5. The Logical Framework

(a) Judgements.
(b) Basic form of rules.
(c) The non-dependent function type and product.
(d) Structural rules. (Omitted 2008).
(e) The dependent function set and $\forall$-quantification.
(f) The dependent product and $\exists$-quantification.
(g) Derivations vs. Agda code. (Omitted 2008).
(h) Presuppositions (Omitted 2008).
(i) The full logical framework

---

Judgements

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

\[ \text{a : A} \]

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

\[ A : \text{Set} \]

Before deriving $a : A$, we first have to show $A : \text{Set}$.

So any derivation of $a : A$ contains implicitly a derivation of $A : \text{Set}$.

---

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).

Jump over example.
Example

- postulate \( A : \text{Set} \)
- postulate \( a : A \)
- postulate \( P : A \rightarrow \text{Set} \)

\[
g : A \rightarrow A
g a = a
\]

\[
a' : A
a' = g a
\]

\[
p : P a \rightarrow P a'
p x = \{! !\}
\]

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:

\[
\begin{array}{c}
a : A \\
b : B
\end{array}
\begin{array}{c}
\pi_{0}(\langle a, b \rangle) = a : A
\end{array}
\]

\( \rightarrow \) is not directed, so we have as well the rule

\[
\begin{array}{c}
a = b : A
\end{array}
\begin{array}{c}
b = a : A
\end{array}
\]

\( \text{Sym}_{\text{Elem}} \)

We can therefore derive:

\[
\begin{array}{c}
a : A \\
b : B
\end{array}
\begin{array}{c}
\pi_{0}(\langle a, b \rangle) = a : A
\end{array}
\begin{array}{c}
a = \pi_{0}(\langle a, b \rangle) : A
\end{array}
\]

\( \text{Sym}_{\text{Elem}} \)

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
\[
x : f a'
\]

Jump over next examples.

Examples (Equality of Types)

More precisely this follows by the following derivation (the equality rule used here will be introduced in Subsect. (d)).

\[
\begin{align*}
\frac{f : A \to A \quad a = a' : A}{x : f a \\ \quad f a = f a' : \text{Set} \\ \quad x : f a'}
\end{align*}
\]

Four Judgements

So we have the following 4 types of judgements:

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

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

- As for the simply typed \(\lambda\)-calculus, in dependent type theory, judgements might depend on a **context**.

- So we obtain judgements of the form

\[
\begin{align*}
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow A : \text{Set} \quad (\text{Later, when we introduce higher types, this requirement has to be replaced by } A_1 : \text{Type}) \\
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow A = B : \text{Set} \\
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow a : A \\
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow a = b : A
\end{align*}
\]

Need for Context Judgements

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

\[
\ldots
\]

- To derive such judgements requires that we know

\[
\begin{align*}
A_1 & : \text{Set} \\
x_1 : A_1 & \Rightarrow A_2 : \text{Set} \\
x_1 : A_1, x_2 : A_2 & \Rightarrow A_3 : \text{Set} \\
\ldots
\end{align*}
\]

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

Context Judgement

- 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.:

\[
\begin{align*}
A_1 : \text{Set} \\
x_1 : A_1 & \Rightarrow A_2 : \text{Set} \\
x_1 : A_1, x_2 : A_2 & \Rightarrow A_3 : \text{Set} \\
\ldots
\end{align*}
\]

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

- 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 a statement \(a : A\) (of which we don’t know whether it is type correct) an expression \(B\) s.t.

\[
a : A \text{ holds iff } B : \text{Set holds.}
\]
**Five Dependent Judgements**

We have therefore 5 dependent judgements:

\[ x_1 : A_1, \ldots, x_n : A_n \Rightarrow A : \text{Set} \]
\[ x_1 : A_1, \ldots, x_n : A_n \Rightarrow A = B : \text{Set} \]
\[ x_1 : A_1, \ldots, x_n : A_n \Rightarrow a : A \]
\[ x_1 : A_1, \ldots, x_n : A_n \Rightarrow a = b : A \]
\[ x_1 : A_1, \ldots, x_n : A_n \Rightarrow \text{Context} \]

**Notations for Judgements, Contexts**

\( \theta \) (pronounced “theta”) will in the following denote an arbitrary non-dep. 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 \( \theta \).

---

**Example**

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

\[ \Gamma, x : \sigma, \Delta \Rightarrow x : \sigma \quad (\text{if } x : \tau \text{ does not occur in } \Delta \text{ for any } \tau) \]

reads in dependent type theory as follows (assuming that \( x : B \) does not occur in \( \Delta \) for any \( B \)):

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

\[ \Gamma, x : A, \Delta \Rightarrow x : A \]

Similarly we have to deal with the rule introducing constants.

---

**Contexts in Agda**

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

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

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 (environment).

See exampleShowContext.agda.
Example: Derivation of double

(See exampleDoubleString2.agda.)

- We derive

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

- We start with

\[
\text{double} : \text{String} \rightarrow \text{String} \\
\text{double } s = \{! !\}
\]

- We can insert into the goal concat:

\[
\text{double} : \text{String} \rightarrow \text{String} \\
\text{double } s = \{! \text{ concat } !\}
\]

Example: Derivation of double

- We insert \(x\) into the first goal and refine:

\[
\text{double} : \text{String} \rightarrow \text{String} \\
\text{double } s = \text{concat } x \{! !\}
\]

- Doing the same with the second goal gives:

\[
\text{double} : \text{String} \rightarrow \text{String} \\
\text{double } s = \text{concat } x \ x
\]

- 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

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

is as follows:

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

\[
\begin{array}{c}
\emptyset : \text{Context} \\
\text{String} : \text{Set}
\end{array}
\]

\[
\frac{}{x : \text{String} \Rightarrow \text{Context}} \quad (\text{Context}_1)
\]

We derive \( x : \text{String} \Rightarrow \text{concat} \) using the previous derivation:

\[
\begin{array}{c}
x : \text{String} \Rightarrow \text{Context} \\
x : \text{String} \Rightarrow x : \text{String}
\end{array}
\]

\[
\frac{}{x : \text{String} \Rightarrow \text{concat} \, x \, x : \text{String} \Rightarrow \text{String}} \quad (\text{Ass})
\]

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

\[
\begin{array}{c}
\text{concat} : \text{String} \Rightarrow \text{String} \\
x : \text{String} \Rightarrow \text{Context}
\end{array}
\]

\[
\frac{}{x : \text{String} \Rightarrow \text{concat} : \text{String} \Rightarrow \text{String} \Rightarrow \text{String}} \quad (\text{Weak})
\]

\[
\frac{x : \text{String} \Rightarrow x : \text{String}}{x : \text{String} \Rightarrow \text{concat} : \text{String} \Rightarrow \text{String} \Rightarrow \text{String}} \quad (\rightarrow \text{El})
\]

\[
\frac{x : \text{String} \Rightarrow \text{concat} : \text{String} \Rightarrow \text{String} \Rightarrow \text{String} \Rightarrow \text{String}}{x : \text{String} \Rightarrow \text{concat} : \text{String} \Rightarrow \text{String} \Rightarrow \text{String}} \quad (\rightarrow \text{I})
\]

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

(b) Basic Form of Rules

Four Kinds of Rules

We derive \( x : \text{String} \Rightarrow x : \text{String} \Rightarrow x : \text{String} \Rightarrow \text{String} \) using the previous derivations:

\[
\begin{array}{c}
x : \text{String} \Rightarrow \text{concat} : \text{String} \Rightarrow \text{String} \Rightarrow \text{String} \\
x : \text{String} \Rightarrow \text{concat} : \text{String} \Rightarrow \text{String} \Rightarrow \text{String} \\
x : \text{String} \Rightarrow \text{concat} : \text{String} \Rightarrow \text{String} \Rightarrow \text{String}
\end{array}
\]

\[
\frac{x : \text{String} \Rightarrow x : \text{String} \Rightarrow x : \text{String} \Rightarrow \text{String}}{x : \text{String} \Rightarrow x : \text{String} \Rightarrow x : \text{String} \Rightarrow \text{String}} \quad (\rightarrow \text{El})
\]

The remaining derivation using the above derivations is as follows:

\[
\begin{array}{c}
x : \text{String} \Rightarrow \text{concat} : \text{String} \Rightarrow \text{String} \Rightarrow \text{String} \\
x : \text{String} \Rightarrow \text{concat} : \text{String} \Rightarrow \text{String} \Rightarrow \text{String} \\
x : \text{String} \Rightarrow \text{concat} : \text{String} \Rightarrow \text{String} \Rightarrow \text{String}
\end{array}
\]

\[
\frac{x : \text{String} \Rightarrow \text{concat} : \text{String} \Rightarrow \text{String} \Rightarrow \text{String} \Rightarrow \text{String}}{x : \text{String} \Rightarrow \text{concat} : \text{String} \Rightarrow \text{String} \Rightarrow \text{String} \Rightarrow \text{String}} \quad (\rightarrow \text{I})
\]

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

- The **formation rules** introduce new sets or types.
- Each set and 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 **set-constructor**, 
  \( a_1, \ldots, a_n \) are its arguments.
  \( n = 0 \) is possible.
- Later, we will introduce higher levels **Type**, **Kind**, etc.
- Then we have formation rules with conclusion
  \[ C \ a_1 \cdots a_n : \text{Type} \text{ (or: Kind, etc.)} \] and \( C \) is called a **Type-constructor**, **Kind-constructor**, etc.

Logical Framework

- 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 \( \text{List} \) itself, which would have type
  \[ \text{List} : \text{Set} \rightarrow \text{Set} \]
  The problem is that \( \text{Set} \rightarrow \text{Set} \) 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
  \[ \text{List} : \text{Set} \rightarrow \text{Set} \]
  similarly for all other set formation constructors.

