-\text{Eq}_+,0)
\]
Equality Rules

Similarly, if we introduce first \( S \, m : N \) and then eliminate it using \((N\text{-El}_+)\) we obtain \( n + S \, m \) which should reduce to \( S \, (n + m) \).

The corresponding equality rule is therefore:

\[
\begin{array}{c}
\hline
n : \mathbb{N} & m : \mathbb{N} \\
\hline
n + S \, m = S \, (n + m) : \mathbb{N}
\end{array}
\]

\( (N\text{-Eq}_{+,S}) \)

A third example is if we first introduce an element \( \langle a, b \rangle : A \times B \) and then eliminate it using \((\times\text{-El}_0)\) we obtain \( \pi_0(\langle a, b \rangle) \) which reduces to \( a \).

The corresponding equality rule is therefore:

\[
\begin{array}{c}
a : A & b : B \\
\hline
\pi_0(\langle a, b \rangle) = a : A
\end{array}
\]

\( (\times\text{-Eq}_0) \)
Example (Equality Rule)

The first equality rule for $A \times B$ is as follows:

$$
\frac{a : A \quad b : B}{\pi_0(\langle a, b \rangle) = a : A} \quad (\times\text{-Eq}_0)
$$

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

$$
\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B} \quad (\times\text{-I})
\frac{\langle a, b \rangle : A \times B}{\pi_0(\langle a, b \rangle) : A} \quad (\times\text{-El}_0)
$$

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

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

$$
\frac{a : A \quad b : B}{\pi_1(\langle a, b \rangle) = b : B} \quad (\times\text{-Eq}_1)
$$
Example 2 (Equality Rule)

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

$$\frac{n : N}{n + 0 = n : N} \quad (N\text{-Eq}_+, 0)$$

$n + 0 : N$ can be derived by first introducing $0 : N$

(this is an introduction rule with no premises, i.e. an axiom)
and then by eliminating it using $+$, using the following derivation:

$$\frac{n : N}{0 : N} \quad (N\text{-El}_+)$$

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

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

\[
\begin{array}{c}
\frac{n : N \quad m : N}{n + S\ m = S\ (n + m) : N} \quad (N\text{-Eq}_+,S)
\end{array}
\]

\( n + S\ m : N \) can be derived by first introducing \( S\ m : N \) and then by eliminating it using \( + \):

\[
\begin{array}{c}
\frac{m : N}{n : N \quad S\ m : N} \quad (N\text{-I}_S)
\end{array}
\]

\[
\begin{array}{c}
\frac{n : N \quad S\ m : N}{n + S\ m : N} \quad (N\text{-El}_+)
\end{array}
\]
Equality Rules in Agda

- Equality Rules in Agda are **implicit**.
- The notation for elimination however indicates already how the reductions take place.

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

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

Jump over Reduction Strategy
Reduction Strategy

The canonical element for an element, which is the result of an elimination, can always be 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
- $\pi_0(\langle a, b \rangle) \rightarrow a$.
- $\pi_1(\langle a, b \rangle) \rightarrow b$. 
Reduction Strategy

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 which we are “eliminating”.
Example of the Reduction Strategy

Consider for instance the term \((1 + 1) + (1 + 0)\), where \(1 = S\ 0\).

It is constructed by using the elimination constant \((+\).\)

The argument we are eliminating using \((+\) is the second one \((1 + 0)\).

So we first reduce this argument to canonical form:

\[
1 + 0 \rightarrow 1
\]

and obtain

\[
(1 + 1) + (1 + 0) \rightarrow (1 + 1) + 1 \equiv (1 + 1) + S\ 0
\]
Example of the Reduction Strategy

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

- Now the argument we are eliminating in is in canonical form, and we can use the reduction rule
  \[x + S \ y \rightarrow S \ (x + y)\] in order to reduce this term:

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

- The result is in this case already in canonical form.
- If it were not, we would continue with our reduction.
- However, even if our example is in canonical form, it can be further reduced:

\[
S((1 + 1) + 0) \rightarrow S(1 + 1) \equiv S(1 + S \ 0) \rightarrow S \ (S \ 1) = 3
\]
Equality Versions of the Rules

- We have equality versions of the 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 \( \text{List} \):

  \[
  \frac{A = B : \text{Set}}{\text{List} \ A = \text{List} \ B} \quad (\text{List-F=})
  \]

- Example: Equality version of the formation rule for \( \text{N} \) (degenerated):

  \[
  N = N : \text{Set} \quad (\text{N-F=})
  \]
Equality Versions of Rules

Example: Equality version of the introduction rules for List:

\[
\begin{align*}
A &= A' : \text{Set} & (\text{List-}\_\text{I}^=) \\
\text{nil } A &= \text{nil } A' : \text{List } A
\end{align*}
\]

\[
\begin{align*}
A &= A' : \text{Set} & a &= a' : A & l &= l' : \text{List } A & (\text{List-}\_\text{I}^=) \\
\text{cons } A a l &= \text{cons } A' a' l' : \text{List } A
\end{align*}
\]

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

\[
\begin{align*}
n &= n' : \text{N} & m &= m' : \text{N} & (\text{N-}\_\text{El}^=) \\
n + m &= n' + m' : \text{N}
\end{align*}
\]
Equality Versions of Rules

- The equality versions of the rules in questions can be formed in a **straight-forward way**, once one knows the non-equality version.
- We will often not mention them.
- In **Agda** they are **implicit** (part of the reduction machinery).

Jump over Weakening Rule
The convention is that all rules can as well be weakened by a common context.

This means that when introducing a rule

\[
\frac{\Gamma_1 \Rightarrow \theta_1 \quad \ldots \quad \Gamma_n \Rightarrow \theta_n}{\Gamma \Rightarrow \theta}
\]

we implicitly introduce as well the following rules

\[
\frac{\Delta, \Gamma_1 \Rightarrow \theta_1 \quad \ldots \quad \Delta, \Gamma_n \Rightarrow \theta_n}{\Delta, \Gamma \Rightarrow \theta}
\]

This convention will not apply to the context rules \((\text{Context}_0)\) and \((\text{Context}_1)\) (see later).
Example

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

$$\frac{A : \text{Set} \quad B : \text{Set}}{A \times B : \text{Set}} (\times\text{-F})$$

can be weakened as follows:

$$\frac{\Gamma \Rightarrow A : \text{Set} \quad \Gamma \Rightarrow B : \text{Set}}{\Gamma \Rightarrow A \times B : \text{Set}} (\times\text{-F'})$$
Example (Cont.)

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

$$
\begin{align*}
  x &: A, y &: A \Rightarrow y &: A \\
  x &: A \Rightarrow \lambda y^A.y &: A \rightarrow A \\
  \lambda x^A.\lambda y^A.y &: A \rightarrow A \rightarrow A
\end{align*}
$$

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.
Weakening of Axioms

If we have an axiom

\[ \theta \]

for any judgement \( \theta \)

- e.g. \( \theta \equiv N : \text{Set} \) or \( \theta \equiv 0 : N \)

and we want to weaken it by context \( \Gamma \), we need to make sure that \( \Gamma \Rightarrow \text{Context} \) holds.

So we need in the weakened form one additional premise:

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

■ The formation rule for $N$

\[
N : \text{Set} \quad (\text{N-F})
\]

will be weakened as follows:

\[
\Gamma \Rightarrow \text{Context} \quad (\text{N-F}')
\]

\[
\Gamma \Rightarrow N : \text{Set}
\]
(c) Nondep. Funct. Type and Product

We introduce in the following non-dependent versions of the product and the function type.
The Non-Dependent Product

**Formation Rule**
\[
\frac{A : \text{Set} \quad B : \text{Set}}{A \times B : \text{Set}} \quad (\times\text{-F})
\]

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

**Elimination Rules**
\[
\frac{c : A \times B}{\pi_0(c) : A} \quad (\times\text{-El}_0) \quad \frac{c : A \times B}{\pi_1(c) : B} \quad (\times\text{-El}_1)
\]

**Equality Rules**
\[
\frac{a : A \quad b : B}{\pi_0(\langle a, b \rangle) = a : A} \quad (\times\text{-Eq}_0)
\]
\[
\frac{a : A \quad b : B}{\pi_1(\langle a, b \rangle) = b : B} \quad (\times\text{-Eq}_1)
\]
The $\eta$-Rule

The $\eta$-rule does not fit into the above schema:

$$
\begin{align*}
&\quad c : A \times B \\
\hline
&c = \langle \pi_0(c), \pi_1(c) \rangle : A \times B
\end{align*}
$$

