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

(a) Judgements

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

Judgements

- Therefore we have not only the judgement as in the λ-calculus

$$a : A$$

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

$$A : \mathrm{Set}$$

- Before deriving $a : A$, we first have to show $A : \mathrm{Set}$.
  - So any derivation of $a : A$ contains implicitly a derivation of $A : \mathrm{Set}$.

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 : Set
postulate a : A
postulate P : $A \rightarrow$ Set

$$\begin{array}{lcl} g & : & A \rightarrow A \\ g\ a & = & a \end{array}$$

$$\begin{array}{lcl} a' & : & A \\ a' & = & g\ a \\ p & : & \mathrm{P}\ a \rightarrow \mathrm{P}\ a' \\ p\ x & = & \{!\ \ !\} \end{array}$$

exampleSimpleEquality2.agda

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

Equality Judgements

- When using the simply typed λ-calculus, we could separate the derivation of λ-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 $\pi_0(\langle a, b \rangle) \longrightarrow a$ reads in this style as follows:

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

- $=$ is not directed, so we have as well the rule

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

- We can therefore derive:

$$\cfrac{\cfrac{a : A \qquad b : B}{\pi_0(\langle a, b \rangle) = a : A} \, (\times\text{-Eq}_0)}{a = \pi_0(\langle a, b \rangle) : A} \, (\text{Sym}_{\text{Elem}})$$

Equality of Types

- We will have as well equality between types, written as

$$A = B : \mathrm{Set}$$

- 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 $f : A \rightarrow \mathrm{Set}$.
  If $a = a' : A$, then

$$f\ a = f\ a' : \mathrm{Set}\quad .$$

  - 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)).

$$\frac{x : f\ a \qquad \dfrac{f : A \to A \qquad a = a' : A}{f\ a = f\ a' : \mathrm{Set}}}{x : f\ a'}$$

Examples (Equality of Types)

- Above we have defined $\mathrm{o2} = \mathrm{o} \to \mathrm{o}$.
  As a judgement this reads:

$$\mathrm{o2} = \mathrm{o} \to \mathrm{o} : \mathrm{Set} \quad .$$

Four Judgements

So we have the following **4 types of judgements**:

$A : \mathrm{Set}$	"A is a type".
$A = B : \mathrm{Set}$	"A and B are equal types".
$a : A$	"a is of type A".
$a = b : A$	"a and b are equal elements of type A".

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

Dependent Judgements

- As for the simply typed λ-calculus, in dependent type theory, judgements might depend on a **context**.
- So we obtain judgements of the form

$$\begin{array}{lcl} x_1 : A_1, \ldots, x_n : A_n & \Rightarrow & A : \mathrm{Set} \\ x_1 : A_1, \ldots, x_n : A_n & \Rightarrow & A = B : \mathrm{Set} \\ x_1 : A_1, \ldots, x_n : A_n & \Rightarrow & a : A \\ x_1 : A_1, \ldots, x_n : A_n & \Rightarrow & a = b : A \end{array}$$

Need for Context Judgements

$$x_1 : A_1, \ldots, x_n : A_n \Rightarrow A : \mathrm{Set}$$

$$\ldots$$

- To derive such judgements requires that we know

$$\begin{array}{rcl} & & A_1 : \mathrm{Set} \\ x_1 : A_1 & \Rightarrow & A_2 : \mathrm{Set} \\ x_1 : A_1, x_2 : A_2 & \Rightarrow & A_3 : \mathrm{Set} \\ & \cdots & \\ x_1 : A_1, x_2 : A_2, \ldots, x_{n-1} : A_{n-1} & \Rightarrow & A_n : \mathrm{Set} \end{array}$$

- (Later, when we introduce higher types, this requirement has to be replaced by $A_1 : \mathrm{Type}$, $x_1 : A_1 \Rightarrow A_2 : \mathrm{Type}$ etc.)
  Jump over next slide

Context Judgement

- Note that we didn't require derivations as above in the simply typed λ-calculus, since it was easy to verify whether something is a valid type.
- In case of dependent types $A : \mathrm{Set}$ requires a derivation.
- It can be as complicated to derive $A : \mathrm{Set}$ 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.

  $a : A$ holds iff $B : \mathrm{Set}$ holds.

Context Judgement

- In order to organise this in a better way we introduce an additional judgement $\Gamma \Rightarrow \mathrm{Context}$ for "Γ is a valid context".
- That $x_1 : A_1, \ldots, x_n : A_n \Rightarrow \mathrm{Context}$ holds means exactly what we had above, i.e.:

$$
\begin{array}{rcl}
 & & A_1 : \mathrm{Set} \\
x_1 : A_1 & \Rightarrow & A_2 : \mathrm{Set} \\
x_1 : A_1, x_2 : A_2 & \Rightarrow & A_3 : \mathrm{Set} \\
 & \cdots & \\
x_1 : A_1, x_2 : A_2, \ldots, x_{n-1} : A_{n-1} & \Rightarrow & A_n : \mathrm{Set}
\end{array}
$$

Five Dependent Judgements

- We have therefore 5 dependent judgements:

$$
\begin{array}{lcl}
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow & A : \mathrm{Set} \\
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow & A = B : \mathrm{Set} \\
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow & a : A \\
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow & a = b : A \\
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow & \mathrm{Context}
\end{array}
$$

Example

- The assumption rule, which in case of the simply typed λ-calculus read

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

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

  $$\frac{\Gamma, x : A, \Delta \Rightarrow \mathrm{Context}}{\Gamma, x : A, \Delta \Rightarrow x : A}(\mathrm{Ass})$$

- Similarly we have to deal with the rule introducing constants.

Notations for Judgements, Contexts

- θ (pronounced "theta") will in the following denote an arbitrary non-dep. judgement, i.e. one of the following :
  - $A : \mathrm{Set}$,
  - $A = B : \mathrm{Set}$,
  - $a : A$,
  - $a = b : A$.
- Γ, Δ will usually denote contexts.
- We have the same notations as before, i.e.
  - Γ, Δ is the result of concatenating contexts Γ, Δ,
  - $\Gamma, x : A$ is the result of extending the context Γ by $x : A$,
  - $\emptyset$ is the empty context.
  - We write for $\emptyset \Rightarrow \theta$ usually simply θ.

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

$$\begin{array}{lcl} f & : & B \to A \\ f\ x & = & \{!\ \ !\} \end{array}$$

- 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**.

Contexts in Agda

- Jump over the next example.

Example: Derivation of double

(See **exampleDoubleString2.agda**.)

- We derive
$\mathrm{double} := \lambda x^{\mathrm{String}}.\mathrm{concat}\ x\ x : ((x : \mathrm{String}) \to \mathrm{String})$ in Agda, assuming definitions of String and concat.
- We start with

$$\begin{array}{lcl} \mathrm{double} & : & \mathrm{String} \to \mathrm{String} \\ \mathrm{double}\ s & = & \{!\ \ !\} \end{array}$$

- We can insert into the goal concat:

$$\begin{array}{lcl} \mathrm{double} & : & \mathrm{String} \to \mathrm{String} \\ \mathrm{double}\ s & = & \{!\ \mathrm{concat}\ !\} \end{array}$$

Example: Derivation of double

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

$$\begin{array}{lcl} \mathrm{double} & : & \mathrm{String} \rightarrow \mathrm{String} \\ \mathrm{double}\ s & = & \mathrm{concat}\ \{!\ \ !\}\ \{!\ \ !\} \end{array}$$

- We can check now using goal-menu **Goal Type** (or **Goal Type (normalised)**) that the two new goals require both type String.
- We can check using goal-menu **Context (environment)** that the context of both goals contain $x : \mathrm{String}$.