Logical Framework

- Agda has the logical framework built in, so in Agda \( \text{List} \) will be a function \( \text{Set} \rightarrow \text{Set} \), in Agda notation:
  \[
  \begin{align*}
  \text{List} & : \text{Set} \rightarrow \text{Set} \\
  \text{List} \ A &= \{! !\}
  \end{align*}
  \]
Example 1: The Set of Lists

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

- The **set-constructor** is **List**.
- List \( A \) is the set of lists of elements of \( A \).
- The \( F \) in the label \((\text{List-F})\) stands for **Formation rule**.

Ex. 2: The Set of Natural Numbers

- Formation rule for the set of natural numbers:

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

- The **set-constructor** is **\( \mathbb{N} \)**.
- Note that the formation rule for \( \mathbb{N} \) has 0 premises (therefore the fraction bar is omitted).

Formation Rules in Agda

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

Example 1:

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

\[
\text{List} \ A = \{! !\}
\]
Example 2: $(\times)$

- Agda syntax for introducing the non-dependent product:

$$
\_ \times \_ : \text{Set} \to \text{Set} \to \text{Set}
$$

$$
A \times B = \{! !\}
$$

Introduction Rule, Example 1a

- The set $\text{NatList}$ of lists of natural numbers with formation rule

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

has two introduction rules:

$$
[] : \text{NatList} \quad \text{(NatList-I[])}
$$

$$
\begin{align*}
n : \mathbb{N} & \quad l : \text{NatList} \\
n \_ : l & : \text{NatList} \quad \text{(NatList-I::)}
\end{align*}
$$

- The $1$ in the labels $(\text{NatList-I[]})$, $(\text{NatList-I::})$ stands for Introduction rule.

Jump to Example 2

(2) Introduction Rules

- The introduction rule introduces elements of a set.
- The conclusion of such a rule will have the form

$$
C \ a_1 \ldots a_n : A
$$

where

- $A$ is a set 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).

Introduction Rule, Example 1b

- We generalise the previous example to lists of arbitrary set.
- Lists of elements in $A$ have two introduction rules:

$$
\begin{align*}
A : \text{Set} & \quad \text{(List-I[])} \\
[]_A & : \text{List} A
\end{align*}
$$

$$
\begin{align*}
A : \text{Set} & \quad a : A \\
A \_ : l & : \text{List} A \quad \text{(List-I::)}
\end{align*}
$$

- 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 [ ] : NatList, similarly for _ :: _. We will usually ignore this distinction, if it doesn’t cause confusion.

Canonical Elements

- Canonical elements of a set 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)).
  - [ ], (1 + 1) :: (concat (0 :: [ ]) [ ]) in case of NatList.

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:
  \[
  0 : \mathbb{N} \quad \text{(N-I}_0) \\
  n : \mathbb{N} \quad \text{S } n : \mathbb{N} \quad \text{(N-I}_S)
  \]

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 set, if it reduces to a canonical element of that set.
- Each element of a set (depending on the empty context) in dependent type theory will either be a canonical or a non-canonical element of that set.
- 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 $\mathbb{N}$.

Example 2: Addition in $\mathbb{N}$

\[
\frac{n : \mathbb{N}}{n + m : \mathbb{N}} \quad \text{(N-El_+)}
\]
- Equality rules will express
  - $n + 0 = n$.
  - $n + S\, m = S\,(n + m)$.
- 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.

(3) Elimination Rules

- Elimination rules allow to take an element of a set and compute from it an element of another set.
- 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 (indicated by label El) are the first and second projections:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi_0(c) : A$</td>
<td>$c : A \times B$</td>
</tr>
<tr>
<td>$\pi_1(c) : B$</td>
<td>$c : A \times B$</td>
</tr>
</tbody>
</table>

- The equality rules will express $\pi_0(\langle a, b \rangle) = a$,
  $\pi_1(\langle a, b \rangle) = b$.

Elimination in Agda

- Elimination for builtin sets has special notation.
- For user defined sets, i.e. those introduced using data, elimination is realized by pattern matching.
- Example: Definition of addition in $\mathbb{N}$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+__ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$</td>
<td>$n + Z = n$</td>
</tr>
<tr>
<td>$n + S, m = S,(n + m)$</td>
<td>$n + S, m = S,(n + m)$</td>
</tr>
</tbody>
</table>
(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 $0 : \mathbb{N}$ and then eliminate it by using $(\mathbb{N}-\text{El}_+)$ 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 $n + 0 = n : \mathbb{N}$ instead.

The equality rule (indicated by label $\text{Eq}$) expresses this:

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

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

The corresponding equality rule is therefore:

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

Example (Equality Rule)

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:

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

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:

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

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 $a : A$).

Jump over next examples
Example (Equality Rule, Cont)

- 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 3 (Equality Rule)

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

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

$n + S \, m : \mathbb{N}$ can be derived by first introducing $S \, m : \mathbb{N}$ and then by eliminating it using $+$:

\[
\frac{n : \mathbb{N} \quad m : \mathbb{N}}{n + S \, m : \mathbb{N}} \quad (\mathbb{N}\text{-I}_S)
\]

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

Example 2 (Equality Rule)

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

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

$n + 0 : \mathbb{N}$ can be derived by first introducing $0 : \mathbb{N}$

(this is an introduction rule with no premises, i.e. an axiom)

and then by eliminating it using $+$, using the following derivation:

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

The equality rule explain how to reduce $n + 0$.

Equality Rules in Agda

- Equality Rules in Agda are implicit.

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

\[
\text{+_+_} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}
\]

\[
\begin{align*}
n + Z &= n \\
n + S \, m &= S \, (n + m)
\end{align*}
\]

- 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 set.
- Reduce it to canonical form.

For instance in case of $A \times B$, (Red) are the reductions
- $\pi_0((a, b)) \rightarrow a$.
- $\pi_1((a, b)) \rightarrow b$.

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

$$(1 + 1) + (1 + 0) \rightarrow (1 + 1) + 1 \equiv (1 + 1) + S \ 0$$

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

- Example: Equality version of the formation rule for \( \mathbb{N} \) (degenerated):
  
  \[
  \frac{}{\mathbb{N} = \mathbb{N} : \text{Set}} \quad (\text{N-F=})
  \]

Equality Versions of Rules

- Example: Equality version of the introduction rules for List:
  
  \[
  \frac{A = A' : \text{Set}}{[\ ]_A = [\ ]_{A'} : \text{List } A} \quad (\text{List-I[ ]=})
  \]

- Example: Equality version of the elimination rule for \((+)\), \( \mathbb{N} \):
  
  \[
  \frac{n = n' : \mathbb{N} \quad m = m' : \mathbb{N}}{n + m = n' + m' : \mathbb{N}} \quad (\text{N-El}_{+})
  \]

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

Common Contexts

- The convention is that all rules can as well be weakened by a common context.
- This means that when introducing a rule

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

  we implicitly introduce as well the following rules

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

- 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 $\times$:

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

can be weakened as follows:

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

Example (Cont.)

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

$$
\frac{x : A, y : A \Rightarrow y : A}{x : A \Rightarrow \lambda y^A. y : A \Rightarrow A} \quad (\rightarrow\text{-}\text{I})
\frac{\lambda x^A. \lambda y^A. y : A \rightarrow A}{\lambda x^A. \lambda y^A. y : A \rightarrow A \rightarrow A} \quad (\rightarrow\text{-}\text{I})
$$

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 $\theta \equiv 0 : \mathbb{N}$

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:

$$
\frac{\Gamma \Rightarrow \text{Context}}{\Gamma \Rightarrow \theta}
$$

Example

The formation rule for $\mathbb{N}$

$$
\frac{}{\mathbb{N} : \text{Set}} \quad (\text{N-F})
$$

will be weakened as follows:

$$
\frac{\Gamma \Rightarrow \text{Context}}{\Gamma \Rightarrow \mathbb{N} : \text{Set}} \quad (\text{N-F})
$$
We introduce in the following non-dependent versions of the product and the function set.

### The Non-Dependent Product

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

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

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

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

### Equality Versions of the $\times$-Rules

**Equality Version of the Formation Rule**
\[
\frac{A = A' : \text{Set} \quad B = B' : \text{Set}}{A \times B = A' \times B' : \text{Set}} (\times^\text{F}^=)
\]

**Equality Version of the Introduction Rule**
\[
\frac{a = a' : A \quad b = b' : B}{\langle a, b \rangle = \langle a', b' \rangle : A \times B} (\times^\text{I}^=)
\]

**Equality Versions of the Elimination Rules**
\[
\frac{c = c' : A \times B}{\pi_0(c) = \pi_0(c') : A} (\times^\text{El}_0^=) \quad \frac{c = c' : A \times B}{\pi_1(c) = \pi_1(c') : B} (\times^\text{El}_1^=)
\]

### The $\eta$-Rule

The $\eta$-rule does not fit into the above schema:
\[
c : A \times B \quad \frac{c = \langle \pi_0(c), \pi_1(c) \rangle : A \times B} (\times^\eta)
\]
The Non-Dependent Function Type

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

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

**Introduction Rule**
\[ x : A \Rightarrow b : B \quad (\rightarrow - I) \]

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

**Elimination Rule**
\[ f : A \rightarrow B \quad a : A \quad (\rightarrow - \text{El}) \]

\[ f a : B \]

**Equality Rule**
\[ x : A \Rightarrow b : B \quad a : A \quad (\rightarrow - \text{Eq}) \]

\[ (\lambda x : A. b) a = b[x := a] : B \]

As for the typed \( \lambda \)-calculus, \( \lambda x^A.b \) is an abbreviation for \( \lambda (x : A).b \).

\( \alpha \)-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).