\text{(}\times-\eta\text{)}
Equality Versions of the $\times$-Rules

Equality Version of the Formation Rule

$$
\frac{A = A' : \text{Set}}{A \times B = A' \times B' : \text{Set}} \quad (\times\text{-}F=)
$$

Equality Version of the Introduction Rule

$$
\frac{a = a' : A \quad b = b' : B}{\langle a, b \rangle = \langle a', b' \rangle : A \times B} \quad (\times\text{-}I=)
$$

Equality Versions of the Elimination Rules

$$
\frac{c = c' : A \times B}{\pi_0(c) = \pi_0(c') : A} \quad (\times\text{-}\text{El}_0=) \quad \frac{c = c' : A \times B}{\pi_1(c) = \pi_1(c') : B} \quad (\times\text{-}\text{El}_1=)
$$
The Non-Dependent Function Type

**Formation Rule**

\[
\frac{A : \text{Set} \quad B : \text{Set}}{A \rightarrow B : \text{Set}} \quad (\rightarrow -F)
\]

**Introduction Rule**

\[
\frac{x : A \Rightarrow b : B}{\lambda x^A.b : A \rightarrow B} \quad (\rightarrow -I)
\]

**Elimination Rule**

\[
\frac{f : A \rightarrow B \quad a : A}{f_a : B} \quad (\rightarrow -\text{El})
\]

**Equality Rule**

\[
\frac{x : A \Rightarrow b : B \quad a : A}{(\lambda x^A.b) \ a = b[x := a] : B} \quad (\rightarrow -\text{Eq})
\]
$\beta$-Reduction

- $b[x := a]$ was as for the simply typed $\lambda$-calculus the result of substituting in $b$ every occurrence of variable $x$ by the term $a$ (after renaming of bound variables as usual).

- The equality rule is a symmetric version of $\beta$-reduction

\[
(\lambda x^A.b)\ a \longrightarrow b[x := a]
\]
\[\alpha\text{-Equivalence}\]

As for the simply typed \(\lambda\)-calculus, terms which differ in the choice of bound variables (i.e. which are \(\alpha\)-equivalent) 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 bound variables in types (see later).
The $\eta$-Rule

Again the $\eta$-rule does not fit into the above schema:

$$
\begin{align*}
&f : A \to B \\
&f = \lambda x^A. f \ x : A \to B
\end{align*}
$$

$(\to -\eta)$
Equality Versions of the $\to$-Rules

Equality Version of the Formation Rule

\[
\frac{A = A' : \text{Set} \quad B = B' : \text{Set}}{A \to B = A' \to B' : \text{Set}} \quad (\to -\mathsf{F}=)
\]

Equality Version of the Introduction Rule

\[
\frac{x : A \Rightarrow b = b' : B}{\lambda x^A.b = \lambda x^A.b' : A \to B} \quad (\to -\mathsf{I}=)
\]

Equality Version of the Elimination Rule

\[
\frac{f = f' : A \to B \quad a = a' : A}{f \; a = f' \; a' : B} \quad (\to -\mathsf{El}=)
\]
(d) Structural Rules
Context Rules

The empty context

$$\emptyset \Rightarrow \text{Context} \quad (\text{Context}_0)$$

Extending a context

$$\frac{\Gamma \Rightarrow A : \text{Set}}{\Gamma, x : A \Rightarrow \text{Context}} \quad (\text{Context}_1)$$

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

• The convention that rules can be weakened by a common context does not apply to the rules $(\text{Context}_0)$ and $(\text{Context}_1)$.  

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

\[ \text{N} : \text{Set} \quad (\text{N-F}) \]

With this rule, following the convention on slide 3-67, we have as well introduced the rules

\[
\frac{\Gamma \Rightarrow \text{Context}}{\Gamma \Rightarrow \text{N} : \text{Set}} \quad (\text{N-F})
\]
The following derives \( x : N, y : N, z : N \Rightarrow \text{Context} \)
(Note that \( N : \text{Set} \) is the same as \( \emptyset \Rightarrow N : \text{Set} \):

\[
\begin{align*}
\text{N : Set} & \quad \text{(Context}_1) \\
x : N \Rightarrow \text{Context} & \quad \text{(N-F)} \\
\text{x : N} & \Rightarrow N : \text{Set} \quad \text{(Context}_1) \\
x : N, y : N \Rightarrow \text{Context} & \quad \text{(N-F')} \\
\text{x : N, y : N} & \Rightarrow N : \text{Set} \quad \text{(Context}_1) \\
x : N, y : N, z : N \Rightarrow \text{Context}
\end{align*}
\]
Assumption Rule

\[ \frac{\Gamma, x : A, \Gamma' \Rightarrow \text{Context}}{\Gamma, x : A, \Gamma' \Rightarrow x : A} \quad (\text{Ass}) \]
We extend the derivation of slide 3-81 to a derivation of
\[ x : N, y : N, z : N \Rightarrow y : N: \]

\[
\begin{align*}
\frac{x : N, y : N, z : N \Rightarrow \text{Context}}{x : N, y : N, z : N \Rightarrow y : N} \tag{Ass}
\end{align*}
\]

Similarly we can derive \[ x : N, y : N, z : N \Rightarrow z : N: \]

\[
\begin{align*}
\frac{x : N, y : N, z : N \Rightarrow \text{Context}}{x : N, y : N, z : N \Rightarrow z : N} \tag{Ass}
\end{align*}
\]
The full derivation of first judgement on the previous slide is as follows:

\[
\begin{align*}
\text{N : Set} & \quad \text{(Context}_1) \\
x : \text{N} & \Rightarrow \text{Context} \quad \text{(N-F)} \\
x : \text{N} & \Rightarrow \text{N : Set} \\
\text{N : Set} & \quad \text{(Context}_1) \\
x : \text{N}, y : \text{N} & \Rightarrow \text{Context} \quad \text{(N-F)} \\
x : \text{N}, y : \text{N} & \Rightarrow \text{N : Set} \\
x : \text{N}, y : \text{N}, z : \text{N} & \Rightarrow \text{Context} \quad \text{(Ass)} \\
x : \text{N}, y : \text{N}, z : \text{N} & \Rightarrow y : \text{N}
\end{align*}
\]
Assumption Rule in Agda

- 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

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

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

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

in order to solve this goal.
More generally we might in the derivation of $a : A \Rightarrow \{! !\} : B$ make anywhere use of $a : A$, as long as this is in the context.

\[
\begin{array}{c}
\begin{align*}
\ldots & (\text{Ass}) \\
& a : A \Rightarrow a : A \\
& \vdots \\
& \vdots \\
& a : A \Rightarrow s : B \\
\lambda(a : A).s : A \rightarrow B & (\rightarrow \text{-I})
\end{align*}
\end{array}
\]
Assumption Rule in Agda (Cont.)

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

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

So we have to solve \( a : A \Rightarrow \{! !\} : B \) in order to derive

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

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

(Weak)

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

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

- The judgement \( \Gamma, \Gamma' \Rightarrow \theta \) is weakened by \( \Delta \).
Remark: One can in fact show that the weakening rule can be \textit{weakly derived}.

\textbf{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 directly.

An exception is when we \textit{additionally assume some judgements} for instance $A : \text{Set}$ (corresponding to “postulate” in Agda).

Then $\Gamma \Rightarrow A : \text{Set}$ doesn’t follow without the weakening rule.
We derive \( a : A, b : B \Rightarrow a : A \), assuming we have already derived \( A : \text{Set}, B : \text{Set} \):
Example Deriv.2 (Weak. Rule)

We derive \( x : A \rightarrow (B \times C), a : A \Rightarrow x : A \rightarrow (B \times C) \), assuming we have already derived \( A : \text{Set}, B : \text{Set}, C : \text{Set} \):

\[
\frac{B : \text{Set} \quad C : \text{Set}}{A : \text{Set} \quad B \times C : \text{Set}} \quad \text{(×-F')}
\]

\[
\frac{A : \text{Set} \quad B \times C : \text{Set}}{A \rightarrow (B \times C) : \text{Set}} \quad \text{(→ -F')}
\]

\[
\frac{A \rightarrow (B \times C) : \text{Set}}{x : A \rightarrow (B \times C) \Rightarrow \text{Context}} \quad \text{(Context₁)}
\]

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

\[
\frac{x : A \rightarrow (B \times C) \Rightarrow A : \text{Set}}{x : A \rightarrow (B \times C) \Rightarrow A : \text{Set}} \quad \text{(Context₁)}
\]

\[
\frac{x : A \rightarrow (B \times C), a : A \Rightarrow \text{Context}}{x : A \rightarrow (B \times C), a : A \Rightarrow \text{Context}} \quad \text{(Ass)}
\]

\[
x : A \rightarrow (B \times C), a : A \Rightarrow x : A \rightarrow (B \times C)
\]
General Equality Rules

**Reflexivity**

\[ \begin{align*}
\text{Ref}_\text{Set} & : \forall A : \text{Set}, A = A : \text{Set} \\
\text{Ref}_\text{Elem} & : \forall a : A, a = a : A
\end{align*} \]

(Reflexivity can be weakly derived, except for global assumptions).