Example: Derivation of double

- We insert x into the first goal and refine:

$$\begin{array}{lcl} \text{double} & : & \text{String} \rightarrow \text{String} \\ \text{double } s & = & \text{concat } x \; \{! \quad !\} \end{array}$$

- Doing the same with the second goal gives:

$$\begin{array}{lcl} \text{double} & : & \text{String} \rightarrow \text{String} \\ \text{double } s & = & \text{concat } x \; x \end{array}$$

- We are done.

double in Type Theory

A derivation of

$$\mathrm{double} := \lambda x^{\mathrm{String}}.\mathrm{double}\ x\ x$$

in Type Theory, assuming global constants

$$\begin{array}{rcl} \mathrm{String} & : & \mathrm{Set} \ , \\ \mathrm{concat} & : & \mathrm{String} \to \mathrm{String} \to \mathrm{String} \ , \end{array}$$

is as follows:
We first derive $x : \mathrm{String} \Rightarrow \mathrm{Context}$:

$$\frac{\emptyset : \mathrm{Context} \qquad \mathrm{String} : \mathrm{Set}}{x : \mathrm{String} \Rightarrow \mathrm{Context}}\,(\mathrm{Context}_1)$$

double in Type Theory

- We derive $x : \mathrm{String} \Rightarrow x : \mathrm{String}$ using the previous derivation:

$$\frac{x : \mathrm{String} \Rightarrow \mathrm{Context}}{x : \mathrm{String} \Rightarrow x : \mathrm{String}}\,\mathrm{Ass}$$

- We derive

$$x : \mathrm{String} \Rightarrow \mathrm{concat} : \mathrm{String} \to \mathrm{String} \to \mathrm{String}$$

using $x : \mathrm{String} \Rightarrow \mathrm{Context}$ as follows:

$$\frac{\mathrm{concat}{:}\mathrm{String}{\to}\mathrm{String}{\to}\mathrm{String} \qquad x{:}\mathrm{String}{\Rightarrow}\mathrm{Context}}{x : \mathrm{String} \Rightarrow \mathrm{concat} : \mathrm{String} \to \mathrm{String} \to \mathrm{String}}\,(\mathrm{Weak})$$

double in Type Theory

- We derive $x : \mathrm{String} \Rightarrow \mathrm{concat}\ x : \mathrm{String} \to \mathrm{String}$ using the previous derivations:

$$\frac{x{:}\mathrm{String}{\Rightarrow}\mathrm{concat}{:}\mathrm{String}{\to}\mathrm{String}{\to}\mathrm{String} \qquad x{:}\mathrm{String}{\Rightarrow}x{:}\mathrm{String}}{x{:}\mathrm{String}{\Rightarrow}\mathrm{concat}\ x{:}\mathrm{String}{\to}\mathrm{String}}\ ({\to}\text{-El})$$

- The remaining derivation using the above derivations is as follows:

$$\frac{\dfrac{x{:}\mathrm{String}{\Rightarrow}\mathrm{concat}\ x{:}\mathrm{String}{\to}\mathrm{String} \qquad x{:}\mathrm{String}{\Rightarrow}x{:}\mathrm{String}}{x{:}\mathrm{String}{\Rightarrow}\mathrm{concat}\ x\ x{:}\mathrm{String}}\ ({\to}\text{-El})}{\mathrm{double}{:=}\lambda x^{\mathrm{String}}.\mathrm{concat}\ x\ x{:}\mathrm{String}{\to}\mathrm{String}}\ ({\to}\text{-I})$$

(b) Basic Form of Rules

Four Kinds of Rules

- For each set or type construction we have usually 4 kinds of rules:
  (1) **Formation Rules.**
  (2) **Introduction Rules.**
  (3) **Elimination Rules.**
  (4) **Equality Rules.**
- 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:

$$C\ a_1\ \cdots\ a_n : \mathrm{Set} \quad .$$

  - where C is a **set-constructor**,
  - $a_1, \ldots, a_n$ are its arguments.
  - $n = 0$ is possible.
- Later, we will introduce higher levels $\mathrm{Type}, \mathrm{Kind}, \ldots$. Then we have formation rules with conclusion $C\ a_1\ \cdots\ a_n : \mathrm{Type}$ (or $: \mathrm{Kind}$, 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

$$\mathrm{List}\ A : \mathrm{Set}$$

for any $A : \mathrm{Set}$, 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 List itself, which would have type

$$\mathrm{List} : \mathrm{Set} \rightarrow \mathrm{Set} \quad .$$

  The problem is that $\mathrm{Set} \rightarrow \mathrm{Set}$ 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

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

  similarly for all other set formation constructors.

Logical Framework

- Agda has the logical framework built in, so in Agda List will be a function $\mathrm{Set} \to \mathrm{Set}$, in Agda notation:

$$\begin{array}{lcl} \mathrm{List} & : & \mathrm{Set} \to \mathrm{Set} \\ \mathrm{List}\ A & = & \{!\ \ !\} \end{array}$$

Example 1: The Set of Lists

$$\frac{A : \mathrm{Set}}{\textbf{List}\ A : \mathrm{Set}}\ (\mathrm{List\text{-}F})$$

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

Ex. 2: The Set of Natural Numbers

- Formation rule for the set of natural numbers:

$$\mathbb{N} : \mathrm{Set} \qquad (\mathbb{N}\text{-F})$$

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

Jump over next example and Agda

Ex. 3: The Non-Dependent Product

- Formation rule for the non-dependent product:

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set}}{A \times B : \mathrm{Set}} \, (\times\text{-F})$$

  - $A \times B$ stands for $(\times)\ A\ B$.
  - The **set-constructor** is $(\times)$.

Formation Rules in Agda

- The formation of a set is usually done by introducing a constant of a certain set.
- **Example 1:**

$$\begin{array}{lcl} \mathrm{List} & : & \mathrm{Set} \longrightarrow \mathrm{Set} \\ \mathrm{List}\ A & = & \{!\ \ !\} \end{array}$$

Example 2: $(\times)$

- Agda syntax for introducing the **non-dependent product**:

$$\begin{array}{lcl} \_ \times \_ & : & \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set} \\ A \times B & = & \{!\ \ !\} \end{array}$$

(2) Introduction Rules

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

$$C\ a_1\ \cdots\ a_n : A$$

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 1a

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

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

has two introduction rules:

$$[\,] : \mathrm{NatList} \qquad (\mathrm{NatList\text{-}I}[\,])$$

$$\frac{n : \mathbb{N} \qquad l : \mathrm{NatList}}{n :: l : \mathrm{NatList}} \ (\mathrm{NatList\text{-}I\_::\_})$$

- The I in the labels $(\mathrm{NatList\text{-}I}[\,])$, $(\mathrm{NatList\text{-}I\_::\_})$ stands for **I**ntroduction rule.
  Jump to Example 2

Introduction Rule, Example 1b

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

$$\frac{A : \mathrm{Set}}{[\,]_A : \mathrm{List}\ A}\ (\text{List-I}[\,])$$

$$\frac{A : \mathrm{Set} \qquad a : A \qquad l : \mathrm{List}\ A}{a ::_A l : \mathrm{List}\ A}\ (\text{List-I}\_::\_)$$

- Note that we need the **premise** $A : \mathrm{Set}$ in order to guarantee that we can form the set $\mathrm{List}\ A$.

Conflicting Constructors

- We shouldn't use the same constructors for different sets. So if we want to use both $\mathrm{NatList}$ and $\mathrm{List}\ A$, we have to choose a notation like natnil instead of $[\,] : \mathrm{NatList}$, similarly for _ $::$ _.
- We will usually ignore this distinction, if it doesn't cause confusion.

Example 2: Natural Numbers.