\( \beta \)-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

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

The \( \eta \)-Rule

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

\[ f : A \rightarrow B \quad (\rightarrow - \eta) \]

\[ f = \lambda x^A.f x : A \rightarrow B \]
Equality Versions of the →-Rules

Equality Version of the Formation Rule

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

Equality Version of the Introduction Rule

\[ x : A \Rightarrow b = b' : B \]
\[ \lambda x.A : b = \lambda x.A' : b' : A \to B \quad (\to I=) \]

Equality Version of the Elimination Rule

\[ f = f' : A \to B \quad a = a' : A \]
\[ f a = f' a' : B \quad (\to \text{El}=) \]

Jump over subsection on structural rules

(d) Structural Rules

Context Rules

The empty context

\[ \emptyset \Rightarrow \text{Context} \quad (\text{Context}_0) \]

Extending a context

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

The convention that rules can be weakened by a common context does not apply to the rules (Context0) and (Context1).

Example Derivation (Context Rules)

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

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

With this rule, following the convention on the previous slide we have as well introduced the rules

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

\[ \Gamma \Rightarrow N : \text{Set} \quad (\text{N-F}) \]
Assumption Rule

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

- **Side condition**: \( \Delta \) must not bind \( x \) again:
  - \( \Delta \) must not be of the form \( \Delta', x : B, \Delta'' \) for some \( \Delta', B, \Delta'' \).
  - Otherwise the assumption \( x : B \) would override the assumption \( x : A \).
  - If \( x : B \) occurs in \( \Delta \), we can only conclude

    \[ \frac{\Gamma, x : A, \Delta \Rightarrow x : B'}{\Gamma, x : A, \Delta \Rightarrow x : B'} \]

    only for the last occurrence of \( x : B' \) in \( \Delta \).

---

Example Deriv. (Assumpt. Rule)

- We extend the derivation of

  \[ x : N, y : N, z : N \Rightarrow Context \]

  above to a derivation of \( x : N, y : N, z : N \Rightarrow y : N \):

  \[ \frac{x : N, y : N, z : N \Rightarrow Context}{x : N, y : N, z : N \Rightarrow y : N} \]  \text{(Ass)}

- Similarly we can derive \( x : N, y : N, z : N \Rightarrow z : N \):

  \[ \frac{x : N, y : N, z : N \Rightarrow Context}{x : N, y : N, z : N \Rightarrow z : N} \]  \text{(Ass)}

---

Example Deriv. (Assumpt. Rule)

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

  \[ \frac{N : Set}{x : N \Rightarrow Context} \]  \text{(Context$_1$)}

  \[ \frac{x : N \Rightarrow N : Set}{x : N, y : N \Rightarrow Context} \]  \text{(N-F)}

  \[ \frac{x : N, y : N \Rightarrow N : Set}{x : N, y : N, z : N \Rightarrow Context} \]  \text{(Context$_1$)}

  \[ \frac{x : N, y : N, z : N \Rightarrow y : N}{x : N, y : N, z : N \Rightarrow y : N} \]  \text{(Ass)}

---

Assumption Rule in Agda

- When we define a function:

  \[ f : A \rightarrow B \]

  \[ f a = \{! !\} \]

  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 \( \{! !\} : B \).
Assumption Rule in Agda (Cont.)

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.

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

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

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

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

Assumption Rule in Agda (Cont.)

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.

\[
\frac{\ldots}{a : A \Rightarrow a : A} \quad (\text{Ass})
\]

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

Weakening Rule

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

\( \theta \) 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 weakly derived.

Weakly derived means: whenever the assumptions of the rule can be derived in the complete set of rules we provide, then as well the conclusion.

However, this can’t be derived from the premise the conclusion directly.

An exception is when we 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.

---

**Example Deriv. (Weak. Rule)**

We derive $a : A, b : B \Rightarrow a : A$, under the global assumptions $A : \text{Set}, B : \text{Set}$:

\[
\begin{align*}
A : \text{Set} & \quad \text{Ass} \quad \text{(Weak)} \\
\quad & \quad \text{(Context$_1$)} \\
\quad & \quad \text{(Context$_1$)} \\
\quad & \quad \text{(Weak)} \\
\quad & \quad \text{(Context$_1$)} \\
\quad & \quad \text{(Weak)} \\
\end{align*}
\]

---

**General Equality Rules**

**Reflexivity**

\[
\begin{align*}
A : \text{Set} & \Rightarrow A = A : \text{Set} \quad \text{(Refl$_\text{Set}$)} \\
a : A & \Rightarrow a = a : A \quad \text{(Refl$_\text{Elem}$)}
\end{align*}
\]

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

**Symmetry**

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

Transitivity

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

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

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

\[
a = c : A
\]

Transfer

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

\[
a : B
\]

\[
a = b : A \quad A = B : \text{Set} \quad (\text{Transfer}_{1})
\]

\[
a = b : B
\]

Example Deriv. (Gen. Equal. Rules)

In the previous derivation, the most complicated step was:

\[
y : \mathbb{N}, x : \mathbb{N} \Rightarrow x : \mathbb{N} \quad y : \mathbb{N} \Rightarrow y : \mathbb{N} \quad (\rightarrow \text{-Eq})
\]

\[
y : \mathbb{N} \Rightarrow (\lambda x. y) y = y : \mathbb{N}
\]

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

\[
x : A \Rightarrow b : B \quad a : A \quad (\rightarrow \text{-Eq})
\]

\[
(\lambda x. b) a = b[x := a] : B
\]

with \( A := \mathbb{N}, b := x, a := y. \)

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

This instance of the rule was weakened by an additional context \( y : \mathbb{N}. \)

Example Deriv. (Gen. Equal. Rules)

Note that from the premises of that rule:

\[
y : \mathbb{N}, x : \mathbb{N} \Rightarrow x : \mathbb{N} \quad y : \mathbb{N} \Rightarrow y : \mathbb{N} \quad (\rightarrow \text{-Eq})
\]

\[
y : \mathbb{N} \Rightarrow (\lambda x. y) y = y : \mathbb{N}
\]

we can derive using the introduction and elimination rule:

\[
y : \mathbb{N} \Rightarrow (\lambda x. y) y : \mathbb{N}
\]

as follows:

\[
y : \mathbb{N}, x : \mathbb{N} \Rightarrow x : \mathbb{N} \quad (\rightarrow \text{-I})
\]

\[
y : \mathbb{N} \Rightarrow \lambda x. y : \mathbb{N} \Rightarrow y : \mathbb{N} \quad (\rightarrow \text{-El})
\]

\[
y : \mathbb{N} \Rightarrow (\lambda x. y) y : \mathbb{N}
\]
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**

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

($\Gamma'[x := a]$ 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 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)

$$\begin{align*}
\cdots &\Rightarrow x : N, y : N \\
\Rightarrow x : N &\Rightarrow y : N \\
\Rightarrow y : N &\Rightarrow x + y : N \\
\Rightarrow 0 : N &\Rightarrow \lambda y^N.0 + y : N \\
\end{align*}$$

$$\begin{align*}
\cdots &\Rightarrow x : N, y : N \\
\Rightarrow x : N &\Rightarrow y : N \\
\Rightarrow y : N &\Rightarrow 0 + y : N \\
\Rightarrow \lambda y^N.0 + y : N &\Rightarrow \lambda y^N.0 + y : N \\
\end{align*}$$
Example Deriv. 2 (Substitution)

\[ \begin{align*}
\vdash & \; \mathsf{N} \vdash \mathsf{Context} \quad \text{(Context)} \\
\vdash & \; z : \mathsf{N} \Rightarrow \mathsf{N} \quad \text{(N-P)} \\
\vdash & \; x : \mathsf{N} \Rightarrow \mathsf{N} \quad \text{(N-I_S)} \\
\vdash & \; y : \mathsf{N} \\
\vdash & \; \mathsf{Context} (\mathsf{Ass}) \\
\vdash & \; z : \mathsf{N} \Rightarrow \mathsf{N} \quad \text{(N-Eq_0)} \\
\vdash & \; x + y : \mathsf{N} \quad \text{(Subst_3)}
\end{align*} \]

... 

\[ \begin{align*}
\vdash & \; z : \mathsf{N} \Rightarrow \mathsf{N} \\
\vdash & \; y : \mathsf{N} \\
\vdash & \; \mathsf{Context} (\mathsf{Subst}) \\
\vdash & \; z : \mathsf{N} \\
\vdash & \; y : \mathsf{N} \\
\vdash & \; x + y : \mathsf{N} \quad \text{(Subst_3)}
\end{align*} \]

Example (Dep. Function Set)

1. Let Gender be the set of genders, informally written
   \[
   \text{Gender} = \{\text{female, male}\}.
   \]
2. In Agda, Gender would be defined by
   \[
   \text{data Gender : Set where}
   \]
   \[
   \begin{align*}
   \text{female} & : \text{Gender} \\
   \text{male} & : \text{Gender}
   \end{align*}
   \]

(e) The Depend. Function Set and $\forall$

The dependent function set is similar to the non-dependent function set (e.g. $A \rightarrow B$), except that we allow that the second set to depend on an element of the first set.

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:

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

Example (Dep. Function Set)

Let for $g : \text{Gender}$ the set

\[
\text{Name } g
\]

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

- Name female = \{jill, sara\},
- Name male = \{tom, jim\}. 

Example (Dep. Function Set)

More formally, \( \text{Name} \) can be defined in Agda as follows:

```agda
data MaleName : Set where
tom : MaleName
jim : MaleName

data FemaleName : Set where
jill : FemaleName
sara : FemaleName

Name : Gender → Set
Name male = MaleName
Name female = FemaleName
```

Example (Dep. Function Set)

Define

```agda
select : (g : Gender) → Name \ g
select female = jill
select male = tom
```

- \( \text{select} \) selects for every gender a name.
- \( \text{select female} \) will be an element of \( \text{Name female} = \text{Name} \ [(\text{Name} \ g) \ [g := \text{female}].} \)
- It wouldn’t make sense to say \( \text{select female} : \text{Name} \ g, \) without knowing what \( \ g \) is.