**Symmetry**

\[ \begin{align*}
\text{Sym}_\text{Set} & : \forall A = B : \text{Set}, B = A : \text{Set} \\
\text{Sym}_\text{Elem} & : \forall a = b : A, b = a : A
\end{align*} \]
General Equality Rules (Cont.)

Transitivity

\[
\frac{A = B : \text{Set}}{\frac{B = C : \text{Set}}{A = C : \text{Set}}} \quad (\text{Trans}_{\text{Set}})
\]

\[
\frac{a = b : A}{\frac{b = c : A}{a = c : A}} \quad (\text{Trans}_{\text{Elem}})
\]

Transfer

\[
\frac{a : A}{\frac{A = B : \text{Set}}{a : B}} \quad (\text{Transfer}_{0})
\]

\[
\frac{a = b : A}{\frac{A = B : \text{Set}}{a = b : B}} \quad (\text{Transfer}_{1})
\]
Example Deriv. (Gen. Equal. Rules)

\[
\frac{N: \text{Set}}{y: N \Rightarrow \text{Context}} \quad \text{(Context}_1\text{)}
\]
\[
\frac{y: N \Rightarrow N: \text{Set}}{y: N \Rightarrow N: \text{Set}} \quad \text{(N-F)}
\]
\[
\frac{y: N, x: N \Rightarrow \text{Context}}{y: N, x: N \Rightarrow x: N} \quad \text{(Context}_1\text{)}
\]
\[
\frac{y: N \Rightarrow \text{Context}}{y: N \Rightarrow \text{Context}} \quad \text{(Ass)}
\]
\[
\frac{y: N, x: N \Rightarrow x: N}{y: N \Rightarrow y: N} \quad \text{(Ass)}
\]
\[
\frac{y: N \Rightarrow (\lambda x^N.x) \ y = y: N}{y: N \Rightarrow (\lambda x^N.x) \ y = y: N} \quad \text{(SymElem)}
\]
\[
\frac{y: N \Rightarrow y = (\lambda x^N.x) \ y: N}{\lambda y^N \cdot y + 0 = \lambda y^N \cdot (\lambda x^N.x) \ y: N \Rightarrow N} \quad \text{(TransElem)}
\]
\[
\frac{y: N \Rightarrow y + 0 = (\lambda x^N.x) \ y: N}{\lambda y^N \cdot y + 0 = \lambda y^N \cdot (\lambda x^N.x) \ y: N \Rightarrow N} \quad \text{(\rightarrow I=)}
\]
Example Deriv. (Gen. Equal. Rules)

In the previous derivation, the most complicated step was:

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

This is an example of the equality rule for the non-dependent function type (slide 3-73):

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

with \( A := B := N, \ b := x, \ a := y. \)

Therefore \( b[x := a] = y. \)

This instance of the rule was weakened by an additional context \( y : N. \)
Example Deriv. (Gen. Equal. Rules)

Note that from the premises of that rule

\[
\frac{y : N, x : N \Rightarrow x : N \quad y : N \Rightarrow y : N}{y : N \Rightarrow (\lambda x^N.x)} \quad (\Rightarrow \text{-Eq})
\]

we can derive using the introduction and elimination rule

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

as follows:

\[
\frac{y : N, x : N \Rightarrow x : N}{y : N \Rightarrow \lambda x^N.x : N \rightarrow N} \quad (\Rightarrow \text{-I})
\]

\[
\frac{y : N \Rightarrow \lambda x^N.x : N \rightarrow N \quad y : N \Rightarrow y : N}{y : N \Rightarrow (\lambda x^N.x)} \quad (\Rightarrow \text{-El})
\]
Example Deriv. (Gen. Equ. Rules)

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 \( b[x := a] \)).
Substitution Rules

The following rules can be weakly derived:

**Substitution 1**

\[
\frac{\Gamma, x : A, \Gamma' \Rightarrow \theta \quad \Gamma \Rightarrow a : A}{\Gamma, \Gamma'[x := a] \Rightarrow \theta[x := a]} \quad (\text{Subst}_1)
\]

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

**Substitution 2**

\[
\frac{\Gamma, x : A, \Gamma' \Rightarrow B : \text{Set} \quad \Gamma \Rightarrow a = a' : A}{\Gamma, \Gamma'[x := a] \Rightarrow B[x := a] = B[x := a'] : \text{Set}} \quad (\text{Subst}_2)
\]
Substitution Rules

Substitution 3

\[
\frac{\Gamma, x : A, \Gamma' \Rightarrow b : B \quad \Gamma \Rightarrow a = a' : A}{\Gamma, \Gamma'[x := a] \Rightarrow b[x := a] = b[x := a'] : B[x := a]} \quad \text{(Subst}_3)\]

Example Deriv. (Substitution)

\[
\begin{align*}
\frac{
\begin{array}{c}
\cdots \\
(\text{Ass})
\end{array}
\qquad \frac{
\begin{array}{c}
\cdots \\
(\text{Ass})
\end{array}
}{
\begin{array}{c}
x : N, y : N \Rightarrow x : N \\
\text{(N-I\_+) }
\end{array}
\quad \begin{array}{c}
x : N, y : N \Rightarrow y : N \\
\text{(N-I\_+) }
\end{array}
}
\Rightarrow \begin{array}{c}
x + y : N \\
\text{(Subst\_1)}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\frac{
\begin{array}{c}
y : N \\
\text{\_{-1}}
\end{array}
}{
\lambda y^N . 0 + y : N \rightarrow N \\
\text{(\rightarrow-I)}
\end{array}
\end{align*}
\]
Example Deriv. 2 (Substitution)

\[
\frac{N: \text{Set}}{z:N \Rightarrow \text{Context}} \quad \text{(Context}_1\text{)} \\
\frac{N: \text{Set}}{z:N \Rightarrow N: \text{Set}} \quad \text{(N-F)} \\
\frac{z:N, u:N \Rightarrow \text{Context}}{z:N, u:N \Rightarrow u:N} \quad \text{(Context}_1\text{)} \\
\frac{z:N, u:N \Rightarrow u:N}{z:N, u:N \Rightarrow S \ u:N} \quad \text{(N-I}_S\text{)} \\
\]

\[
\frac{z:N \Rightarrow z + 0 = z:N}{z:N \Rightarrow z + 0 = z:N} \quad \text{(Subst}_3\text{)}
\]

\[
\frac{N: \text{Set}}{z:N \Rightarrow \text{Context}} \quad \text{(Context}_1\text{)} \\
\frac{z:N \Rightarrow z:N}{z:N \Rightarrow z + 0 = z:N} \quad \text{(N-E}_{Q_0}\text{)} \\
\]

\[
\frac{z:N, x:N, y:N \Rightarrow x + y:N}{z:N \Rightarrow S (z + 0) = S z:N} \quad \text{(Subst}_3\text{)}
\]

\[
\frac{z:N \Rightarrow \lambda y^N. (S z + 0) + y = S z + y:N}{z:N \Rightarrow \lambda y^N. (S z + 0) + y = S z + y:N} \quad \text{(-I}^=\text{)}
\]

\[
\frac{\lambda z^N \Rightarrow \lambda y^N. (S z + 0) + y = \lambda z^N \Rightarrow \lambda y^N. S z + y:N \Rightarrow N \Rightarrow N}{\lambda z^N \Rightarrow \lambda y^N. (S z + 0) + y = \lambda z^N \Rightarrow \lambda y^N. S z + y:N \Rightarrow N \Rightarrow N} \quad \text{(-I}^=\text{)}
\]
We introduce in the following the dependent versions of the product and the function type.
The Dependent Product

The dependent product is similar as the non-dependent product (e.g. $A \times B$), except that we allow that the second set depends on an element of the first set.

The type theoretic notation is

$$(a : A) \times B$$

Elements of $(a : A) \times B$ are pairs

$$\langle a' , b' \rangle$$

s.t.