- The **natural numbers** $\mathbb{N}$ can be considered as being formed from two operations:
  - 0,
  - S where $\mathrm{S}\ n$ stands for $n+1$.
- Using these two operations we can form 0, $\mathrm{S}\ 0 = 1$, $\mathrm{S}\ 1 = 2$, ... and therefore all natural numbers.
  - So the **constructors** of $\mathbb{N}$ are 0 and S.
- The **introduction rules** of $\mathbb{N}$ are:

$$0 : \mathbb{N} \qquad (\mathbb{N}\text{-}\mathrm{I}_0)$$

$$\frac{n : \mathbb{N}}{\mathrm{S}\ n : \mathbb{N}}\ (\mathbb{N}\text{-}\mathrm{I}_{\mathrm{S}})$$

Canonical Elements

- Canonical elements of a set are those introduced by an introduction rule.
- Canonical elements therefore always start with a **constructor**.
- **Examples:**
  - 0, $\mathrm{S}\ (2+3)$ in case of $\mathbb{N}$.
    - Here 2 stands for $\mathrm{S}\ (\mathrm{S}\ 0)$ and 3 for $\mathrm{S}\ (\mathrm{S}\ (\mathrm{S}\ 0))$.
  - $[\,]$, $(1+1) :: (\mathrm{concat}\ (0 :: [\,])\ [\,])$ in case of $\mathrm{NatList}$.

Non-Canonical Elements

- Terms can usually be reduced further
  - Example:

$$2 + 3 = 2 + \mathrm{S}\ 2 \longrightarrow \mathrm{S}\ (2 + 2) \quad .$$

- The underlying reduction system is essentially a term rewriting system combined with the λ-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 $\mathrm{S}\,(2 + 2)$ 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 $\mathbb{N}$.

(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

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

The elimination rules (indicated by label El) are the first and second projections:

$$\frac{c : A \times B}{\pi_0(c) : A} \, (\times\text{-El}_0) \qquad \frac{c : A \times B}{\pi_1(c) : B} \, (\times\text{-El}_1)$$

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

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

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

- Equality rules will express
  - $n + 0 = n$.
  - $n + \mathrm{S}\, m = \mathrm{S}\, (n + m)$.
- The equality rules show that n is only a parameter, we are eliminating the second argument m.
- Proceeding like this would require **one elimination rule for each function** from $\mathbb{N}$ we want to define.
- Instead we will later introduce one **generic elimination rule,** which will allow to **introduce all functions** we expect to be definable, including **all primitive-recursive** ones.

Elimination in Agda

- Elimination for builtin 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 $\mathbb{N}$:

$$
\begin{array}{lllll}
\_+\_ & & & : & \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
n & + & \mathrm{Z} & = & n \\
n & + & \mathrm{S}\ m & = & \mathrm{S}\ (n + m)
\end{array}
$$

(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 $\longrightarrow$, we obtain $n + 0 = n : \mathbb{N}$ instead.
  - The equality rule (indicated by label Eq) expresses this:

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

Equality Rules

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

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

Jump over next examples

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:

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

Example (Equality Rule)

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

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

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

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

- So it is derived by first introducing $\langle a, b \rangle$ and then eliminating it immediately.
- The equality rule explains how to reduce that element (namely to $a : A$).

Example (Equality Rule, Cont)

- The second equality rule for $\times$ is similar:

$$\frac{a : A \qquad b : B}{\pi_1(\langle a, b \rangle) = b : B}\,(\times\text{-Eq}_1)$$

Example 2 (Equality Rule)

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

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

- $n + 0 : \mathbb{N}$ 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:

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

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

Example 3 (Equality Rule)

- The second equality rule for $+$ is a s follows:

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

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

$$\frac{n : \mathbb{N} \qquad \dfrac{m : \mathbb{N}}{\mathrm{S}\ m : \mathbb{N}}\ (\mathbb{N}\text{-}\mathrm{I}_{\mathrm{S}})}{n + \mathrm{S}\ m : \mathbb{N}}\ (\mathbb{N}\text{-}\mathrm{El}_{+})$$

Equality Rules in Agda

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

$$\begin{array}{lllll} \_+\_ & & & : & \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ n & + & \mathrm{Z} & = & n \\ n & + & \mathrm{S}\ m & = & \mathrm{S}\ (n+m) \end{array}$$

- 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
  - $\pi_0(\langle a, b\rangle) \longrightarrow a$.
  - $\pi_1(\langle a, b\rangle) \longrightarrow b$.

Reduction Strategy

- In case of $(+)$, (Red) are the reductions
  - $n + 0 \longrightarrow n$.
  - $n + \mathrm{S}\ m \longrightarrow \mathrm{S}\ (n + m)$.
  - Note that the second argument is the argument which we are "eliminating".

Example of the Reduction Strategy

- Consider for instance the term $(1 + 1) + (1 + 0)$, where $1 = \mathrm{S}\ 0$.
- 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:

$$1 + 0 \longrightarrow 1$$

and obtain

$$(1 + 1) + (1 + 0) \longrightarrow (1 + 1) + 1 \equiv (1 + 1) + \mathrm{S}\ 0$$

Example of the Reduction Strategy

$$(1+1)+(1+0) \longrightarrow (1+1)+1 \equiv (1+1)+\mathrm{S}\ 0$$

- Now the argument we are eliminating in is in canonical form, and we can use the reduction rule $x + \mathrm{S}\ y \longrightarrow \mathrm{S}\ (x+y)$ in order to reduce this term:

$$(1+1)+\mathrm{S}\ 0 \longrightarrow \mathrm{S}\ ((1+1)+0)$$

- 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:

$$\mathrm{S}((1+1)+0) \longrightarrow \mathrm{S}\ (1+1) \equiv \mathrm{S}\ (1+\mathrm{S}\ 0) \longrightarrow \mathrm{S}\ (\mathrm{S}\ 1) = 3$$

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:

$$\frac{A = B : \mathrm{Set}}{\mathrm{List}\ A = \mathrm{List}\ B : \mathrm{Set}}\ (\mathrm{List}\text{-}\mathrm{F}^{=})$$

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

$$\mathbb{N} = \mathbb{N} : \mathrm{Set} \qquad (\mathbb{N}\text{-}\mathrm{F}^{=})$$

Equality Versions of Rules

- Example: Equality version of the introduction rules for List:

$$\frac{A = A' : \mathrm{Set}}{[\,]_A = [\,]_{A'} : \mathrm{List}\ A}\ (\mathrm{List\text{-}I}[\,]^{=})$$

$$\frac{A = A' : \mathrm{Set} \qquad a = a' : A \qquad l = l' : \mathrm{List}\ A}{a ::_A l = a' ::_{A'} l' : \mathrm{List}\ A}\ (\mathrm{List\text{-}I}^{=}_{\_::\_})$$

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

$$\frac{n = n' : \mathbb{N} \qquad m = m' : \mathbb{N}}{n + m = n' + m' : \mathbb{N}}\ (\mathbb{N}\text{-}\mathrm{El}^{=}_{+})$$

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

$$\frac{\Gamma_1 \Rightarrow \theta_1 \qquad \cdots \qquad \Gamma_n \Rightarrow \theta_n}{\Gamma \Rightarrow \theta}$$

we implicitly introduce as well the following rules

$$\frac{\Delta, \Gamma_1 \Rightarrow \theta_1 \qquad \cdots \qquad \Delta, \Gamma_n \Rightarrow \theta_n}{\Delta, \Gamma \Rightarrow \theta}$$

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

Example

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

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set}}{A \times B : \mathrm{Set}} \,(\times\text{-F})$$

  can be weakened as follows:

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

Example (Cont.)

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

$$\dfrac{\dfrac{x : A, y : A \Rightarrow y : A}{x : A \Rightarrow \lambda y^A.y : A \to A}\,(\to\text{-I})}{\lambda x^A.\lambda y^A.y : A \to A \to A}\,(\to\text{-I})$$

- The first rule used is the rule for λ-introduction, weakened by the context $x : A$.
- The second rule used is the rule for λ-introduction without any weakening.