Example (Dep. Function Set)

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

```
select male = jill
```

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

```
select female = tom
```

will result in a **type error**.

Example (Dep. Function Set)

Note that for instance we **don’t** have

```
λ\ g \text{Gender}. \text{tom} : (g : \text{Gender}) → \text{Name} \ g
```

since we **don’t** have

```
(λ\ g \text{Gender}. \text{tom}) \text{female} : \text{Name female}
```

```agda
Example (Dep. Function Set)

CS_M36 (part 2)/CS_M46 Interactive Theorem Proving, Lent Term 2008, Sec. 5 (e) 5-105
```
Rules of the Dep. Funct. Set

**Formation Rule**

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

**Introduction Rule**

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

The \(\eta\)-Rule

The \(\eta\)-rule has a special status:

\[
\begin{array}{c}
\eta\text{-Rule} \\
\hline
f : (x : A) \rightarrow B \\
\frac{f = \lambda x^A.f}{(x : A) \rightarrow B} \quad (\rightarrow \text{-}\eta)
\end{array}
\]

- 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**

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

**Equality Version of the Introduction Rule**

\[
\begin{array}{c}
\frac{x : A \Rightarrow b = b' : B}{\lambda x^A.b = \lambda x^A.b' : (x : A) \rightarrow B} \quad (\rightarrow \text{-I=})
\end{array}
\]

**Equality Version of the Elimination Rule**

\[
\begin{array}{c}
f = f' : (x : A) \rightarrow B \\
\frac{f a = f' a' : B[x := a]}{a = a' : A} \quad (\rightarrow \text{-El=})
\end{array}
\]
Non-Dep. Funct. Set as an Abbrev.

The non-dependent function set

\[ A \rightarrow B \]

can be regarded as an abbreviation for the dependent function set

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

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 set is written as \(A \rightarrow B\) in Agda.

The notation for the dependent function set is \((x : A) \rightarrow B\).

The Dep. Function Set in Agda

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

- either \(\lambda\)-abstraction, i.e. we can define

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

or shorter (if Agda – as in most cases – can work the type \(A\) of \(x\))

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

Requires that \(b : B\) depending on \(x : A\).

Note that the type \(B\) of \(b\) depends on \(x : A\).
The Dep. Function Set in Agda

- Elimination is application using the same notation as before.
  - E.g., if $f : (x : A) \to B$ and $a : A$, then $f a : B[x := a]$.

$(x : A) \to \cdots \textbf{vs.} \; \lambda(x : A) \to \cdots$

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

Abbreviations

- We can write $(n \; m : \mathbb{N}) \to A$ instead of $(n : \mathbb{N}) \to (m : \mathbb{N}) \to A$

$(x : A) \to \cdots \textbf{vs.} \; \lambda(x : A) \to \cdots$

- $\lambda(x : A) \to t$ is a term.
  - the function, mapping an element $x : A$ to the element $t$.
  - It does not make sense to say $s$ is an element of a function.
  - Correspondingly it does not make sense to talk about $s : (\lambda(x : A) \to t)$.
  - $(\lambda(x : A) \to t)$ never occurs in a position where a set/type is required.
  - It therefore never occurs on the right hand side of $\; ::$.
  - It does however make sense to talk about $(\lambda(x : A) \to t) : B$ for some set (or type) $B$. 
We have already seen how to represent the propositional connectives and decidable atomic formulae in Agda and therefore as well in dependent type theory:

- Implication
  \[ A \rightarrow B \]
is represented as the nondependent function set

- Conjunction
  \[ A \land B \]
is represented as one of the two versions of the product of \( A \) and \( B \).

Disjunction will be introduced later (as the disjoint union).

- \( \neg A \) has been introduced as \( A \rightarrow \bot \).

- If \( f : A_1 \rightarrow \cdots \rightarrow A_n \rightarrow \text{Bool} \) is a function, we can represent the predicate “\( f \text{ } a_1 \cdots a_n \text{ is true} \)” as

  \[ \text{Atom} \left( f \text{ } a_1 \cdots a_n \right) \]

We will investigate, how to represent universal and (in the next section) existential quantification in dependent type theory.

- Since we have many types, we have to write when using quantifiers explicitly the type, the bound variable is ranging over:
  We write therefore

  - \( \forall x : A. B \) or \( \forall^A x. B \) for “for all \( x \) of type \( A \), \( B \) holds” (where \( B \) usually depends on \( x \));

  - \( \exists x : A. B \) or \( \exists^A x. B \) for “there exists an \( x \) of type \( A \), s.t. \( B \) holds” (again \( B \) usually depends on \( x \)).
Universal Quantification

∀x^A.B is true iff, for all x : A there exists a proof of B (with that x).

Therefore a proof of ∀x^A.B is a function, which takes an x:A and computes an element of B.

Therefore the set of proofs of ∀x^A.B is the set of functions, mapping an element x : A to an element of B.

This set is just the dependent function set (x : A) → B.

Therefore we can identify ∀x^A.B with (x : A) → B.

∀ in Agda

∀x^A.B is represented by (x : A) → B in Agda.

Remember that ∀x : A.B is another notation for ∀x^A.B.

As an example,

we define a < -operation on Bool using ff < tt is true and b < b' is false, otherwise.

Then we show ∀x^Bool.¬(x < x).

See exampleLessBool.agda.

Example (∀, Cont.)

First we define a Boolean valued less-than relation on Bool as follows:

_<_Bool_ : Bool → Bool → Bool
ff <Bool b = b
tt <Bool _ = ff

This means that <Bool has the following truth table:

<table>
<thead>
<tr>
<th>&lt;Bool</th>
<th>ff</th>
<th>tt</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
<td>ff</td>
<td>tt</td>
</tr>
<tr>
<td>tt</td>
<td>ff</td>
<td>ff</td>
</tr>
</tbody>
</table>

Explanation of this definition:

If we identify ff with the number 0, tt with 1, then b <Bool b' means that for the corresponding numbers we have b < b'.

Especially we have:

if a is false, then a is less than b iff b is true, so the truth value of a <Bool b is the same as b.

if a is true, then a is never less than b.
The above defines the same function as the following long version:

\[
\begin{align*}
_\text{<_Boollong}_ & : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\
\text{ff <Boollong } & = \text{ff} \\
\text{tt <Boollong } & = \text{ff}
\end{align*}
\]

Proving properties for \(<\text{Boollong}\) is more complicated since the proof usually requires the same more complicated splitting up into cases.

It is usually easier to proof properties for versions of functions, in which the number of case distinctions is reduced to a minimum.

We introduce \(\neg\):

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

\[
\neg A = A \rightarrow \bot
\]

The statement that \(<\) is antireflexive is

\[
\forall a : \text{Bool}. \neg(a < a)
\]

which is represented in Agda as follows:

\[
\text{Lemma4} : \text{Set} \\
= (a : \text{Bool}) \rightarrow \neg(a < a)
\]
Lemma 4 : Set

\[\forall (a : \text{Bool}) \rightarrow \neg (a < a)\]

Since \(\neg (a < a) = (a < a) \rightarrow \bot\), we have

\[\text{Lemma 4} = (a : \text{Bool}) \rightarrow \neg (a < a)\]

\[= (a : \text{Bool}) \rightarrow (a < a) \rightarrow \bot\]

Example (\(\forall\), Cont.)

We want to prove Lemma 4.

A proof of Lemma 4 will be an element

\[\text{lemma}4 : \text{Lemma 4}\]

\[\text{lemma}4 = \{! !\}\]

The type of goal is \((a : \text{Bool}) \rightarrow (a < a) \rightarrow \bot\).

We need to make use of our assumptions, namely

\[a : \text{Bool} \text{ and } aa : a < a.\]

\[a < b\] is defined by case disjunction on \(a\) and \(b\).

Unless we know that \(a = \text{tt}\) or \(a = \text{ff}\), we don’t know much about \(a < a\).

So it seems to be a good step to make pattern matching using the cases \(a = \text{tt}\) and \(a = \text{ff}\).
The type of both goals is the same as before, namely \( \bot \), since it didn’t depend on \( a \).

However, we know now more about the assumptions \( aa : a < a \).

In case of \( a = \text{ff} \), we have \( aa : (a < a) = (\text{ff} < \text{ff}) = \bot \).

So there is no case for \( aa : \bot \), and we can solve this case by

\[ \text{lemma4 \ ff ()} \]

We obtain the code

\[ \text{lemma4 \ : \ Lemma4} \]
\[ \text{lemma4 \ ff ()} \]
\[ \text{lemma4 \ tt ()} \]

In the previous example,

- the type of goal was \( \bot \),
- and \( aa : \bot \).