- $a' : A$
- $b' : B[a := a']$. 
Example 1 (Dep. Products)

One example for its use are the set of sorted lists:

- \( \text{Sorted}(l) \) is a predicate on \( \text{NatList} \) expressing that \( l \) is sorted.
- An element of

\[
\text{SortedList} := (l : \text{NatList}) \times \text{Sorted}(l)
\]

is a pair

\[\langle l, p \rangle\]

s.t.

- \( l : \text{NatList}, \)
- \( p : \text{Sorted}(l), \) i.e. \( p \) is a **proof** that \( l \) is sorted.

So elements of \( \text{SortedList} \) are lists \( l \) together with a proof that \( l \) is sorted.
Example 2 (Dep. Products)

Let $G$ be the set of genders, informally written

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

Let for $g : G$ the set

$$\text{Names } g$$

be the collection of names of that gender, e.g. informally written

- (Names male) = \{Tom, Jim\},
- (Names female) = \{Jill, Sara\}.
Example 2 (Dep. Products)

- The **set of names with their gender** is the set of pairs \( \langle g, n \rangle \) s.t. \( g \) is a Gender and \( n : (\text{Names } g) \).

- This set is written as

\[ \text{NamesWithGender} := (g : G) \times (\text{Names } g) \]
Rules of the Dependent Product

Formation Rule

\[
\begin{array}{c}
A : \text{Set} \\
x : A \Rightarrow B : \text{Set}
\end{array}
\]
\[
(x : A) \times B : \text{Set}
\]

Introduction Rule

\[
\begin{array}{c}
x : A \Rightarrow B : \text{Set} \\
a : A \\
b : B[x := a]
\end{array}
\]
\[
\langle a, b \rangle : (x : A) \times B
\]
Extra Premise in the Introd. Rule

In the last introduction rule, an extra premise
\[ x : A \Rightarrow B : \text{Set} \] was required.

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

In case of the non-dependent product, this premise was not necessary:
\[ a : A \text{ and } b : B \] indirectly implies that we know \(A : \text{Set}\) and \(B : \text{Set}\) from which it follows \(A \times B : \text{Set}\).
Example

Assuming we have defined the set of genders \( G : \text{Set} \) and the set of names \( g : G \Rightarrow (\text{Name}s \ g) : \text{Set} \), we can introduce the set

\[
\text{NamesWithGender} := (g : G) \times (\text{Name}s \ g) : \text{Set}
\]

by using the formation rule:

\[
\frac{G : \text{Set} \quad g : G \Rightarrow (\text{Name}s \ g) : \text{Set}}{(g : G') \times (\text{Name}s \ g') : \text{Set}} \quad (\times \text{-I})
\]
Example

Furthermore we can introduce

\[ \langle \text{male}, \text{Tom} \rangle : \text{NamesWithGender} \]

as follows:

\[ g : G \Rightarrow (\text{Names } g) : \text{Set} \quad \text{male} : G \quad \text{Tom} : (\text{Names male}) \]

\[ \langle \text{male}, \text{Tom} \rangle : (g : G) \times (\text{Names } g) \]

Note that we need the premise

\[ g : G \Rightarrow (\text{Names } g) : \text{Set} \]

Otherwise we only know that \((\text{Names male}) : \text{Set}\), but not that \((\text{Names female}) : \text{Set}\).
Example

Note that we don’t have

\[ \langle \text{female, Tom} \rangle : \text{NamesWithGender} \]

since we don’t have

\[ \text{Tom} : (\text{Names female}) \]

So here dependent types prevent errors. In an ordinary programming language without dependent types, we can’t define a corresponding type \( \text{NamesWithGender} \) which allows at compile time to define

\[ \langle \text{male, Tom} \rangle : \text{NamesWithGender} \]

but not

\[ \langle \text{female, Tom} \rangle : \text{NamesWithGender} \]
Rules of the Dependent Product

Elimination Rules

\[
\begin{align*}
\cfrac{c : (x : A) \times B}{\pi_0(c) : A} & \quad (\times\text{-El}_0) \\
\cfrac{c : (x : A) \times B}{\pi_1(c) : B[x := \pi_0(c)]} & \quad (\times\text{-El}_1)
\end{align*}
\]

Equality Rules

\[
\begin{align*}
\cfrac{x : A \Rightarrow B : \text{Set}}{} & \quad (\times\text{-Eq}_0) \\
\cfrac{a : A \quad b : B[x := a]}{\pi_0(\langle a, b \rangle) = a : A} & \\
\cfrac{x : A \Rightarrow B : \text{Set}}{} & \quad (\times\text{-Eq}_1) \\
\cfrac{a : A \quad b : B[x := a]}{\pi_1(\langle a, b \rangle) = b : B[x := a]}
\end{align*}
\]

Note that the last two rules require the extra premise \(x : A \Rightarrow B : \text{Set}\) (which is not implied by the premises).
Example

In the "Names-example we have that, if $a : \text{NamesWithGender}$, then $\pi_0(a) : G$ and $\pi_1(a) : (\text{Names } \pi_0(a))$:

\[
\frac{a : (g : G) \times (\text{Names } g)}{\pi_0(a) : G} (\times\text{-El}_0)
\]

\[
\frac{a : (g : G) \times (\text{Names } g)}{\pi_1(a) : (\text{Names } \pi_0(a))} (\times\text{-El}_1)
\]
Rules of the Dependent Product

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

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

- As before, the \( \eta \)-rule expresses that every element of \((x : A) \times B\) is of the form \(\langle \text{something}_0, \text{something}_1 \rangle\).
- The \( \eta \)-rule cannot be derived, if the element in question is a variable.
Equality Versions of the above

Equality Version of the Formation Rule

\[
\begin{align*}
A = A' : \text{Set} & \quad x : A \Rightarrow B = B' : \text{Set} \\
(x : A) \times B = (x : A') \times B' : \text{Set}
\end{align*}
\]  

(\times\text{-F=} )

Equality Version of the Introduction Rule

\[
\begin{align*}
x : A \Rightarrow B : \text{Set} & \quad a = a' : A \\
b = b' : B[x := a]
\end{align*}
\]

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

(\times\text{-I=} )

Equality Versions of the Elimination Rules

\[
\begin{align*}
c = c' : (x : A) \times B & \quad \pi_0(c) = \pi_0(c') : A \\
(\times\text{-El}_{0} )
\end{align*}
\]

\[
\begin{align*}
c = c' : (x : A) \times B & \quad \pi_1(c) = \pi_1(c') : B[x := \pi_0(c)] \\
(\times\text{-El}_{1} )
\end{align*}
\]
The Non-Dep. Product as an Abbrev.

- 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 that the **rules for the non-dependent product are special cases of the rules for the dependent product**.
The Non-Dep. Product as an Abbrev.

More precisely this can be seen as follows:

- From \( A : \text{Set} \) and \( B : \text{Set} \) we can derive \( x : A \Rightarrow B : \text{Set} \) using the \textit{weakening rule}.

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

- From a derivation of \( a : A \) we can derive \( A : \text{Set} \) (we need the concept of presupposition for that, as introduced later).

- Therefore the \textit{premises of the introduction rule for the non-dependent product imply those of the dependent product}.

- Similarly for the elimination, equality and \( \eta \)-rule.
The Dependent Product in Agda

In Agda, the record type allows already dependencies of later sets on previous ones:

Assume $A :: \text{Set}$, and $B :: \text{Set}$, possibly depending on $a :: A$.

Then we can form $\text{sig}\{a :: A; b :: B\}$.

Elements of this type can be introduced in the same way as before, i.e. if $a' :: A$ and $b' :: B[a := a']$ then we can form

$$\text{struct}\{a = a'; b = b'\} :: \text{sig}\{a :: A; b :: B\}.$$

Note that $b' :: B[a := a']$, so the type of $b'$ depends on $a'$.

Furthermore, if $c : \text{sig}\{a :: A; b :: B\}$, then $c.a :: A$ and $c.b :: B[a := c.a]$. 
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:

```agda
data G = male | female

data maleNames = Tom | Jim

data femaleNames = Jill | Sara
```
The “Names”-Example in Agda

Names (g :: G) :: Set
= case g of
   (male) → maleNames
   (female) → femaleNames

NamesWithGender :: Set
= sig
   g :: G
   n :: Names g

See exampleAllNames.agda.
The “Names”-Example in Agda

Note that in the above example we have

\[ \text{Names male } = \text{ maleNames } = \text{ data Tom | Jim} \]
\[ \text{Names female } = \text{ femaleNames } = \text{ data Jill | Sara} \]

Further we have for instance

\[
\text{struct}\{g=\text{male}, n=\text{Tom}\} :: \text{NamesWithGender}
\]

whereas we **don’t** have

\[
\text{struct}\{g=\text{male}, n=\text{Jill}\} :: \text{NamesWithGender}
\]
The Dependent Function Set

In the presence of dependent types we have as well a dependent function set, where the type of the result depends on the argument of the function.