Weakening of Axioms

- If we have an axiom

$$\theta$$

  for any judgement θ
  - e.g. $\theta \equiv N : \mathrm{Set}$ or $\theta \equiv 0 : \mathbb{N}$

  and we want to weaken it by context Γ, we need to make sure that $\Gamma \Rightarrow \mathrm{Context}$ holds.
- So we need in the weakened form one additional premise:

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

Example

- The formation rule for $\mathbb{N}$

$$\mathbb{N} : \mathrm{Set} \qquad (\mathbb{N}\text{-F})$$

will be weakened as follows:

$$\frac{\Gamma \Rightarrow \mathrm{Context}}{\Gamma \Rightarrow \mathbb{N} : \mathrm{Set}} \ (\mathbb{N}\text{-F})$$

(c) Nondep. Funct. Type and Produ

We introduce in the following non-dependent versions of the product and the function set.

The Non-Dependent Product

Formation Rule

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

Introduction Rule

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

Elimination Rules

$$\frac{c : A \times B}{\pi_0(c) : A} \ (\times\text{-El}_0) \qquad \frac{c : A \times B}{\pi_1(c) : B} \ (\times\text{-El}_1)$$

Equality Rules

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

$$\frac{a : A \qquad b : B}{\pi_1(\langle a, b \rangle) = b : B} \ (\times\text{-Eq}_1)$$

The η-Rule

The η-rule does not fit into the above schema:

$$\frac{c : A \times B}{c = \langle \pi_0(c), \pi_1(c) \rangle : A \times B}\,(\times\text{-}\eta)$$

Equality Versions of the $\times$-Rules

Equality Version of the Formation Rule

$$\frac{A = A' : \mathrm{Set} \qquad B = B' : \mathrm{Set}}{A \times B = A' \times B' : \mathrm{Set}} \; (\times\text{-F}^{=})$$

Equality Version of the Introduction Rule

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

Equality Versions of the Elimination Rules

$$\frac{c = c' : A \times B}{\pi_0(c) = \pi_0(c') : A} \; (\times\text{-El}_0^{=}) \qquad \frac{c = c' : A \times B}{\pi_1(c) = \pi_1(c') : B} \; (\times\text{-El}_1^{=})$$

The Non-Dependent Function Type

Formation Rule

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set}}{A \to B : \mathrm{Set}}\,(\to\text{-F})$$

Introduction Rule

$$\frac{x : A \Rightarrow b : B}{(\lambda x : A.b) : A \to B}\,(\to\text{-I})$$

Elimination Rule

$$\frac{f : A \to B \qquad a : A}{f\ a : B}\,(\to\text{-El})$$

Equality Rule

$$\frac{x : A \Rightarrow b : B \qquad a : A}{(\lambda x : A.b)\ a = b[x := a] : B}\,(\to\text{-Eq})$$

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

β-Reduction

- $b[x := a]$ was as for the simply typed λ-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 β-reduction

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

α-Equivalence

- As for the simply typed λ-calculus, terms which differ in the choice of bound variables (i.e. which are α-equivalent) are identified:
  - E.g. $\lambda x^A.x$ and $\lambda y^A.y$ are identified.
  - E.g. $\lambda x^{\mathbb{N}}.x + x$ and $\lambda y^{\mathbb{N}}.y + y$ are identified.
  - A similar rule applies to bound variables **in types** (see later).

The η-Rule

Again the η-rule does not fit into the above schema:

$$\frac{f : A \to B}{f = \lambda x^A.f\ x : A \to B}\,(\to\text{-}\eta)$$

Equality Versions of the $\rightarrow$-Rules

Equality Version of the Formation Rule

$$\frac{A = A' : \mathrm{Set} \qquad B = B' : \mathrm{Set}}{A \rightarrow B = A' \rightarrow B' : \mathrm{Set}} \; (\rightarrow\text{-F}^{=})$$

Equality Version of the Introduction Rule

$$\frac{x : A \Rightarrow b = b' : B}{\lambda x^{A}.b = \lambda x^{A}.b' : A \rightarrow B} \; (\rightarrow\text{-I}^{=})$$

Equality Version of the Elimination Rule

$$\frac{f = f' : A \rightarrow B \qquad a = a' : A}{f\ a = f'\ a' : B} \; (\rightarrow\text{-El}^{=})$$

Jump over subsection on structural rules

(d) Structural Rules

Context Rules

The empty context

$$\emptyset \Rightarrow \mathrm{Context} \qquad (\mathrm{Context}_0)$$

Extending a context

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

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

Example Derivation (Context Rules)

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

$$\mathbb{N} : \mathrm{Set} \qquad (\mathbb{N}\text{-F})$$

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

$$\frac{\Gamma \Rightarrow \mathrm{Context}}{\Gamma \Rightarrow \mathbb{N} : \mathrm{Set}} \; (\mathbb{N}\text{-F})$$

Example Derivation (Context Rules)

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

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathbb{N} : \mathrm{Set}}{x : \mathbb{N} \Rightarrow \mathrm{Context}}\,(\mathrm{Context}_1)}{x : \mathbb{N} \Rightarrow \mathbb{N} : \mathrm{Set}}\,(\mathbb{N}\text{-F})}{x : \mathbb{N}, y : \mathbb{N} \Rightarrow \mathrm{Context}}\,(\mathrm{Context}_1)}{x : \mathbb{N}, y : \mathbb{N} \Rightarrow \mathbb{N} : \mathrm{Set}}\,(\mathbb{N}\text{-F})}{x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow \mathrm{Context}}\,(\mathrm{Context}_1)
$$

Assumption Rule

$$\frac{\Gamma, x : A, \Delta \Rightarrow \mathrm{Context}}{\Gamma, x : A, \Delta \Rightarrow x : A} \, (\mathrm{Ass})$$

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

    $$\Gamma, x : A, \Delta \Rightarrow x : B'$$

    only for the last occurrence of $x : B'$ in Δ.

Example Deriv. (Assumpt. Rule)

- We extend the derivation of

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

above to a derivation of $x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow y : \mathbb{N}$:

$$\frac{x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow \mathrm{Context}}{x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow y : \mathbb{N}}\,(\mathrm{Ass})$$

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

$$\frac{x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow \mathrm{Context}}{x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow z : \mathbb{N}}\,(\mathrm{Ass})$$

Example Deriv. (Assumpt. Rule)

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

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\mathbb{N} : \mathrm{Set}}{x : \mathbb{N} \Rightarrow \mathrm{Context}}\ (\mathrm{Context}_1)}{x : \mathbb{N} \Rightarrow \mathbb{N} : \mathrm{Set}}\ (\mathbb{N}\text{-F})}{x : \mathbb{N}, y : \mathbb{N} \Rightarrow \mathrm{Context}}\ (\mathrm{Context}_1)}{x : \mathbb{N}, y : \mathbb{N} \Rightarrow \mathbb{N} : \mathrm{Set}}\ (\mathbb{N}\text{-F})}{x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow \mathrm{Context}}\ (\mathrm{Context}_1)}{x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \Rightarrow y : \mathbb{N}}\ (\mathrm{Ass})
$$

Assumption Rule in Agda

- When we define a function:

$$\begin{array}{lcl} f & : & A \to B \\ f\ a & = & \{!\ \ !\} \end{array}$$

  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

$$\frac{a : A \Rightarrow \{!\ \ !\} : B}{\lambda(a : A).\{!\ \ !\} : A \to B}\,(\to\text{-I})$$

- If B is equal to A we can use the assumption rule directly

$$\frac{a : A \Rightarrow a : A}{\lambda(a : A).a : A \to A}\,(\to\text{-I})$$

  in order to solve this goal.