So, instead of using case distinction on \( aa \) we could have as well inserted \( aa \) in those goals:

\[ \text{lemma4 \ : \ Lemma4} \]
\[ \text{lemma4 \ ff \ aa = aa} \]
\[ \text{lemma4 \ tt \ aa = aa} \]
The Dependent Product and \( \exists \)

The dependent product is similar as the non-dependent product (e.g. \( A \times B \)), except that we allow that the second set to depend 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:

- 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)

Remember the Gender-example as in the last section:

- Gender = \{female, male\}.
- For \( g : \text{Gender} \), \( \text{Name} g \) is a collection of names of that gender, e.g.
  - Name female = \{jill, sara\},
  - Name male = \{tom, jim\}.
- The set of names with their gender is the set of pairs \( \langle g, n \rangle \) s.t. \( g \) is a Gender and \( n : \text{Name} g \).
- This set is written as

\[
\text{NameWithGender} := (g : \text{Gender}) \times \text{Name} g
\]

Rules of the Dependent Product

**Formation Rule**

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

**Introduction Rule**

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

Example

- Assuming we have defined the set of genders $\text{Gender} : \text{Set}$ and the set of names $g : \text{Gender} \Rightarrow \text{Name } g : \text{Set}$, we can introduce the set $\text{NameWithGender} := (g : \text{Gender}) \times \text{Name } g : \text{Set}$ by using the formation rule:

  \[
  \begin{array}{c}
  \text{Gender} : \text{Set} \\
  g : \text{Gender} \Rightarrow \text{Name } g : \text{Set}
  \end{array}
  \quad
  \Rightarrow
  \quad
  (g : \text{Gender}) \times \text{Name } g : \text{Set}
  \] (×-1)

Example

- Furthermore we can introduce $\langle \text{male}, \text{tom} \rangle : \text{NameWithGender}$

  as follows:

  \[
  \frac{g : \text{Gender} \Rightarrow \text{Name } g : \text{Set}}{\langle \text{male}, \text{tom} \rangle : (g : \text{Gender}) \times \text{Name } g}
  \] (×-1)

- Note that we need the premise $g : \text{Gender} \Rightarrow \text{Name } g : \text{Set}$

  Otherwise we only know that $\text{Name male} : \text{Set}$, but not that $\text{Name female} : \text{Set}$.

Note that we **don’t** have $\langle \text{female}, \text{tom} \rangle : \text{NameWithGender}$

since we **don’t** have $\text{tom} : \text{Name female}$

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

$\langle \text{male}, \text{tom} \rangle : \text{NameWithGender}$

but not

$\langle \text{female}, \text{tom} \rangle : \text{NameWithGender}$
### Elimination Rules

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

### Equality Rules

\[
\frac{x : A \Rightarrow B : \text{Set}}{a : A \quad b : B[x := a]} \quad (\times - \text{Eq}_0)
\]
\[
\frac{x : A \Rightarrow B : \text{Set}}{a : A \quad b : B[x := a]} \quad (\times - \text{Eq}_1)
\]

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

### Example

#### In the "Name"-example we have that, if \( a : \text{NameWithGender} \), then \( \pi_0(a) : \text{Gender} \) and \( \pi_1(a) : \text{Name} \pi_0(a) \):

\[
\frac{a : (g : \text{Gender}) \times \text{Name} g}{\pi_0(a) : \text{Gender}} \quad (\times - \text{El}_0)
\]
\[
\frac{a : (g : \text{Gender}) \times \text{Name} g}{\pi_1(a) : \text{Name} \pi_0(a)} \quad (\times - \text{El}_1)
\]

#### Furthermore

\[\pi_0(\langle\text{male}, \text{tom}\rangle) = \text{male} : \text{Gender}\]

therefore

\[\text{Name} \pi_0(\langle\text{male}, \text{tom}\rangle) = \text{Name male}\]

\[\pi_1(\langle\text{male}, \text{tom}\rangle) = \text{tom} : \text{Name} \pi_0(\langle\text{male}, \text{tom}\rangle)\]

therefore as well

\[\pi_1(\langle\text{male}, \text{tom}\rangle) = \text{tom} : \text{Name male}\]

#### Rules of the Dependent Product

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

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

- 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

\[
\frac{A = A': \text{Set}}{x : A \Rightarrow B = B': \text{Set}} \quad (\times=^1)
\]

Equality Version of the Introduction Rule

\[
\frac{x : A \Rightarrow B : \text{Set}}{a = a': A \quad b = b': B[x := a]} \quad \langle a, b \rangle = \langle a', b' \rangle : (x : A) \times B \quad (\times=^1)
\]

Equality Versions of the Elimination Rules

\[
\begin{align*}
\pi_0(c) &= \pi_0(c') : A & c = c' : (x : A) \times B \\
\pi_1(c) &= \pi_1(c') : B[x := \pi_0(c)] & (\times=^1)
\end{align*}
\]

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.

Jump to the dependent product in Agda.

The Non-Dep. Product as an Abbrev. (part 2)

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 weakening rule.
- Therefore the 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 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

\[
\begin{align*}
\text{record AB : Set where} \\
\text{field} \\
\quad a & : A \\
\quad b & : B
\end{align*}
\]
The Dependent Product in Agda

record AB : Set where
  field
  a : A
  b : B

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

\[
\text{record } \{ a : A = a'; b : B = b' \} : AB .
\]

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

Furthermore, if \( ab : AB \), then
\[
AB.a \ ab : A,
AB.b \ ab : B[a := AB.a \ ab].
\]

dependentProduct1.agda

The “Name”-Example in Agda

Remember:

data Gender : Set where
  female : Gender
  male : Gender

data FemaleName : Set where
  jill : FemaleName
  sara : FemaleName

data MaleName : Set where
  tom : MaleName
  jim : MaleName

data AB : Set where
  prod : (a' : A) → B[a := a'] → AB

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

\[
\text{prod } a' \ b' : AB .
\]

Note that \( b' : B[a := a'] \), so the type of \( b' \) depends on \( a' \).
The “Name”-Example in Agda

```
data MaleName : Set where
tom : MaleName
jim : MaleName

data FemaleName : Set where
jill : FemaleName
sara : FemaleName

Name : Gender → Set
Name male = MaleName
Name female = FemaleName
```

### Now we define

```
record NameWithGender : Set where
  field
gender : Gender
name   : Name gender
```

See `exampleAllNames.agda`.

---

### Existential Quantification

- \( \exists x^A . B \) is true iff there exists an \( a : A \) such that \( B[x := a] \) is true.
- Therefore a proof of \( \exists x^A . B \) is a **pair** \( \langle a, p \rangle \) **consisting of an element** \( a : A \) **and a proof** \( p \) **of** \( B[x := a] \).
- Therefore the set of proofs of \( \exists x^A . B \) is the **dependent product** \( (x : A) \times B \).
- We can **identify** \( \exists x^A . B \) with \( (x : A) \times B \).
\exists \text{ in Agda}

\[ \exists x^A. B \] is represented therefore in Agda by one of the two dependent products in Agda:

```agda
class Version1 : Set where
  field
  \( a \) : A
  \( b \) : \( B[x := a] \)

data Version2 : Set where
  exists : \( (a : A) \to B[x := a] \to Version2 \)
```

Here \( B[x := a] \) is the result of substituting in \( B \) for \( x \) the variable \( a \).

\exists \text{ in Agda}

A generic version, depending on \( A : \text{Set} \) and \( B : A \to \text{Set} \) can be defined as follows (The symbol \( \exists \) can be obtained by typing in “\text{\textbackslash exists}”):

```agda
class \( \exists \text{\textbackslash r} \) \( (A : \text{Set}) \) \( (B : A \to \text{Set}) \) : \text{Set} where
  field
  \( a \) : A
  \( b \) : \( B a \)

data \( \exists \text{\textbackslash d} \) \( (A : \text{Set}) \) \( (B : A \to \text{Set}) \) : \text{Set where}
  exists : \( (a : A) \to B a \to \exists \text{\textbackslash d} A B \)
```

Example (\( \exists \))

As an example,

- we define negation \( \neg \text{Bool} \) on \( \text{Bool} \),
- define an equality \( == \) on \( \text{Bool} \),
- and show \( \forall a^{\text{Bool}}. \exists b^{\text{Bool}}. a == \neg \text{Bool} b. \)
- See `exampleproofprologic11.agda`.

Example (\( \exists \), Cont.)

\( \neg \text{Bool} \) is defined as follows:

\[ \neg \text{Bool} : \text{Bool} \to \text{Bool} \]

\[ \neg \text{Bool} \; \texttt{tt} = \text{ff} \]

\[ \neg \text{Bool} \; \text{ff} = \text{tt} \]
Example (∃)

A Boolean valued equality on \(\text{Bool}\) is defined as follows:

\[
\text{_==_} : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}
\]

\[
\text{tt} \, ==\, \text{Bool} \, b \, = \, b
\]

\[
\text{ff} \, ==\, \text{Bool} \, b \, = \, \neg \text{Bool} \, b
\]

This corresponds to the following truth table:

<table>
<thead>
<tr>
<th>==Bool</th>
<th>ff</th>
<th>tt</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
<td>tt</td>
<td>ff</td>
</tr>
<tr>
<td>tt</td>
<td>ff</td>
<td>tt</td>
</tr>
</tbody>
</table>

Example (∃, Cont.)

Then we define

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

\[
b \, ==\, b' = \text{Atom} \, (b \, ==\, \text{Bool} \, b')
\]

Example (∃, Cont.)

In order to introduce the statement mentioned above, we introduce first the formula \(\exists b^{\text{Bool}}. a == \neg \text{Bool} \, b\) depending on \(a : \text{Bool}\):

\[
\text{record Lemma5aux (a : \text{Bool}) : Set where}
\]

\[
\text{field}
\]

\[
b : \text{Bool}
\]

\[
ab : a == \neg \text{Bool} \, b
\]

The statement \(\forall a^{\text{Bool}}. \exists b^{\text{Bool}}. a == \neg \text{Bool} \, b\) is now as follows:

\[
\text{Lemma5} : \text{Set}
\]

\[
\text{Lemma5} = (a : \text{Bool}) \rightarrow \text{Lemma5aux} \, a
\]

Example (∃, Cont.)

A proof of Lemma5 is an element

\[
\text{lemma5} : \text{Lemma5}
\]

and we get the goal

\[
\text{lemma5} : \text{Lemma5}
\]

\[
\text{lemma5} = \{! !\}
\]

The type of goal is

\[
\text{Lemma5} = (a : \text{Bool}) \rightarrow \text{Lemma5aux} \, a
\]

This goal is solved by applying \text{lemma5} to \(a : \text{Bool}\).
Example (3, Cont.)