Notation: \((x : A) \rightarrow B\), for the set of functions \(f\) which map an element \(a : A\) to an element of \(B[x := a]\).

In set-theoretic notation this is:

\[
\{ f \mid f \text{ function} \land \text{dom}(f) = A \land \forall a \in A. f(a) \in B[x := a] \}
\]
Example (Dep. Function Set)

Consider the “Names example” as above
\((G : \text{Set set of genders}, \text{Names } g \text{ set of names for gender } g)\).

Define

\[
\begin{align*}
\text{select} : & (g : G) \to (\text{Names } g) \\
\text{select male} & = \text{Tom} \\
\text{select female} & = \text{Jill}
\end{align*}
\]

**select** selects for every gender a name.

**select male** will be an element of
\((\text{Names male}) = (\text{Names } g)[g := \text{male}]\).

It wouldn’t make sense to say \((\text{select male}) : (\text{Names } g)\), without knowing what \(g\) is.
Example (Dep. Function Set)

Note that for instance we don't have

$$\lambda g^G.\text{Tom} : (g : G) \rightarrow (\text{Names } g)$$

since we don't have

$$(\lambda g^G.\text{Tom})\text{ female} : (\text{Names female})$$
Formation Rule

\[ A : \text{Set} \quad x : A \Rightarrow B : \text{Set} \quad (\rightarrow -F') \]

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

Introduction Rule

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

\[ \lambda x^A . b : (x : A) \rightarrow B \quad (\rightarrow -I) \]
Rules of the Dep. Funct. Set

**Elimination Rule**

\[
\frac{f : (x : A) \rightarrow B}{f \ a : B[x := a]} \quad (\rightarrow \text{-El})
\]

**Equality Rule**

\[
\frac{x : A \Rightarrow b : B \quad a : A}{(\lambda x^A.b) \ a = b[x := a] : B[x := a]} \quad (\rightarrow \text{-Eq})
\]
The $\eta$-Rule

The $\eta$-rule has a special status:

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

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

\[
f = \lambda x^A . f\ x : (x : A) \rightarrow B \quad (\rightarrow -\eta)
\]

- As before, the $\eta$-rule expresses that every element of $(x : A) \rightarrow B$ is of the form $\lambda x^A . \text{something}$.
- The $\eta$-rule cannot be derived, if the element in question is a variable.
Equality Versions of the above

Equality Version of the Formation Rule

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

Equality Version of the Introduction Rule

\[
\lambda x^A.b = \lambda x^A.b' : (x : A) \rightarrow B
\]
\[
\rightarrow -I=)\]

Equality Version of the Elimination Rule

\[
f = f' : (x : A) \rightarrow B \quad a = a' : A \quad (\rightarrow -El=)
\]
\[
f a = f' a' : B[x := a]
\]
Non-Dep. Funct. Set as an Abbrev.

The non-dependent function type

\[ A \rightarrow B \]

can be regarded as an abbreviation for the dependent function type

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

where \( B \) does not depend on \( x \).

As for the product one can see that the rules for the non-dependent function set are special cases of the rules for the dependent function set.
The Dep. Function Set in Agda

- We have seen that the non-dependent function type is written as $A \rightarrow B$ in Agda.
- The notation for the dependent function set is $(x :: A) \rightarrow C$. 
The Dep. Function Set in Agda

Elements of \((x :: A) \rightarrow C\) are introduced as before by using

- either \(\lambda\)-abstraction, i.e.
  \[
  \lambda(x :: A) \rightarrow t :: (x :: A) \rightarrow B.
  \]
  Requires that \(t :: B\) depending on \(x :: A\).
  Note that the type \(B\) of \(t\) depends on \(x :: A\).

- or by writing
  \[
  f(x :: A) :: C = \cdots
  \]

Elimination is application using the same notation as before.

E.g., if \(f :: (x :: A) \rightarrow C\) and \(a :: A\), then
\[
f a :: C[x := a].\]
The Dep. Function Set in Agda

- Internally, Agda has only the dependent function set.
- That’s why one often sees in code generated by Agda (e.g. when showing context, when using solve) types of the form

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

where one could use as well

\[A \rightarrow B\]
Abbreviations

We can write

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

instead of

\[(n::N) \rightarrow (m::N) \rightarrow A(n,m)\]
The “select”-Name Example in Agda

The code for the select-Name example in Agda is now as follows (the first definitions are as before):

```
data G  =  male  |  female

data maleNames  =  Tom  |  Jim

data femaleNames  =  Jill  |  Sara
```
The “select”-Name Example in Agda

Names \((g :: G)\) \n:: Set
= \text{case } g \text{ of}
\hspace{1cm} (\text{male}) \rightarrow \text{maleNames}
\hspace{1cm} (\text{female}) \rightarrow \text{femaleNames}

select \:: (g:: G) \rightarrow \text{Names } g
= \lambda \:(g:: G) \rightarrow \text{case } g \text{ of}
\hspace{1cm} (\text{male}) \rightarrow \text{Tom}
\hspace{1cm} (\text{female}) \rightarrow \text{Jill}

See \text{exampleNamesFunction2.agda}. 
The “select”-Name Example in Agda

An attempt to define select s.t. select male is not in maleNames, e.g.

\[
\text{select male} = \text{Jill}
\]

or that select female is not in femaleNames, e.g.

\[
\text{select female} = \text{Tom}
\]

will result in a **type error**.

Jump to full logical framework.
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:

\[ f :: A \to A \]

\[ = \{! !\} \]

Agda is an indentation-sensitive language. The complete definition of \( f \) must be intended otherwise Agda regards this as a new definition.
Example 1 (Cont.)

Step 3:

- We want to derive an element of function type $A \to A$.
- Elements of the function type $A \to A$ are introduced by using $\lambda$-terms.
- If introduced as a $\lambda$-term, the term in question will be of the form $\lambda(a :: A) \to \text{something}$.
- Agda has a command `agda-intro (Intro)` which does this step automatically.
- Has to be executed while the cursor is inside one goal.
Example 1 (Cont.)

Step 3 (Cont)

After executing it we get:

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

(The precise Agda code uses \(\backslash\) instead of \(\lambda\), and \(\rightarrow\) instead of \(\rightarrow\).)
Step 4:

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

Use `agda-solve` (Solve)

We obtain:

\[
\begin{align*}
  f &:: A \rightarrow A \\
  &\equiv \lambda(h :: A) \rightarrow \{!\} 
\end{align*}
\]
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:

```
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:

```
agda-load-buffer (Load Buffer)
```
Otherwise the changes will not be known by Agda.
Step 4 (Cont)

We obtain:

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

- \( A \) :: Set.
- \( f :: A \rightarrow A \).

See next slide.

- \( a :: A \).
  
  - Since we are defining 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).
Termination Check (Cont.)

- 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 Turing halting problem is undecidable).
Example 1 (Cont.)

Step 5 (Cont.)

Now everything with result type $A$ (i.e. which has at the right side of 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 $A :: \text{Set}$. This corresponds to assuming that we have already derived $A : \text{Set}$.

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

$$f :: A \rightarrow A$$
$$= \{! \quad !\}$$

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 \( \lambda(a :: A) \rightarrow \{! !\} \):

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

In terms of rules this means that we replace \( d_0 \) by \( \lambda a^A.d_1 \) which is derived by an introduction rule

\[
\frac{a : A \Rightarrow d_1 : A}{\lambda a^A.d_1 : A \rightarrow A} \rightarrow -I
\]
Example 1, Using Rules (Cont.)

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

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

In terms of rules this means that we replace \(d_1\) by \(a\).
\(a : A \Rightarrow a : A\) follows by an assumption rule:

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

The assumption rule will be discussed later.

Essentially it allows to derive if \(x : B\) occurs in the context that \(x : B\) holds.
Example 2

We consider a derivation of

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

(See example exampleSampleDerivation2.agda).

Step 1:

We postulate \( A \):

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

We state our goal:

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

\texttt{agda-load-buffer (Load Buffer)}

so that Agda realizes this change:

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

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

Step 3:

- The type of the new goal is $A$, which is the result type of the function 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 have as result the type of the goal in question.
Example 2 (Cont.)

Step 3 (Cont):

Therefore we type into the goal \( x \) and use **agda-refine (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

\[
f :: ((A \to A) \to A) \to A \\
= \lambda(x :: (A \to A) \to A) \to x \{! !\}
\]
Step 4:

The type of the new goal is $A \rightarrow A$.

This is since $x :: (A \rightarrow A) \rightarrow A$ needs to be applied to an element of type $A \rightarrow A$ in order to obtain an element of type $A$.

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

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

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

Step 4 (Cont)

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

\[
f :: ((A \rightarrow A) \rightarrow A) \rightarrow A \\
= \lambda(x :: (A \rightarrow A) \rightarrow A) \rightarrow x \left( \lambda(h :: A) \rightarrow \{! !\} \right)
\]

We wanted to define an element of $A \rightarrow A$, so the domain of the $\lambda$ term 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 $\lambda(a :: A) \to \{! !\}$ should have type $A \to A$, so $\{! !\}$ must have type $A$.
- The context contains $A :: \text{Set}$, $f$, $x$ and $a$.

- We can use both $x$ and $a$ here.
- There is usually more than one solution for proceeding in Agda.

This means that we sometimes have to backtrack and try a different solution.
Example 2 (Cont.)

Step 5 (Cont.)

We try $a :: A$. After inserting it and using \texttt{agda-refine (Refine)} we obtain the following and are done.

$$f :: ((A 	o A) 	o A) 	o A$$

$$= \lambda(x :: (A 	o A) 	o A) \to x (\lambda(a :: A) \to a)$$
Example 2, Using Rules

- Postulating $A :: \text{Set}$ corresponds to that we assume that we have already derived $A : \text{Set}$.

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

$$d_0 : ((A \rightarrow A) \rightarrow A) \rightarrow A$$
Example 2, Using Rules

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

This corresponds to replacing \( d_0 \) by
\[
\lambda(x :: (A \to A) \to A).d_1
\]
and having as last step an introduction rule:

\[
\frac{x : (A \to A) \to A \Rightarrow d_1 : A}{\lambda x((A \to A) \to A).d_1 : ((A \to A) \to A) \to A} \ (\to\ -I)
\]
Example 2, Using Rules

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

- This corresponds to replacing `d_1` by `x d_2`, and using one elimination rule in order to derive it:

\[
\begin{array}{c}
x: (A \rightarrow A) \rightarrow A \Rightarrow x: (A \rightarrow A) \rightarrow A \\
x: (A \rightarrow A) \rightarrow A \Rightarrow d_2: A \rightarrow A
\end{array}
\]  
\[\text{\rightarrow-El}\]

\[
\begin{array}{c}
x: (A \rightarrow A) \rightarrow A \Rightarrow x d_2: A \\
\lambda x: (A \rightarrow A) \rightarrow A . x d_2: ((A \rightarrow A) \rightarrow A) \rightarrow A
\end{array}
\]  
\[\text{\rightarrow-I}\]

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

We then used intro on the goal which was then replaced by \( \lambda (x :: A) \rightarrow \{!\} \).

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

\[
\begin{align*}
\frac{}{x : (A \rightarrow A) \rightarrow A \Rightarrow x : (A \rightarrow A) \rightarrow A} & \quad (\rightarrow \text{-I}) \\
\frac{}{x : (A \rightarrow A) \rightarrow A \Rightarrow x : (A \rightarrow A) \rightarrow A} & \quad (\rightarrow \text{-I}) \\
\frac{x : (A \rightarrow A) \rightarrow A \Rightarrow x : (A \rightarrow A) \rightarrow A}{(\lambda x^{(A \rightarrow A)} . x) \ (\lambda a^A . d_3) : ((A \rightarrow A) \rightarrow A) \rightarrow A} & \quad (\rightarrow \text{-I})
\end{align*}
\]
Example 2, Using Rules

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

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

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

The right hand derivation can again be derived by an **assumption 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 $A, B$:
  
  \[
  \text{postulate } A :: \text{Set} \\
  \text{postulate } B :: \text{Set}
  \]

- We introduce the product of $A, B$:
  - This will be a record with element $a : A, b : B$.
  
  \[
  AB :: \text{Set} \\
  = \text{sig}\{a :: A; b :: B\}
  \]
Example 3 (Cont.)

Step 2:

Our goal is:

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

**Step 3:**

- We use intro.
- An element of $A \rightarrow B \rightarrow AB$ will be of the form

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

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 $a, b$ are used as 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 rule.
- Elements of type $AB$ introduced by the introduction principle will have the form