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.

$$\cfrac{\cfrac{\cfrac{\cdots}{a : A \Rightarrow a : A}\ (\mathrm{Ass}) \\ \vdots}{a : A \Rightarrow s : B}}{\lambda(a : A).s : A \to B}\ (\to\text{-I})$$

Assumption Rule in Agda (Cont.)

- Similarly, when solving the goal

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

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

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

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

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

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

Weakening Rule

$$\frac{\Gamma, \Gamma' \Rightarrow \theta \qquad \Gamma, \Delta, \Gamma' \Rightarrow \mathrm{Context}}{\Gamma, \Delta, \Gamma' \Rightarrow \theta} \; (\mathrm{Weak})$$

- θ stands for an **arbitrary non-dependent judgement**.
- This rule allows to **add an additional context piece** (Δ) to the **context of a judgement**.
  - The judgement $\Gamma, \Gamma' \Rightarrow \theta$ is **weakened** by Δ.

Weakening Rule (Cont.)

- 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 $A : \mathrm{Set}$ (corresponding to "postulate" in Agda).
  - Then $\Gamma \Rightarrow A : \mathrm{Set}$ doesn't follow without the weakening rule.

Example Deriv. (Weak. Rule)

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

$$\dfrac{\dfrac{\dfrac{A{:}\mathrm{Set}}{a{:}A\Rightarrow\mathrm{Context}}\,(\mathrm{Context}_1)}{a{:}A\Rightarrow a{:}A}\,(\mathrm{Ass}) \qquad \dfrac{\dfrac{B{:}\mathrm{Set} \qquad \dfrac{A{:}\mathrm{Set}}{a{:}A\Rightarrow\mathrm{Context}}\,(\mathrm{Context}_1)}{a{:}A\Rightarrow B{:}\mathrm{Set}}\,(\mathrm{Weak})}{a{:}A,b{:}B\Rightarrow\mathrm{Context}}\,(\mathrm{Context}_1)}{a{:}A,b{:}B\Rightarrow a{:}A}\,(\mathrm{Weak})$$

Example Deriv.2 (Weak. Rule)

- We derive $x : A \to (B \times C), y : A \Rightarrow x : A \to (B \times C)$, under the global assumptions $A : \mathrm{Set}$, $B : \mathrm{Set}$, $C : \mathrm{Set}$:

$$
\dfrac{\dfrac{A : \mathrm{Set} \qquad \dfrac{\dfrac{\dfrac{A : \mathrm{Set} \qquad \dfrac{B : \mathrm{Set} \qquad C : \mathrm{Set}}{B \times C : \mathrm{Set}}\,(\times\text{-F})}{A \to (B \times C) : \mathrm{Set}}\,(\to\text{-F})}{x : A \to (B \times C) \Rightarrow \mathrm{Context}}\,(\mathrm{Context}_1)}{x : A \to (B \times C) \Rightarrow A : \mathrm{Set}}\,(\mathrm{Weak})}{\dfrac{x : A \to (B \times C), y : A \Rightarrow \mathrm{Context}}{x : A \to (B \times C), y : A \Rightarrow x : A \to (B \times C)}\,(\mathrm{Ass})}\,(\mathrm{Context}_1)
$$

General Equality Rules

Reflexivity

$$\frac{A : \mathrm{Set}}{A = A : \mathrm{Set}}\ (\mathrm{Refl}_{\mathrm{Set}})$$

$$\frac{a : A}{a = a : A}\ (\mathrm{Refl}_{\mathrm{Elem}})$$

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

Symmetry

$$\frac{A = B : \mathrm{Set}}{B = A : \mathrm{Set}}\ (\mathrm{Sym}_{\mathrm{Set}})$$

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

General Equality Rules (Cont.)

Transitivity

$$\frac{A = B : \mathrm{Set} \qquad B = C : \mathrm{Set}}{A = C : \mathrm{Set}} \ (\mathrm{Trans}_{\mathrm{Set}})$$

$$\frac{a = b : A \qquad b = c : A}{a = c : A} \ (\mathrm{Trans}_{\mathrm{Elem}})$$

Transfer

$$\frac{a : A \qquad A = B : \mathrm{Set}}{a : B} \ (\mathrm{Transfer}_0)$$

$$\frac{a = b : A \qquad A = B : \mathrm{Set}}{a = b : B} \ (\mathrm{Transfer}_1)$$

Example Deriv. (Gen. Equal. Rules)

$$
\begin{prooftree}
\AxiomC{$\mathbb{N}{:}\mathrm{Set}$}
\RightLabel{$(\mathrm{Context}_1)$}
\UnaryInfC{$y{:}\mathbb{N}\Rightarrow\mathrm{Context}$}
\RightLabel{(Ass)}
\UnaryInfC{$y{:}\mathbb{N}\Rightarrow y{:}\mathbb{N}$}
\RightLabel{$(\mathbb{N}\text{-}\mathrm{Eq}_{+,0})$}
\UnaryInfC{$y{:}\mathbb{N}\Rightarrow y+0=y{:}\mathbb{N}$}
\AxiomC{$\mathbb{N}{:}\mathrm{Set}$}
\RightLabel{$(\mathrm{Context}_1)$}
\UnaryInfC{$y{:}\mathbb{N}\Rightarrow\mathrm{Context}$}
\RightLabel{$(\mathbb{N}\text{-}\mathrm{F})$}
\UnaryInfC{$y{:}\mathbb{N}\Rightarrow\mathbb{N}{:}\mathrm{Set}$}
\RightLabel{$(\mathrm{Context}_1)$}
\UnaryInfC{$y{:}\mathbb{N},x{:}\mathbb{N}\Rightarrow\mathrm{Context}$}
\RightLabel{(Ass)}
\UnaryInfC{$y{:}\mathbb{N},x{:}\mathbb{N}\Rightarrow x{:}\mathbb{N}$}
\AxiomC{$\mathbb{N}{:}\mathrm{Set}$}
\RightLabel{$(\mathrm{Contex}$}
\UnaryInfC{$y{:}\mathbb{N}\Rightarrow\mathrm{Context}$}
\RightLabel{(Ass)}
\UnaryInfC{$y{:}\mathbb{N}\Rightarrow y{:}\mathbb{N}$}
\RightLabel{$(\rightarrow\text{-}\mathrm{Eq})$}
\BinaryInfC{$y{:}\mathbb{N}\Rightarrow(\lambda x^{\mathbb{N}}.x)\ y=y{:}\mathbb{N}$}
\RightLabel{$(\mathrm{Sym}_{\mathrm{Elem}})$}
\UnaryInfC{$y{:}\mathbb{N}\Rightarrow y=(\lambda x^{\mathbb{N}}.x)\ y{:}\mathbb{N}$}
\RightLabel{$(\mathrm{Trans}_{\mathrm{Elem}})$}
\BinaryInfC{$y{:}\mathbb{N}\Rightarrow y+0=(\lambda x^{\mathbb{N}}.x)\ y{:}\mathbb{N}$}
\RightLabel{$(\rightarrow\text{-}\mathrm{I}^{=})$}
\UnaryInfC{$\lambda y^{\mathbb{N}}.y+0=\lambda y^{\mathbb{N}}.(\lambda x^{\mathbb{N}}.x)\ y{:}\mathbb{N}\rightarrow\mathbb{N}$}
\end{prooftree}
$$

Example Deriv. (Gen. Equal. Rules)

- In the previous derivation, the most complicated step was:

$$\frac{y : \mathbb{N}, x : \mathbb{N} \Rightarrow x : \mathbb{N} \qquad y : \mathbb{N} \Rightarrow y : \mathbb{N}}{y : \mathbb{N} \Rightarrow (\lambda x^{\mathbb{N}}.x)\ y = y : \mathbb{N}} (\rightarrow\text{-Eq})$$