Lemma5 : Set
Lemma5 = (a : Bool) → Lemma5aux a

We get

lemma5 : Lemma5
lemma5 ff = {! !}
lemma5 tt = {! !}

The type of the goal is (in pseudo Agda syntax)

Lemma5aux a = record { b : Bool; ab : a == ¬Bool b }

Example (3, Cont.)

We cannot show this goal universally for all a directly.
 We have to provide a different b depending on whether a = tt or a = ff.
 So we introduce pattern matching on whether a = tt or a = ff.

In case of a = ff, the type of goal is

Lemma5aux ff = record { b : Bool; ab : ff == ¬Bool b }

This goal can be solved as follows

lemma5 ff = record { b = tt; ab = true }
(Note that (ff == ¬Bool tt) = T, so true : (ff == ¬Bool tt)).
Example (∃, Cont.)

lemma5 : Lemma5
lemma5 ff = record {b = tt; ab = true}
lemma5 tt = {! !}

The second goal can be solved as follows

lemma5 tt = record {b = ff; ab = true}

So we get the complete proof:

lemma5 : Lemma5
lemma5 ff = record {b = tt; ab = true}
lemma5 tt = record {b = ff; ab = true}

Complex Example

We assume $A, B : \text{Set}$ and equality relations on $A, B$:

postulate A : Set
postulate _==A_ : A → A → Set
postulate B : Set
postulate _==B_ : B → B → Set

We will introduce

- the product $AB$ of $A$ and $B$
- an equality $==AB$ on $AB$
- and show that if $==A$ and $==B$ are symmetric, so is $==AB$.

See exampleProductEqual.agda.

Equality Sets

- $==A$ (and $==B$) could be decidable equalities,
  - i.e. $==A = \lambda(a, b : A) \rightarrow \text{Atom} (\text{eqboolA} a b)$, where $\text{eqboolA} : A \rightarrow A \rightarrow \text{Bool}$,

- Or an undecidable equality.
  - E.g. the equality on $\mathbb{N} \rightarrow \mathbb{N}$ is in standard logic

\[
f = g : \iff \forall n : \mathbb{N}. f(n) = g(n)
\]

which reads in Agda as follows:

\[
_==\mathbb{N}→_ : (f \ g : \mathbb{N} \rightarrow \mathbb{N}) \rightarrow \text{Set}
\]

\[
f ==\mathbb{N}→ g = (n : \mathbb{N}) \rightarrow f n == g n
\]

where $==$ is the equality on $\mathbb{N}$.

Undecidable Equalities

The last equality is undecidable, since in order to check whether $f ==\mathbb{N}→ g$ holds we have to check for all $n : \mathbb{N}$ whether $f n = g n$ holds.
Complex Example (Cont.)

The formation of \( AB = A \times B \) is straightforward:

\[
data \_ \times \_ (A \ B : \text{Set}) : \text{Set} \where
p : A \rightarrow B \rightarrow A \times B
\]

\( AB : \text{Set} \)

\( AB = A \times B \)

Complex Example (Cont.)

We introduce the formulae expressing that an equality on a set is symmetric.

We define this generically depending on an arbitrary set \( A \) and an arbitrary equality \( ==_\_ \) on \( A \).

It is the formula

\[
\forall a, a' : A. a == a' \rightarrow a' == a
\]

The Agda code is as follows:

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

\[
\text{Sym } A \_==\_ = (a \ a' : A) \rightarrow a == a' \rightarrow a' == a
\]

Complex Example (Cont.)

We define the equality \( ==AB \) on \( A \times B \) as follows:

Assume \( ab, ab' : A \times B \).

\( ab \) and \( ab' \) are equal, if there first projections are equal w.r.t. \( ==A \) and their second projections are equal w.r.t. \( ==B \).

So we get

\[
_==AB_ : AB \rightarrow AB \rightarrow \text{Set}
\]

\[
(p \ a \ b) ==AB (p \ a' \ b') = (a ==A a') \land (b ==B b')
\]

Specialisation of \( \text{Sym} \)

We create instances of \( \text{Sym} \) for symmetry on \( A, B, AB \):

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

\[
\text{SymA} = \text{Sym } A \_==\_A_
\]

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

\[
\text{SymB} = \text{Sym } B \_==\_B_
\]

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

\[
\text{SymAB} = \text{Sym } AB \_==\_AB_
\]
**Formulæ vs. Proofs**

- Note that SymA is the **statement** expressing that $==A$ is symmetric.
- It is not a proof that $==A$ is symmetric.
- We can define SymA independently of whether $==A$ is symmetric or not.
- A proof that $==A$ is symmetric is an **element of** SymA, i.e a term symA s.t.

\[ \text{symA : SymA} \]

- Note that we don’t have to show that SymA holds.
- We have to show that if SymA and SymB hold, then SymAB holds as well.

**Complex Example**

- What we want to show is that SymA and SymB implies SymAB.
- So we need to solve

\[ \text{symAB : SymA} \rightarrow \text{SymB} \rightarrow \text{SymAB} \]

\[ \text{symAB} = \{! !\} \]

- We apply symAB to elements symA : SymA, symB : SymB and obtain

\[ \text{symAB : SymA} \rightarrow \text{SymB} \rightarrow \text{SymAB} \]

\[ \text{symAB} \text{ symA symB} = \{! !\} \]

**Complex Example**

- The type of the goal is SymAB which is

\[ (ab \ ab' : \text{AB}) \rightarrow ab \ ==\AB \ ab' \rightarrow ab' \ ==\AB \ ab \]

- In order to solve the goal we apply symAB symA symB to $ab$, $ab'$ and $abab'$ : $ab \ ==\AB \ ab'$. We obtain

\[ \text{symAB : SymA} \rightarrow \text{SymB} \rightarrow \text{SymAB} \]

\[ \text{symAB} \text{ symA symB ab ab' abab'} = \{! !\} \]

**Complex Example**

- The type of the goal is now $ab' \ ==\AB \ ab$.
- $ab' \ ==\AB \ ab$ is defined by pattern matching on $ab$ and $ab'$. In order to show it we use the same pattern matching:

\[ \text{symAB : SymA} \rightarrow \text{SymB} \rightarrow \text{SymAB} \]

\[ \text{symAB} \text{ symA symB (p a b) (p a' b') abab'} = \{! !\} \]
Complex Example

\[ \text{symAB} : \text{SymA} \rightarrow \text{SymB} \rightarrow \text{SymAB} \]
\[ \text{symAB symA symB} (p \ a \ b) (p \ a' \ b') (\text{and} \ aa' \ bb') = \{! !\} \]

- \[ \text{abab'} : a ==A \ a' \wedge b ==B \ b'. \]

  In order to obtain the two components \[ \text{aa'} : a ==A \ a' \]
  and \[ \text{bb'} : b ==B \ b', \] we apply pattern matching to \text{abab'} as well.
- We obtain

\[ \text{symAB} : \text{SymA} \rightarrow \text{SymB} \rightarrow \text{SymAB} \]
\[ \text{symAB symA symB} (p \ a \ b) (p \ a' \ b') (\text{and} \ aa' \ bb') = \{! !\} \]

Complex Example

\[ \text{symAB} : \text{SymA} \rightarrow \text{SymB} \rightarrow \text{SymAB} \]
\[ \text{symAB symA symB} (p \ a \ b) (p \ a' \ b') (\text{and} \ aa' \ bb') = \{! !\} \]

- The type of the first goal is \[ a' ==A \ a. \]
- We have \[ aa' : a ==A \ a' \] and
\[ \text{symA} : (a a' : A) \rightarrow a ==A \ a' \rightarrow a' ==A \ a. \]
- So

\[ \text{symA a a'} : a' ==A \ a \]

and this term can be used in order to solve the first goal:

\[ \text{symAB} : \text{SymA} \rightarrow \text{SymB} \rightarrow \text{SymAB} \]
\[ \text{symAB symA symB} (p \ a \ b) (p \ a' \ b') (\text{and} \ aa' \ bb') = \{! !\} \]

Jump over next 2 sections: Derivations vs. Agda Code and Presuppositions
(g) Derivations vs. Agda Code

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

Example 1

We want to derive in Agda

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

(See example file exampleIdentity.agda)

Step 1:

We need to introduce the type \( A \) first.

Since we want to to have the definition for an arbitrary type \( A \), we postulate (i.e. assume) one type \( A \):

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

Example 1 (Cont.)

Step 2: We state our goal:

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

Step 3:

We want to derive an element of function type \( A \rightarrow A \).

Elements of the function type \( A \rightarrow A \) are introduced by using \( \lambda \)-terms.

If introduced as a \( \lambda \)-term, the term in question will be of the form \( \lambda (a : A) \rightarrow \text{something} \).

So we insert into the goal \( \lambda (a : A) \rightarrow \{! !\} \), use agda-give and obtain

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

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

Step 4:

- In order for $\lambda(a : A) \to \{! !\}$ to be of type $A \to A$, $\{! !\}$ 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 $A \to A$.
- So the type of the goal is $A$.
- This can be inspected by using the goal menu
  Goal type
  which shows the type of the current goal.
  - Has to be executed while the cursor is inside one goal.
  - It shows $A$.

Step 4 (Cont.)

We can inspect the context.

The context contains as only element $a : A$.
- Since we are defining a an element of type $A$ depending on $a : A$, we can use $a$.

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 Refine
- We obtain:

$$f : A \to A = \lambda(a : A) \to a$$

and are done.

derivationsagdacode1.agda

Example 1 (Cont.)

Step 4 (Cont.)

- In Agda step 1 we postulated $A : \text{Set}$. This corresponds to having the global assumption $A : \text{Set}$.
- In Agda step 2 we stated our goal:

$$f : A \to A = \lambda(a : A) \to a$$

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

$$d_0 : A \to A$$
Example 1, Using Rules (Cont.)