```
struct {a = {! !};
b = {! !}}
```
Example 3 (Cont.)

**Step 4 (Cont):**

When using intro we get:

\[
\begin{align*}
f &:: A \rightarrow B \rightarrow AB \\
&= \lambda(a' :: A) \rightarrow \lambda(b' :: B) \rightarrow \text{struct}\{a = \{! !\}; \\
&\quad b = \{! !\}\}
\end{align*}
\]
Example 3 (Cont.)

**Step 5:**

The first goal has as context:

- \( A, B : \text{Set}, \)
- \( AB : \text{Set}, \)
- \( f : A \rightarrow B \rightarrow AB, \)
- \( a' : A, \)
- \( b' : B, \)
- \( a : A, \)
- \( b : B. \)

\( a : A, b : B \) 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.
We insert \( a \), use refine and solve the first goal:

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

\[
= \lambda(a' :: A) \rightarrow \lambda(b' :: B) \rightarrow \text{struct}\{a = a'; b = \{! !\}\}
\]
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 (assuming that we have already derived $A : \text{Set}, B : \text{Set}$):

$$
\frac{A : \text{Set} \quad B : \text{Set}}{A \times B : \text{Set}} (\times\text{-F})
$$

We won’t use this however, since it is required for the assumption rules only, 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 \( \lambda a'^A . \lambda b'^B . d_1 \)
which is introduced by two introduction rules:

\[
\begin{align*}
  a' : A, b' : B & \Rightarrow d_1 : A \times B \\
  a' : A & \Rightarrow \lambda b'^B . d_1 : B \rightarrow (A \times B) \\
  \lambda a'^A . \lambda b'^B . d_1 : A \rightarrow B \rightarrow (A \times B)
\end{align*}
\]

\[
(\rightarrow -I) \\
(\rightarrow -I)
\]
Using intro again means that we replace \( d_1 \) by \( \langle d_2, d_3 \rangle \), which can be introduced by an introduction rule:

\[
\array{c}
\frac{a' : A, b' : B \Rightarrow d_2 : A \quad a' : A, b' : B \Rightarrow d_3 : B}{a' : A, b' : B \Rightarrow \langle d_2, d_3 \rangle : A \times B} (\times\text{-I})
\end{array}
\]

\[
\frac{a' : A \Rightarrow \lambda b'^B. \langle d_2, d_3 \rangle : B \to (A \times B)}{\lambda a'^A. \lambda b'^B. \langle d_2, d_3 \rangle : A \to B \to (A \times B)} (\to\text{-I})
\]
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 \(c\):

\[
\begin{align*}
    a' : A, b' : B & \Rightarrow a' : A & a' : A, b' : B & \Rightarrow b' : B & (\times - I) \\
    a' : A, b' : B & \Rightarrow \langle a', b' \rangle : A \times B & (\rightarrow - I) \\
    a' : A & \Rightarrow \lambda b' : B. \langle a', b' \rangle : B \rightarrow (A \times B) & (\rightarrow - I) \\
    \lambda a' : A. \lambda b' : B. \langle a', b' \rangle : A & \rightarrow B \rightarrow (A \times B) 
\end{align*}
\]

- The premises require an assumption rule (which will use the derivation of \(A \times B\)), see later for details.
Example 4

We derive an element of type

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

where \(BC\) is the product of \(B\) and \(C\). (See exampleProductElim.agda).
**Step 1:**

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

  - postulate $A :: \text{Set}$
  - postulate $B :: \text{Set}$
  - postulate $C :: \text{Set}$

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

  - $BC :: \text{Set}$
  - $= \text{sig}\{b :: B;\}
  - \quad c :: C\}$
Step 2:

Our goal is:

\[
\begin{align*}
f &: (A \rightarrow BC) \rightarrow A \rightarrow B \\
&= \{! !\}
\end{align*}
\]
Example 4 (Cont.)

Step 3:

We use intro and get (after using agda-solve and renaming of variables):

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

\[ = \lambda(x :: A \rightarrow BC) \rightarrow \lambda(a :: A) \rightarrow \{! !\} \]
**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 to $B$.
- We introduce first an element of type $BC$ by a let-expression, and then derive from it the desired element of type $B$: 
Example 4 (Cont.)

Step 4 (Cont):

Using *agda-let* (*Make let expression*) with variable $bc$ we obtain:

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

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

$$= \{! !\}$$

in $\{! !\}$
Step 5:
We insert as type of variable \( bc \) the type \( BC \) (using refine) and obtain:

\[
\begin{align*}
  f &: (A \to BC) \to A \to B \\
  &= \lambda(x :: A \to BC) \to \lambda(a :: A) \to \text{let } bc :: BC \\
  &= \{! !\} \\
  \text{in } &\{! !\}
\end{align*}
\]
Example 4 (Cont.)

Step 6:

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

\[
f :: (A \to BC') \to A \to B \\
= \lambda(x :: A \to BC') \to \lambda(a :: A) \to \text{let } bc :: BC \\
= x \{! !\} \\
\text{in } \{! !\}
\]
Step 7:
The new goal can be solved by refining it with variable $a$:

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

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

$$= x \ a$$

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

**Step 8:**

Currently, Agda doesn’t have any direct support for refining $bc$ to an element of type $B$.

We have to do this by hand, insert $bc.b$, choose refine and obtain:

\[ f :: (A \rightarrow BC') \rightarrow A \rightarrow B \]
\[ = \lambda(x :: A \rightarrow BC') \rightarrow \lambda(a :: A) \rightarrow \text{let } bc :: BC \]
\[ = x \ a \]
\[ \text{in } bc.b \]
Example 4 (Cont.)

- 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 $(x \ a)$.
- We get

$$f :: (A \rightarrow BC) \rightarrow A \rightarrow B = \lambda(x :: A \rightarrow BC) \rightarrow \lambda(a :: A) \rightarrow (x \ a).b$$
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:

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

$$\frac{x : A \rightarrow (B \times C) \Rightarrow \lambda a^A.d_1 : A \rightarrow B}{\lambda x^{A \rightarrow (B \times C)}. \lambda a^A.d_1 : (A \rightarrow (B \times C)) \rightarrow A \rightarrow B} \quad (\rightarrow -I)$$
In Agda, we then replace the goal corresponding to $d_1$ by $(x\ a).b$.

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

This can be introduced by two applications of elimination rules:

\[
\begin{align*}
\frac{x: A \to (B \times C), a: A \Rightarrow x: A \to (B \times C) \quad x: A \to (B \times C), a: A \Rightarrow a: A}{\rightarrow\text{-El}} \\
\frac{x: A \to (B \times C), a: A \Rightarrow x\ a: B \times C}{\times\text{-El}} \\
\frac{x: A \to (B \times C), a: A \Rightarrow \pi_0(x\ a): B}{\rightarrow\text{-I}} \\
\frac{x: A \to (B \times C) \Rightarrow \lambda a^A.\pi_0(x\ a): A \to B}{\rightarrow\text{-I}}
\end{align*}
\]

The two initial judgements can be introduced by assumption rules.
Change of Structure of this Section

The original Subsection 3 (g) has been moved to Sect. 4. Therefore the new structure of this section is as follows:

(a) Judgements.
(b) Basic form of rules.
(c) The non-dependent function type and product.
(d) Structural rules. (Omitted 2005).
(e) The dependent function type and product.
(f) Derivations vs. Agda code. (Omitted 2005).
(g) Presuppositions (Omitted 2005).
(h) The full logical framework

JumpOverPresuppositions
(g) Presuppositions

In order to derive \( x : A, y : B \Rightarrow C : \text{Set} \) we need to show:

- \( A : \text{Set} \).
- \( x : A \Rightarrow B : \text{Set} \).

So the judgement

\[
x : A, y : B \Rightarrow C : \text{Set}
\]

implicitly contains the judgements

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

\[
x : A \Rightarrow B : \text{Set}.
\]
Presuppositions (Cont.)

\[ A : \text{Set} \quad \text{and} \quad x : A \Rightarrow B : \text{Set} \quad \text{are presuppositions of the judgement} \]

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

A → B : Set .

and of the judgement

A × B : Set .

The next slide shows the presuppositions of judgements.
## Presuppositions

<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{Set} ).</td>
</tr>
<tr>
<td>( \Gamma \Rightarrow A : \text{Set} )</td>
<td>( \Gamma \Rightarrow \text{Context} ).</td>
</tr>
<tr>
<td>( \Gamma \Rightarrow A = B : \text{Set} )</td>
<td>( \Gamma \Rightarrow A : \text{Set} ), ( \Gamma \Rightarrow B : \text{Set} ).</td>
</tr>
</tbody>
</table>
## Presuppositions

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Presuppositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma \Rightarrow a : A)</td>
<td>(\Gamma \Rightarrow A : \text{Set}).</td>
</tr>
<tr>
<td>(\Gamma \Rightarrow a = b : A)</td>
<td>(\Gamma \Rightarrow a : A,) (\Gamma \Rightarrow b : A).</td>
</tr>
</tbody>
</table>
### Presuppositions

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Presuppositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \Rightarrow (x : A) \to B : \text{Set}$</td>
<td>$\Gamma, x : A \Rightarrow B : \text{Set}$.</td>
</tr>
<tr>
<td>$\Gamma \Rightarrow (x : A) \times B : \text{Set}$</td>
<td>$\Gamma, x : A \Rightarrow B : \text{Set}$.</td>
</tr>
</tbody>
</table>
Presuppositions

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:

- \( \emptyset \Rightarrow \text{Context}, \)
- \( A : \text{Set}, \)
- \( x : A \Rightarrow \text{Context}, \)
- \( x : A \Rightarrow B : \text{Set}, \)
- \( x : A, y : B \Rightarrow \text{Context}, \)
- \( x : A, y : B \Rightarrow C : \text{Set}, \)
- \( x : A, y : B, z : C \Rightarrow \text{Context}, \)
- \( x : A, y : B, z : C \Rightarrow D : \text{Set}, \)
- \( x : A, y : B \Rightarrow (z : C) \times D : \text{Set}, \)
- \( x : A, y : B \Rightarrow a : (z : C) \times D, \)
- \( x : A, y : B \Rightarrow b : (z : C) \times D. \)
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 $A \rightarrow B : \text{Set}$ (which abbreviates $\emptyset \Rightarrow A \rightarrow B : \text{Set}$) are:
  - $\emptyset \Rightarrow \text{Context}$,
  - $A : \text{Set}$,
  - $x : A \Rightarrow \text{Context}$,
  - $x : A \Rightarrow B : \text{Set}$.