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

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

  with $A := B := \mathbb{N}$, $b := x$, $a := y$.
  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

$$\frac{y : \mathbb{N}, x : \mathbb{N} \Rightarrow x : \mathbb{N} \qquad y : \mathbb{N} \Rightarrow y : \mathbb{N}}{y : \mathbb{N} \Rightarrow (\lambda x^{\mathbb{N}}.x)\ y = y : \mathbb{N}} (\rightarrow \text{-Eq})$$

  we can derive using the introduction and elimination rule

$$y : \mathbb{N} \Rightarrow (\lambda x^{\mathbb{N}}.x)\ y : \mathbb{N}$$

  as follows:

$$\frac{\dfrac{y : \mathbb{N}, x : \mathbb{N} \Rightarrow x : \mathbb{N}}{y : \mathbb{N} \Rightarrow \lambda x^{\mathbb{N}}.x : \mathbb{N} \rightarrow \mathbb{N}} (\rightarrow \text{-I}) \qquad y : \mathbb{N} \Rightarrow y : \mathbb{N}}{y : \mathbb{N} \Rightarrow (\lambda x^{\mathbb{N}}.x)\ y : \mathbb{N}} (\rightarrow \text{-El})$$

Example Deriv. (Gen. Equ. Rules)

- The equality rule expresses how the function $\lambda x^{\mathbb{N}}.x$ 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

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

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

Substitution 2

$$\frac{\Gamma, x : A, \Gamma' \Rightarrow B : \mathrm{Set} \qquad \Gamma \Rightarrow a = a' : A}{\Gamma, \Gamma'[x := a] \Rightarrow B[x := a] = B[x := a'] : \mathrm{Set}} \ (\mathrm{Subst}_2)$$

Substitution Rules

Substitution 3

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

Example Deriv. (Substitution)

$$
\dfrac{\dfrac{\dfrac{\cdots}{x{:}\mathbb{N},y{:}\mathbb{N}\Rightarrow x{:}\mathbb{N}}\,(\mathrm{Ass}) \qquad \dfrac{\cdots}{x{:}\mathbb{N},y{:}\mathbb{N}\Rightarrow y{:}\mathbb{N}}\,(\mathrm{Ass})}{x : \mathbb{N}, y : \mathbb{N} \Rightarrow x + y : \mathbb{N}}\,(\mathbb{N}\text{-I}_{+}) \qquad 0 : \mathbb{N}}{\dfrac{y : \mathbb{N} \Rightarrow 0 + y : \mathbb{N}}{\lambda y^{\mathbb{N}}.0 + y : \mathbb{N} \to \mathbb{N}}\,(\to\text{-I})}\,(\mathrm{Subst}_1)
$$

Example Deriv. 2 (Substitution)

$$
\dfrac{
  \dfrac{\cdots}{z{:}\mathbb{N},x{:}\mathbb{N},y{:}\mathbb{N}\Rightarrow x+y{:}\mathbb{N}}
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{
            \dfrac{\mathbb{N}{:}\mathrm{Set}}{z{:}\mathbb{N}\Rightarrow\mathrm{Context}}\;(\mathrm{Context}_1)
          }{z{:}\mathbb{N}\Rightarrow\mathbb{N}{:}\mathrm{Set}}\;(\mathbb{N}\text{-F})
        }{z{:}\mathbb{N},u{:}\mathbb{N}\Rightarrow\mathrm{Context}}\;(\mathrm{Context}_1)
      }{z{:}\mathbb{N},u{:}\mathbb{N}\Rightarrow u{:}\mathbb{N}}\;(\mathrm{Ass})
    }{z{:}\mathbb{N},u{:}\mathbb{N}\Rightarrow \mathrm{S}\ u{:}\mathbb{N}}\;(\mathbb{N}\text{-I}_{\mathrm{S}})
    \qquad
    \dfrac{
      \dfrac{
        \dfrac{\mathbb{N}{:}\mathrm{Set}}{z{:}\mathbb{N}\Rightarrow\mathrm{Context}}\;(\mathrm{Contex}
      }{z{:}\mathbb{N}\Rightarrow z{:}\mathbb{N}}\;(\mathrm{Ass})
    }{z{:}\mathbb{N}\Rightarrow z+0=z{:}\mathbb{N}}\;(\mathbb{N}\text{-Eq}
  }{z{:}\mathbb{N}\Rightarrow \mathrm{S}\ (z+0)=\mathrm{S}\ z{:}\mathbb{N}}\;(\mathrm{Subst}
}{
  \dfrac{
    \dfrac{z{:}\mathbb{N},y{:}\mathbb{N}\Rightarrow(\mathrm{S}\ z+0)+y=\mathrm{S}\ z+y{:}\mathbb{N}}{z{:}\mathbb{N}\Rightarrow\lambda y^{\mathbb{N}}.(\mathrm{S}\ z+0)+y)=\lambda y^{\mathbb{N}}.\mathrm{S}\ z+y{:}\mathbb{N}\to\mathbb{N}}\;(\to\text{-I}^{=})
  }{\lambda z^{\mathbb{N}}.\lambda y^{\mathbb{N}}.(\mathrm{S}\ z+0)+y=\lambda z^{\mathbb{N}}.\lambda y^{\mathbb{N}}.\mathrm{S}\ z+y{:}\mathbb{N}\to\mathbb{N}\to\mathbb{N}}\;(\to\text{-I}^{=})
}\;(\mathrm{Subst}_3)
$$

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

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

$$\begin{aligned}\{f \mid\ & f \text{ function} \\ & \wedge \mathrm{dom}(f) = A \\ & \wedge \forall a \in A.f(a) \in B[x := a]\}\end{aligned}$$

Example (Dep. Function Set)

- Let Gender be the set of **genders**, informally written

$$\mathrm{Gender} = \{\mathrm{female}, \mathrm{male}\} \ .$$

- In Agda, Gender would be defined by

$$\begin{array}{l}
\mathrm{data\ Gender : Set\ where} \\
\quad \begin{array}{lcl} \mathrm{female} & : & \mathrm{Gender} \\ \mathrm{male} & : & \mathrm{Gender} \end{array}
\end{array}$$

Example (Dep. Function Set)

- Let for $g : \mathrm{Gender}$ the set

$$\mathrm{Name}\ g$$

  be the collection of **names of that gender**, e.g. informally written
  - $\mathrm{Name\ female} = \{\mathrm{jill}, \mathrm{sara}\}$,
  - $\mathrm{Name\ male} = \{\mathrm{tom}, \mathrm{jim}\}$.