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

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

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

\[
\begin{align*}
a & : A \Rightarrow d_1 : A \\
\lambda a^A.d_1 & : A \rightarrow A
\end{align*}
\]

(See example **exampleSampleDerivation2.agda**).

**Step 1:**

- We postulate \( A \):
  
  \[
  \text{postulate } A \text{ : Set}
  \]

- We state our goal:

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

Example 2 (Cont.)

**Step 2:**

- The type of the goal is a function type. We therefore insert into the goal

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

Essentially it allows to derive if \( x : B \) occurs in the context that \( x : B \) holds.
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 $a\rightarrow a\rightarrow a : (A \rightarrow A) \rightarrow A$.
- We can as well use $f$ (for recursive definitions) and $A$ for solving the goal.
- $a\rightarrow a\rightarrow a$ is a function of result type $A$. Applying it to its argument would have as result an element of the type of the goal in question.

Step 3 (Cont):
- Therefore we type into the goal $a\rightarrow a\rightarrow a$ and use goal command **Refine**.
- Agda will then apply $a\rightarrow a\rightarrow a$ to as many goals as needed in order to obtain an element of the desired type.
- In our case it is one (of type $A \rightarrow A$).
- We type

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

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

Example 2 (Cont.)

Step 4:
- The type of the new goal is $A \rightarrow A$.
- This is since $a\rightarrow a\rightarrow a : (A \rightarrow A) \rightarrow A$ needs to be applied to an element of type $A \rightarrow A$ in order to obtain an element of type $A$.
- An element of type $A \rightarrow A$ can be introduced by a $\lambda$-expression $\lambda(a : A) \rightarrow \{! !\}$.
- We type this into the goal and use **Refine** and obtain:

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

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

Example 2 (Cont.)

Step 5:
- The new goal has type $A$.
- The complete expression $\lambda(a : A) \rightarrow \{! !\}$ should have type $A \rightarrow A$, so $\{! !\}$ must have type $A$.
- The context contains $a\rightarrow a\rightarrow a$ and $a$; we can use as well $f$, $A$.
- Both $a\rightarrow a\rightarrow a$ and $a$ have the correct result type $A$.
- 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 Refine we obtain the following and are done.

\[
f : ((A \to A) \to A) \to A
g = \lambda(a-a-a : (A \to A) \to A) \to a-a-a \ (\lambda(a : A) \to a)
\]

Example 2, Using Rules

Postulating $A : \text{Set}$ corresponds to that we make a global assumption $A : \text{Set}$.

Stating the goal means that we have as last line of the derivation:
\[
d_0 : ((A \to A) \to A) \to A
\]

We will in the following use $aaa$ instead of $a-a-a$ in order to save space in derivations.

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

This corresponds to replacing $d_0$ by $\lambda(aaa : (A \to A) \to A).d_1$ and having as last step an introduction rule:
\[
\frac{aaa : (A \to A) \to A \Rightarrow d_1 : A}{\lambda aaa ((A \to A) \to A).d_1 : ((A \to A) \to A) \to A} \quad (\to\text{-I})
\]

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 $\lambda(a : A) \to \{ \text{!} \}$. This corresponds to replacing $d_2$ by $\lambda a^A, d_3$ which can be introduced by an introduction rule:

  $\frac{\text{aaa}(A \to A) \to A, a : A \Rightarrow d_3 : A}{\text{aaa}(A \to A) \Rightarrow \text{aaa}(A \to A) \Rightarrow A} \quad (\to \text{-I})$

  $\frac{\text{aaa}(A \to A) \Rightarrow A \Rightarrow \lambda a^A, d_3 : A \Rightarrow A}{\lambda a^A, d_3 : A \Rightarrow (\text{aaa}(A \to A) \Rightarrow A)} \quad (\to \text{-El})$

Finally we used refine with $a$, which replaced the goal by $a$.
This corresponds to replacing $d_3$ by $a$.

Example 3

- We derive an element of type $A \to B \to A \times B$

  (See exampleProductIntro.agda).

Example 3 (Cont.)

- **Step 1:**
  - We postulate types $A$, $B$:
    postulate $A : \text{Set}$
    postulate $B : \text{Set}$
  - We introduce the product type:
    record _×_ ($A : \text{Set}$) : $A \times B$
    field
    first : $A$
    second : $B$
Example 3 (Cont.)

Step 2:
Our goal is:

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

Step 3:
An element of \( A \to B \to A \times B \) will be of the form

\[ \lambda(a : A) \to \lambda(b : B) \to \{! !\} \]

We insert this into our goal and use \textbf{Refine} and obtain

\[ f : A \to B \to A \times B \]
\[ f = \lambda(a : A) \to \lambda(b : B) \to \{! !\} \]

Step 4:
The new goal is of type \( A \times B \) which is a record type. An element of it can be introduced by an introduction rule.

Elements of type \( A \times B \) introduced by the introduction principle will have the form

\[ \text{record } \{ \text{first} = \{! !\}; \text{second} = \{! !\} \} \]
Example 3 (Cont.)

**Step 5:**
- The first goal has as context:
  - \( a : A , \)
  - \( b : B \)
- We could use as well
  - \( A , B : \text{Set}, \)
  - \( A \times B : \text{Set}, \)
  - \( f : A \rightarrow B \rightarrow A \times B. \)

**Step 5 (Cont)**
- We insert \( a \), use refine and solve the first goal:
  - \( f : A \rightarrow B \rightarrow A \times B \)
  - \( f = \lambda (a : A) \rightarrow \lambda (b : B) \rightarrow \text{record} \{ \text{first} = a ; \text{second} = b \} \)

Example 3 (Cont.)

**Step 6:**
- Similarly we can solve the second one:
  - \( f : A \rightarrow B \rightarrow A \times B \)
  - \( f = \lambda (a : A) \rightarrow \lambda (b : B) \rightarrow \text{record} \{ \text{first} = a ; \text{second} = b \} \)

**Example 3, Using Rules**

- \( A \times B \) is formed as follows (assuming the global assumptions \( A : \text{Set}, B : \text{Set} \)):
  - \[ \frac{A : \text{Set} \quad B : \text{Set}}{A \times B : \text{Set}} (\times - F) \]
- 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:
  \[ d_0 : A \to B \to (A \times B) \]

- Using $\lambda$-abstraction means that we replace $d_0$ by $\lambda a^A \lambda b^B . d_1$ which is introduced by two introduction rules:

\[
\begin{align*}
  & a : A, b : B \Rightarrow d_1 : A \times B \quad (\to -1) \\
  & \lambda a^A \lambda b^B . d_1 : A \Rightarrow B \Rightarrow (A \times B) \quad (\to -1)
\end{align*}
\]

Example 3, Using Rules (Cont.)

- The use of record is reflected by replacing $d_1$ by $\langle d_2, d_3 \rangle$, which can be introduced by an introduction rule:

\[
\begin{align*}
  & a : A, b : B \Rightarrow d_2 : A \quad a : A, b : B \Rightarrow d_3 : B \quad (\times -1) \\
  & a : A, b : B \Rightarrow \langle d_2, d_3 \rangle : A \times B \quad (\to -1) \\
  & \lambda a^A \lambda b^B . \langle d_2, d_3 \rangle : B \Rightarrow (A \times B) \quad (\to -1)
\end{align*}
\]

Example 3, Using Rules (Cont.)

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

\[
\begin{align*}
  & a : A, b : B \Rightarrow a : A \quad a : A, b : B \Rightarrow b : B \quad (\times -1) \\
  & a : A, b : B \Rightarrow \langle a, b \rangle : A \times B \quad (\to -1) \\
  & a : A \Rightarrow \lambda b^B . \langle a, b \rangle : B \Rightarrow (A \times B) \quad (\to -1)
\end{align*}
\]

- 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

\[ (A \to B \times C) \to A \to B \]

(See exampleProductElim.agda).
Example 4 (Cont.)

**Step 1:**
- We postulate types $A$, $B$, $C$:
  
  postulate $A : \text{Set}$
  postulate $B : \text{Set}$
  postulate $C : \text{Set}$

- The product is introduced as before:
  
  record $\_ \times \_ (A B : \text{Set}) : \text{Set}$ where
  field
  first $: A$
  second $: B$

Example 4 (Cont.)

**Step 2:**
- Our goal is:
  
  $f : (A \to B \times C) \to A \to B$
  $f = \{! !\}$

Example 4 (Cont.)

**Step 3:**
- We insert a $\lambda$-expression into the goal, *refine*, and obtain:
  
  $f : (A \to B \times C) \to A \to B$
  $f = \lambda(a-bc : A \to B \times C) \to \lambda(a : A) \to \{! !\}$

Example 4 (Cont.)

**Step 4:**
- The context has no element with result type $B$.
- However, $a-bc$ has function type with result type $B \times C$, which can be projected to $B$.
- We introduce first an element of type $B \times C$ by a *let*-expression, and then derive from it the desired element of type $B$: 
Example 4 (Cont.)

**Step 4 (Cont.):**
- We insert before the goal a let-expression and obtain:

\[ f : (A \rightarrow B \times C) \rightarrow A \rightarrow B \]
\[ f = \lambda(a - bc : A \rightarrow B \times C) \]
\[ \rightarrow \lambda(a : A) \]
\[ \rightarrow \text{let } bc : B \times C \]
\[ bc = \{! !\} \]
\[ \text{in } \{! !\} \]

Example 4 (Cont.)

**Step 5:**
- For solving the first goal (definition of \(bc\)) we can refine \(a - bc\), which has as result type \(B \times C\).

\[ f : (A \rightarrow B \times C) \rightarrow A \rightarrow B \]
\[ f = \lambda(a - bc : A \rightarrow B \times C) \]
\[ \rightarrow \lambda(a : A) \]
\[ \rightarrow \text{let } bc : B \times C \]
\[ bc = a - bc \{! !\} \]
\[ \text{in } \{! !\} \]

Example 4 (Cont.)