We would like to add operations on types, such as

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

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

The problem is that for this we need

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

and our rules allow this only if we had

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

as a rule results however in an inconsistent theory: using this rule we can prove everything, especially false formulas.

The corresponding paradox is called Girard’s paradox.
Jean-Yves Girard
Instead we introduce a new level on top of Set called **Type**.

So besides judgements $A : \text{Set}$ we have as well judgements of the form

$$A : \text{Type}$$

One rule will especially express

$$\text{Set} : \text{Type}$$

Elements of **Type** are **types**, elements of **Set** are **small types**.
Set (Cont.)

- We add rules asserting that \textbf{if} A: Set \textbf{then} A: Type.
- Further we add rules asserting that Type is closed under the dependent function type and product.
- \textbf{Since} Set : Type \textbf{we get therefore} (by closure under the function type)

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

and we can \textit{assign to} \texttt{prod} \textit{above the type}

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

(The definition of \texttt{prod} \textit{will be given later.})
Set and Type

Type

Set → Set × Set

N → N × N

N → Set × N

Set
However, we cannot use \texttt{prod} in order to form the product of two sets, i.e. we cannot introduce

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

since \texttt{Set : Set} does not hold.
Rules for Set

Formation Rule for Set

Set : Type \quad \text{(SetIsType)}

Every Set is a Type

\[
\frac{A : \text{Set}}{A : \text{Type}} \quad \text{(Set2Type)}
\]
Closure of Type

Further we add rules stating that Type is closed under the dependent function type and the dependent product:

Closure of Type under the dependent product

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

Closure of Type under the dependent function type

\[
\frac{A : \text{Type} \quad x : A \Rightarrow B : \text{Type}}{(x : A) \rightarrow B : \text{Type}} \quad (\rightarrow \text{-} \text{FType})
\]
Nondependent Case

A special case of the above rule is the closure under the non-dependent function type and product. This rule can be derived (e.g. from the premises one can derive using the other rules the conclusion).

**Closure of Type under the non-dependent product**

\[
\frac{A : \text{Type} \quad B : \text{Type}}{A \times B : \text{Type}} \quad (\times \text{-F}_{\text{Type}})
\]

**Closure of Type under the non-dependent function type**

\[
\frac{A : \text{Type} \quad B : \text{Type}}{A \rightarrow B : \text{Type}} \quad (\rightarrow \text{-F}_{\text{Type}})
\]
Equality Versions of the Rules

Formation Rule for Set

\[ \text{Set} = \text{Set : Type} \quad (\text{SetIsType}^=) \]

Every Set is a Type

\[
\frac{A = B : \text{Set}}{A = B : \text{Type}} \quad (\text{Set2Type}^=)
\]
Equality Versions of the Rules

Closure of Type under the dependent product

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

Closure of Type under the dependent function type

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

Similarly for the non-dependent versions of the above.

Jump over Context Rule.
Context Rules

- The types in the contexts, which were before only elements of $\text{Set}$, can now be as well elements of $\text{Type}$.
- Therefore we need an additional context rule

\[
\frac{\Gamma \Rightarrow A : \text{Type}}{\Gamma, x : A \Rightarrow \text{Context}} \quad (\text{Context}_1^{\text{Type}})
\]
Example: prod

We can now introduce $\text{prod} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$:

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

\[
\text{Set} : \text{Type} \quad \quad \text{(Context$_1$)}
\]

\[
\frac{X : \text{Set} \Rightarrow \text{Context}}{X : \text{Set} \Rightarrow \text{Set} : \text{Type}} \quad \quad \text{(SetIsType)}
\]

\[
\frac{X : \text{Set}, Y : \text{Set} \Rightarrow \text{Context}}{X : \text{Set}, Y : \text{Set} \Rightarrow X : \text{Set}} \quad \quad \text{(Context$_1$)}
\]

\[
\frac{X : \text{Set}, Y : \text{Set} \Rightarrow \text{Context}}{X : \text{Set}, Y : \text{Set} \Rightarrow Y : \text{Set}} \quad \quad \text{(Ass)}
\]

Similarly we derive $X : \text{Set}, Y : \text{Set} \Rightarrow Y : \text{Set}$. 
Now we can derive our desired judgement:

\[
\begin{align*}
X : \text{Set}, Y : \text{Set} \Rightarrow X : \text{Set} & \quad X : \text{Set}, Y : \text{Set} \Rightarrow Y : \text{Set} \\
\hline
X : \text{Set}, Y : \text{Set} \Rightarrow X \times Y : \text{Set} & \quad (\times -F) \\
\hline
X : \text{Set} \Rightarrow \lambda Y : \text{Set}. X \times Y : \text{Set} \rightarrow \text{Set} & \quad (\rightarrow -I) \\
\hline
\lambda (X, Y : \text{Set}). X \times Y : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set} & \quad (\rightarrow -I)
\end{align*}
\]

So define

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

If one wants to form

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

one needs to have a further level \text{Kind} above \text{Type}, s.t.

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

Then

\[
\text{Type} \rightarrow \text{Type} \rightarrow \text{Type} : \text{Kind}.
\]
Hierarchy of Types (Set, Type, Kind)

Type

Kind

Set → Set

N → N

N → Set

Set

Type → Type

CS_336/CS_M36 (part 2) Interactive Theorem Proving; Lentterm 2005, Sec. 3 (h)
Rules for Type as a Kind

Type is a Kind

Type : Kind

Every Type is a Kind

\[
\frac{A : \text{Type}}{A : \text{Kind}} \quad \text{(Type2Kind)}
\]
Closure of Kind

Closure of Kind under the dependent product

\[
\frac{A : \text{Kind} \quad x : A \Rightarrow B : \text{Kind}}{(x : A) \times B : \text{Kind}} \quad (\times^\text{-FKind})
\]

Closure of Kind under the dependent function type

\[
\frac{A : \text{Kind} \quad x : A \Rightarrow B : \text{Kind}}{(x : A) \rightarrow B : \text{Kind}} \quad (\rightarrow^\text{-FKind})
\]

Plus equality versions of the above rules.
Again, the context rules have to be expanded:

\[
\frac{\Gamma \Rightarrow A : \text{Kind}}{\Gamma, x : A \Rightarrow \text{Context}} \quad (\text{Context}_{1}^{\text{Kind}})
\]
Hierarchies of Types (Cont.)

- This can be iterated further, forming
  \[ \text{Type} = \text{Type}_1, \text{Kind} = \text{Type}_2, \text{Type}_3, \text{Type}_4 \ldots \]

- Agda has a hierarchy of types built in, written as \#0 (which is \text{Set}), \#1 (which is \text{Type}), \#2 (in the rule calculus called \text{Kind}), \#3 etc.

- So we have
  - \#0 = \text{Set} : \text{Type},
  - \#0 = \text{Set} : \#2, \#1 = \text{Type} : \#2,
  - \#0 = \text{Set} : \#3, \#1 = \text{Type} : \#3, \#2 : \#3,
  - \#0 = \text{Set} : \#4, \#1 = \text{Type} : \#4, \#2 : \#4, \#3 : \#4,
  - etc.
Hierarchy of Types (#0, #1, #2, ...)

Type\(_3 = #3\)

Kind = Type\(_2 = #2\)

Type = Type\(_1 = #1\)

Set = #0
Changes To Presuppositions

If we have the two type levels $\text{Set}$ and $\text{Type}$, the presuppositions change.

E.g. the presupposition of $\Gamma \Rightarrow a : A$ is no longer $A : \text{Set}$ but $A : \text{Type}$.

It might be that the derivation derives actually $A : \text{Set}$, but that implies $A : \text{Type}$.

But it might be that we can only derive $A : \text{Type}$.

Therefore the presuppositions have to be changed as in the following table.
## Presuppositions (with Set, Type)

<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>$\Gamma \Rightarrow A : \text{Set}$</td>
<td>$\Gamma \Rightarrow A : \text{Type}$.</td>
</tr>
<tr>
<td>$\Gamma \Rightarrow A : \text{Type}$</td>
<td>$\Gamma \Rightarrow \text{Context}$</td>
</tr>
</tbody>
</table>
## Presuppositions (with Set, Type)

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Presuppositions</th>
</tr>
</thead>
</table>
| $\Gamma \Rightarrow A = B : \text{Set}$ | $\Gamma \Rightarrow A : \text{Set}$,  
| & $\Gamma \Rightarrow B : \text{Set}$,  
| & $\Gamma \Rightarrow A = B : \text{Type}$. |
| $\Gamma \Rightarrow A = B : \text{Type}$ | $\Gamma \Rightarrow A : \text{Type}$,  
| & $\Gamma \Rightarrow B : \text{Type}$. |
| $\Gamma \Rightarrow a : A$ | $\Gamma \Rightarrow A : \text{Type}$. |
# Presuppositions (with Set, Type)

<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, ) ( \Gamma \Rightarrow b : A. )</td>
</tr>
<tr>
<td>( \Gamma \Rightarrow (x : A) \times B : \text{Set} )</td>
<td>( \Gamma \Rightarrow A : \text{Set}, ) ( \Gamma, x : A \Rightarrow B : \text{Set}. )</td>
</tr>
<tr>
<td>( \Gamma \Rightarrow (x : A) \times B : \text{Type} )</td>
<td>( \Gamma, x : A \Rightarrow B : \text{Type}. )</td>
</tr>
</tbody>
</table>
### Presuppositions (with Set, Type)

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Presuppositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \Rightarrow (x : A) \rightarrow B : \text{Set}$</td>
<td>$\Gamma \Rightarrow A : \text{Set},$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma, x : A \Rightarrow B : \text{Set}.$</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>
Changes To Presuppositions

If we have more levels (\text{Kind} or \#i), then the presuppositions have to be changed again.

E.g., if we have levels \text{Set}, \text{Type}, \text{Kind}, the presupposition

- of $\Gamma \Rightarrow A : \text{Set}$ is $\Gamma \Rightarrow A : \text{Type}$,
- of $\Gamma \Rightarrow A : \text{Type}$ is $\Gamma \Rightarrow A : \text{Kind}$,
- of $\Gamma \Rightarrow A : \text{Kind}$ is $\Gamma \Rightarrow \text{Context}$.