Example (Dep. Function Set)

- More formally, Name can be defined in Agda as follows:

$$
\begin{array}{l}
\text{data MaleName : Set where} \\
\qquad \begin{array}{lcl} \text{tom} & : & \text{MaleName} \\ \text{jim} & : & \text{MaleName} \end{array}
\end{array}
$$

$$
\begin{array}{l}
\text{data FemaleName : Set where} \\
\qquad \begin{array}{lcl} \text{jill} & : & \text{FemaleName} \\ \text{sara} & : & \text{FemaleName} \end{array}
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{Name} : \text{Gender} \to \text{Set}} \\
\text{Name male} & = & \text{MaleName} \\
\text{Name female} & = & \text{FemaleName}
\end{array}
$$

Example (Dep. Function Set)

- Define

$$\begin{aligned}\mathrm{select} &: (g : \mathrm{Gender}) \to \mathrm{Name}\ g \\ \mathrm{select\ female} &= \mathrm{jill} \\ \mathrm{select\ male} &= \mathrm{tom}\end{aligned}$$

- select selects for every gender a name.
- $\mathrm{select\ female}$ will be an element of $\mathrm{Name\ female} = (\mathrm{Name}\ g)[g := \mathrm{female}]$.
- It wouldn't make sense to say $(\mathrm{select\ female}) : \mathrm{Name}\ g$, 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

$$\lambda g^{\mathrm{Gender}}.\mathrm{tom} : (g : \mathrm{Gender}) \to \mathrm{Name}\ g$$

since we **don't** have

$$(\lambda g^{\mathrm{Gender}}.\mathrm{tom})\,\mathrm{female} : \mathrm{Name\ female}$$

Rules of the Dep. Funct. Set

Formation Rule

$$\frac{A : \mathrm{Set} \qquad x : A \Rightarrow B : \mathrm{Set}}{(x : A) \to B : \mathrm{Set}} \; (\to\text{-F})$$

Introduction Rule

$$\frac{x : A \Rightarrow b : B}{\lambda x^{A}.b : (x : A) \to B} \; (\to\text{-I})$$

Rules of the Dep. Funct. Set

Elimination Rule

$$\frac{f : (x : A) \to B \qquad a : A}{f\ a : B[x := a]}\ (\to\text{-El})$$

Equality Rule

$$\frac{x : A \Rightarrow b : B \qquad a : A}{(\lambda x^A.b)\ a = b[x := a] : B[x := a]}\ (\to\text{-Eq})$$

The η-Rule

The η-rule has a special status:

η-Rule

$$\frac{f : (x : A) \to B}{f = \lambda x^A . f\ x : (x : A) \to B}\ (\to\text{-}\eta)$$

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

Equality Versions of the above

Equality Version of the Formation Rule

$$\frac{A = A' : \mathrm{Set} \qquad x : A \Rightarrow B = B' : \mathrm{Set}}{(x : A) \to B = (x : A') \to B' : \mathrm{Set}} (\to \text{-F}^{=})$$

Equality Version of the Introduction Rule

$$\frac{x : A \Rightarrow b = b' : B}{\lambda x^A.b = \lambda x^A.b' : (x : A) \to B} (\to \text{-I}^{=})$$

Equality Version of the Elimination Rule

$$\frac{f = f' : (x : A) \to B \qquad a = a' : A}{f\ a = f'\ a' : B[x := a]} (\to \text{-El}^{=})$$

Non-Dep. Funct. Set as an Abbrev.

- The **non-dependent function set**

$$A \to B$$

can be regarded as an **abbreviation** for the **dependent function set**

$$(x : A) \to B \quad ,$$

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 $(x : A) \to B$ are introduced as before by using
  - either λ-abstraction, i.e. we can define

$$
\begin{array}{lcl}
f & : & (x : A) \to B \\
f & = & \lambda(x : A) \to b
\end{array}
$$

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

$$
\begin{array}{lcl}
f & : & (x : A) \to B \\
f & = & \lambda x \to b
\end{array}
$$

    - 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

- or by writing

$$\begin{array}{rcl} f & : & (x : A) \to B \\ f\ x & = & b \end{array}$$

depfunctionset.agda

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 $f\ a : B[x := a]$.

Abbreviations

- We can write

$$(n \;\; m : \mathbb{N}) \to A$$

instead of

$$(n : \mathbb{N}) \to (m : \mathbb{N}) \to A$$

$(x : A) \to \cdots$ **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 $s : ((x : A) \to B)$.

$(x : A) \to \cdots$ 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.

Predicate Log. in Dep. Type Theo.

- 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

    $$A \longrightarrow B$$

    is represented as the nondependent function set

    $$A \rightarrow B$$

  - Conjunction

    $$A \wedge B$$

    is represented as one of the two versions of the product of A and B.

Predicate Log. in Dep. Type Theo.

- Disjunction will be introduced later (as the disjoint union).
- $\neg A$ has been introduced as $A \to \bot$.
- If $f : A_1 \to \cdots \to A_n \to \mathrm{Bool}$ is a function, we can represent the predicate "$f\ a_1\ \cdots\ a_n$ is true" as

$$\mathrm{Atom}\ (f\ a_1\ \cdots\ a_n)$$

Jump over next slide

Predicate Log. in Dep. Type Theo.

- The definitions of $\neg A$, Atom rely on the rules for $\bot$, $\top$, Bool and Atom.
- They have been only introduced in the λ-calculus (and the rules for Atom have not been introduced at all), but not yet in the context of dependent type theory.
- They will be introduced in detail later.
- In this Subsect. we will deal mainly with the predicate calculus in Agda.
- Therefore an understanding of the rules as they occur in the λ-calculus (or in case of Atom an understanding of how to use it in Agda) suffices.
  - The rules of the typed λ-calculus can easily be translated into type theory.

Predicate Log. in Dep. Type Theo.

- 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
  - $\forall \mathbf{x} : \mathbf{A}.\mathbf{B}$ or $\forall \mathbf{x}^{\mathbf{A}}.\mathbf{B}$ for
    "for all x of type A, B holds"
    (where B usually depends on x);
  - $\exists \mathbf{x} : \mathbf{A}.\mathbf{B}$ or $\exists \mathbf{x}^{\mathbf{A}}.\mathbf{B}$ for
    "there exists an x of type A, s.t. B holds"
    (again B usually depends on x).

Universal Quantification

- $\forall x^A.B$ is true iff, for all $x : A$ there exists a proof of B (with that x).
- Therefore a proof of $\forall x^A.B$ is a **function, which takes an x:A and computes an element of B**.
- Therefore the set of proofs of $\forall x^A.B$ is the set of functions, mapping an element $x : A$ to an element of B.
- This set is just the **dependent function set** $(x : A) \to B$.
- Therefore we can **identify** $\mathbf{\forall x^A.B}$ with $\mathbf{(x : A) \to B}$.

$\forall$ in Agda

- $\forall x^A.B$ is represented by $(x : A) \to B$ in Agda.
  - Remember that $\forall x : A.B$ is another notation for $\forall x^A.B$.
- As an example,
  - we define a $<$-operation on Bool using $\mathrm{ff} < \mathrm{tt}$ is true and $b < b'$ is false, otherwise.
  - Then we show $\forall x^{\mathrm{Bool}}.\neg(x < x)$.
- See **exampleLessBool.agda**.

Example (∀, Cont.)

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

$$\begin{aligned}
_ <_{\mathrm{Bool}} _ &: \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool} \\
\mathrm{ff} <_{\mathrm{Bool}} b &= b \\
\mathrm{tt} <_{\mathrm{Bool}} _ &= \mathrm{ff}
\end{aligned}$$

- This means that $<_{\mathrm{Bool}}$ has the following truth table:

$<_{\mathrm{Bool}}$	ff	tt
ff	ff	tt
tt	ff	ff

Example ($\forall$, Cont.)

- Explanation of this definition:
  - If we identify ff with the number 0, tt with 1, then $b <_{\mathrm{Bool}} b'$ 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 $a <_{\mathrm{Bool}} b$ is the same as b.
    - if a is true, then a is never less than b.

$<$Boollong

$$\_{<}\mathrm{Bool}\_ : \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool}$$
$$\mathrm{ff} \ {<}\mathrm{Bool}\ b \ = b$$
$$\mathrm{tt}\ {<}\mathrm{Bool}\ \_ = \mathrm{ff}$$

- The above defines the same function as the following long version:

$$\begin{aligned}
&_{<}\mathrm{Boollong}_ : \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool}\\
&\mathrm{ff}\ {<}\mathrm{Boollong}\ \mathrm{ff} \ = \mathrm{ff}\\
&\mathrm{ff}\ {<}\mathrm{Boollong}\ \mathrm{tt} = \mathrm{tt}\\
&\mathrm{tt}\ {<}\mathrm{Boollong}\ \mathrm{ff} \ = \mathrm{ff}\\
&\mathrm{tt}\ {<}\mathrm{Boollong}\ \mathrm{tt} = \mathrm{ff}
\end{aligned}$$

<Boollong

- Proving properties for <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.