**Step 6:**
- The new goal can be solved by refining it with variable \(a\):

\[ f : (A \rightarrow B \times C) \rightarrow A \rightarrow B \]
\[ f = \lambda(a - bc : A \rightarrow B \times C) \]
\[ \rightarrow \lambda(a : A) \]
\[ \rightarrow \text{let } bc : B \times C \]
\[ bc = a - bc a \]
\[ \text{in } \{! !\} \]

Example 4 (Cont.)

**Step 7:**
- The type of the new goal is \(B\).
- We obtain from \(bc\) an element of this type, by applying the first projection to it.
- This projection is \(_\times_{_\text{first}}\).
- We obtain:

\[ f : (A \rightarrow B \times C) \rightarrow A \rightarrow B \]
\[ f = \lambda(a - bc : A \rightarrow B \times C) \]
\[ \rightarrow \lambda(a : A) \]
\[ \rightarrow \text{let } bc : B \times C \]
\[ bc = a - bc a \]
\[ \text{in } _\times_{_\text{first}} bc \]
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 $(a - bc a)$.

We get

$$f : (A \to B \times C) \to A \to B$$

$$f = \lambda(a - bc : A \to B \times C)$$

$$\rightarrow \lambda(a : A)$$

$$\rightarrow \text{\_\_\_.first } (a - bc a)$$

Example 4, Using Rules

Using rules we make the global assumptions

$A : \text{Set}, B : \text{Set}, C : \text{Set}.$

Then we start with our goal

$$d_0 : (A \to (B \times C)) \to A \to B$$

Example 4, Using Rules (Cont.)

The use of a $\lambda$-expression amounts to replacing $d_0$ by

$$\lambda a - bc : A \to (B \times C). \lambda a : A \to d_1 : A$$

introduced by two applications of an introduction rule:

$$a - bc : A \to (B \times C), a : A \Rightarrow d_1 : A \to B$$

$$\Rightarrow \lambda a - bc : A \to (B \times C). \lambda a : A \to d_1 : (A \to (B \times C)) \rightarrow A \to B$$

In Agda, we then replace the goal corresponding to $d_1$ by

$$\text{\_\_\_.first } (a - bc a).$$

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

This can be introduced by two applications of elimination rules:

\[
\begin{align*}
    a - bc : A \to (B \times C), a : A &\Rightarrow a - bc : A \to (B \times C) \\
    a - bc : A \to (B \times C), a : A &\Rightarrow a - bc : B \times C \\
    \Rightarrow a - bc : A \to (B \times C). a : A &\Rightarrow \pi_0 (a - bc a) : B \\
    \Rightarrow a - bc : A \to (B \times C) &\Rightarrow \lambda a : A \to d_1 : (A \to (B \times C)) \rightarrow A \to B
\end{align*}
\]

The two initial judgements can be introduced by assumption rules.
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 \( x : A, y : B \Rightarrow C : \text{Set} \)

implicit contains the judgements
- \( A : \text{Set} \),
- \( x : A \Rightarrow B : \text{Set} \).

A : Set and \( x : A \Rightarrow B : \text{Set} \) are presuppositions of the judgement

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

and of the judgement

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

The next slide shows the presuppositions of judgements.

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Presuppositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, x : A \Rightarrow \text{Context} )</td>
<td>( \Gamma \Rightarrow A : \text{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>

Furthermore, presuppositions of presuppositions of $\Gamma \Rightarrow \theta$ are as well presuppositions of $\Gamma \Rightarrow \theta$.

Example of Presuppositions

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

$x : A, y : B \Rightarrow a = b : (z : C) \times D$ presupposes:

- $\emptyset \Rightarrow \text{Context}$,
- $A : \text{Set}$,
- $x : A \Rightarrow \text{Context}$,
- $x : A \Rightarrow B : \text{Set}$,
- $x : A, y : B \Rightarrow \text{Context}$,
- $x : A, y : B \Rightarrow C : \text{Set}$,
- $x : A, y : B, z : C \Rightarrow \text{Context}$,
- $x : A, y : B, z : C \Rightarrow D : \text{Set}$,
- $x : A, y : B \Rightarrow (z : C) \times D : \text{Set}$,
- $x : A, y : B \Rightarrow a : (z : C) \times D$,
- $x : A, y : B \Rightarrow b : (z : C) \times D$. 

CS_336/CS_M36 Interactive Theorem Proving, Lent Term 2008, Sec. 5 (h)
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:
- $\emptyset \Rightarrow \text{Context}$,
- $A : \text{Set}$,
- $x : A \Rightarrow \text{Context}$,
- $x : A \Rightarrow B : \text{Set}$.

(i) The Full Logical Framework

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

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

Set

Adding $\text{Set} : \text{Set}$ 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
Set (Cont.)

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.

We add rules asserting that if $A : \text{Set}$ then $A : \text{Type}$.

Further we add rules asserting that Type is closed under the dependent function type and product.

Since $\text{Set} : \text{Type}$ we get therefore (by closure under the function type)

$$\text{Set} \rightarrow \text{Set} \rightarrow \text{Type}$$

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

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

(The definition of $\text{prod}$ will be given later.)

Set (Cont.)

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

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

since $\text{Set} : \text{Set}$ does not hold.
Rules for Set (as an El. of Type)

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 \text{Type} is closed under the dependent function type and the dependent product:

Closure of Type under the dependent product

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

Closure of Type under the non-dependent function type

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

Equality Versions of the Rules

Formation Rule for Set

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

Every Set is a Type

\[
\frac{A = B : \text{Set}}{A = B : \text{Type}} \quad (\text{Set2Type})
\]

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

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

Closure of Type under the non-dependent function type

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

Closure of Type under the dependent product

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

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

(x \text{-} \text{F} = , \text{Type})

Closure of Type under the dependent function type

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

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

(\rightarrow \text{-} \text{F} = , \text{Type})

Similarly for the non-dependent versions of the above.

Definition of prod

- Now \( \text{Set} \rightarrow \text{Set} \rightarrow \text{Set} : \text{Type} \).
- And we can derive
  
  \[ \text{prod} \ := \ \lambda (X, Y : \text{Set}) \times X \times Y \]
  
  \[ : \ \text{Set} \rightarrow \text{Set} \rightarrow \text{Set} \]

- We jump over the details. Jump over the details.

Example: prod

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

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

\[ X : \text{Set} \Rightarrow \text{Context} \]

(\text{SetIsType})

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

(\text{Context}_1)

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

(\text{Ass})

Similarly we derive \( X : \text{Set}, Y : \text{Set} \Rightarrow Y : \text{Set} : \)
Example: prod (Cont.)

Now we can derive our desired judgement:

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

and define

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

Set vs. Type in Agda

- In Agda Type will be written as \(\text{Set1}\).
- Set can be written as well as \(\text{Set0}\).
- In Agda, we don’t have that if \(A : \text{Set}\) then \(A : \text{Set1}\).
  - Idea is that from \(A\) we can derive an (up to \(\beta\)-reduction) unique \(B\) s.t. \(A : B\)
- However we have in Agda.
  - Assume \(A : \text{Set}\) or \(A : \text{Set1}\).
  - Assume \(x : A \Rightarrow B : \text{Set}\) or \(x : A \Rightarrow B : \text{Set1}\).
  - Assume that we have at least one of \(A : \text{Set}\) or \(x : A \Rightarrow B : \text{Set}\) or \(x : A \Rightarrow B : \text{Set1}\).
  - Then \((x : A) \Rightarrow B\) or \((x : A) \times B : \text{Set1}\).
  - So \((x : A) \Rightarrow B\) and \((x : A) \times B\) belongs to the maximum type level of \(A\) and \(B\).

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 Kind above Type, s.t.
  \[
  \text{Type} : \text{Kind}.
  \]
- Then
  \[
  \text{Type} \rightarrow \text{Type} \rightarrow \text{Type} : \text{Kind}.
  \]
- In Agda Kind is written as \(\text{Set2}\).

Hierarchy of Types (Set, Type, Kind)
Rules for Type as a Kind

Type is a Kind

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

Every Type is a Kind

\[
A : \text{Type} \quad (\text{Type2Kind})
\]

\[
A : \text{Kind}
\]

Closure of Kind

Closure of Kind under the dependent product

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

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

Closure of Kind under the dependent function type

\[
A : \text{Kind} \quad x : A \Rightarrow B : \text{Kind} \quad (\to - \text{Kind})
\]

\[
(x : A) \to B : \text{Kind}
\]

Plus equality versions of the above rules.

Jump over Context Rule.

Context Rules

Again, the context rules have to be expanded:

\[
\frac{\Gamma \Rightarrow A : \text{Kind}}{\Gamma, x : A \Rightarrow \text{Context}} (\text{Context1Kind})
\]

Definition of \(\text{prod}'\)

Now we can define

\[
\text{prod}' : \lambda (X, Y : \text{Type}). X \times Y
\]

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

This can be iterated further, forming:

\[ \text{Type} = \text{Type}_1, \text{Kind} = \text{Type}_2, \text{Type}_3, \text{Type}_4 \cdots \]

So we have:

- Set : Type,
- Set : Type$_2$, Type = Type$_1$ : Type$_2$,
- Set : Type$_3$, Type = Type$_1$ : Type$_3$, Type$_2$ : Type$_3$,
- Set : Type$_4$, Type = Type$_1$ : Type$_4$, Type$_2$ : Type$_4$, Type$_3$ : Type$_4$,
- etc.

Changes To Presuppositions

If we have the two type levels Set and 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>
<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>
If we have more levels (Kind or Set), then the presuppositions have to be changed again.

E.g., if we have levels Set, Type, 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 $\Gamma \Rightarrow \text{Context}$. 