Example ($\forall$, Cont.)

- Now we define $<$ as follows

$$\_<\_ : \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Set}$$
$$b \ < \ b' = \mathrm{Atom}\ (b \ <\!\mathrm{Bool} \ b')$$

Example ($\forall$, Cont.)

- We introduce $\neg$:

$$\neg : \mathrm{Set} \to \mathrm{Set}$$
$$\neg A = A \to \bot$$

- The statement that $<$ is antireflexive is

$$\forall a^{\mathrm{Bool}}.\neg(a \ < \ a)$$

which is represented in Agda as follows:

$$\begin{array}{lcl} \mathrm{Lemma4} & : & \mathrm{Set} \\ & = & (a : \mathrm{Bool}) \to \neg \ (a \ < \ a) \end{array}$$

Example (∀, Cont.)

$$\begin{aligned}\mathrm{Lemma4} &: \mathrm{Set} \\ &= (a : \mathrm{Bool}) \rightarrow \neg\,(a < a)\end{aligned}$$

- Since $\neg\,(a < a) = (a < a) \rightarrow \perp$, we have

$$\begin{aligned}\mathrm{Lemma4} &= (a : \mathrm{Bool}) \rightarrow \neg\,(a < a) \\ &= (a : \mathrm{Bool}) \rightarrow (a < a) \rightarrow \perp\end{aligned}$$

Example ($\forall$, Cont.)

$$\mathrm{Lemma4} = (a : \mathrm{Bool}) \to (a < a) \to \bot$$

- We want to prove Lemma4.
  - A proof of Lemma4 will be an element lemma4 : Lemma4.
- So we have to solve the following goal:

$$\begin{array}{lcl} \mathrm{lemma4} & : & \mathrm{Lemma4} \\ \mathrm{lemma4} & = & \{!\ \ !\} \end{array}$$

- The type of the goal is

$$\mathrm{Lemma4} = (a : \mathrm{Bool}) \to (a < a) \to \bot$$

Example ($\forall$, Cont.)

$$\begin{array}{lcl} \mathrm{lemma4} & : & \mathrm{Lemma4} \\ \mathrm{lemma4} & = & \{!\ \ !\} \end{array}$$

Type of goal is $\mathrm{Lemma4} = (a : \mathrm{Bool}) \to (a < a) \to \perp$.

- An element $\mathrm{lemma4} : (a : \mathrm{Bool}) \to (a < a) \to \perp$ can be introduced by applying it to $a : A$ and $aa : a < a$:

$$\begin{array}{l} \mathrm{lemma4} : \mathrm{Lemma4} \\ \mathrm{lemma4}\ a\ aa = \{!\ \ !\} \end{array}$$

- The type of goal is now the conclusion of $(a : \mathrm{Bool}) \to (a < a) \to \perp$, namely $\perp$.

Example ($\forall$, Cont.)

$\mathrm{lemma4} : \mathrm{Lemma4}$

$\mathrm{lemma4}\ a\ aa = \{!\ \ !\}$

Type of goal is $\perp$.

- We need to make use of our assumptions, namely $a : \mathrm{Bool}$ and $aa : a < a$.
  - $a < b$ is defined by case disjunction on a and b.
    - Unless we know that $a = \mathrm{tt}$ or $a = \mathrm{ff}$, we don't know much about $a < a$.
    - So it seems to be a good step to make pattern matching using the cases $a = \mathrm{tt}$ and $a = \mathrm{ff}$.

Example ($\forall$, Cont.)

$$\begin{array}{l} \mathrm{lemma4} : \mathrm{Lemma4} \\ \begin{array}{llllll} \mathrm{lemma4} & \mathrm{ff} & aa & = & \{! & !\} \\ \mathrm{lemma4} & \mathrm{tt} & aa & = & \{! & !\} \end{array} \end{array}$$

- The type of both goals is the same as before, namely $\perp$, since it didn't depend on a.

Example ($\forall$, Cont.)

$$\begin{array}{l}\mathrm{lemma4} : \mathrm{Lemma4} \\ \mathrm{lemma4} \quad \mathrm{ff} \quad aa \quad = \quad \{!\ \ !\} \\ \mathrm{lemma4} \quad \mathrm{tt} \quad aa \quad = \quad \{!\ \ !\}\end{array}$$

- However, we know now more about the assumptions $aa : a < a$.
  - In case of $a = \mathrm{ff}$, we have $aa : (a < a) = (\mathrm{ff} < \mathrm{ff}) = \perp$
    - So there is no case for $aa : \perp$, and we can solve this case by

$$\mathrm{lemma4}\ \mathrm{ff}\ ()$$

Example ($\forall$, Cont.)

$$
\begin{array}{llll}
\mathrm{lemma4} : \mathrm{Lemma4} & & & \\
\mathrm{lemma4} & \mathrm{ff} & () & \\
\mathrm{lemma4} & \mathrm{tt} & aa & = \{!\ \ !\}
\end{array}
$$

- In case of $a = \mathrm{tt}$, we have $aa : (a < a) = (\mathrm{tt} < \mathrm{tt}) = \perp$
  - Again we can solve this case by

$$\mathrm{lemma4}\ \mathrm{tt}\ ()$$

We obtain the code

$$
\begin{array}{lll}
\mathrm{lemma4} : \mathrm{Lemma4} & & \\
\mathrm{lemma4} & \mathrm{ff} & () \\
\mathrm{lemma4} & \mathrm{tt} & ()
\end{array}
$$

Example ($\forall$, Cont.)

- In the previous example,
  - the type of goal was $\perp$,
  - and $aa : \perp$.
- So, instead of using case distinction on aa we could have as well inserted aa in those goals:

$$
\begin{array}{l}
\mathrm{lemma4} : \mathrm{Lemma4} \\
\begin{array}{lllll}
\mathrm{lemma4} & \mathrm{ff} & aa & = & aa \\
\mathrm{lemma4} & \mathrm{tt} & aa & = & aa
\end{array}
\end{array}
$$

(f) 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:
  - $\mathrm{Sorted}\ l$ is a predicate on $\mathrm{NatList}$ expressing that l is sorted.
  - An element of
    $$\mathrm{SortedList} := (l : \mathrm{NatList}) \times \mathrm{Sorted}\ l$$
    is a pair
    $$\langle l, p \rangle$$
    s.t.
    - $l : \mathrm{NatList}$,
    - $p : \mathrm{Sorted}\ l$, i.e. p is a **proof** that l is sorted.
  - So elements of $\mathrm{SortedList}$ are lists l together with a proof that l is sorted.

Example 2 (Dep. Products)

- Remember the Gender-example as in the last section:
  - $\mathrm{Gender} = \{\mathrm{female}, \mathrm{male}\}$.
  - For $g : \mathrm{Gender}$

    $$\mathrm{Name}\ g$$

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

$$\mathrm{NameWithGender} := (g : \mathrm{Gender}) \times \mathrm{Name}\ g$$

Rules of the Dependent Product

Formation Rule

$$\frac{A : \mathrm{Set} \qquad x : A \Rightarrow B : \mathrm{Set}}{(x : A) \times B : \mathrm{Set}} \, (\times\text{-F})$$

Introduction Rule

$$\frac{x : A \Rightarrow B : \mathrm{Set} \qquad a : A \qquad b : B[x := a]}{\langle a, b \rangle : (x : A) \times B} \, (\times\text{-I})$$

Extra Premise in the Introd. Rule

- In the last introduction rule, an **extra premise** $x : A \Rightarrow B : \mathrm{Set}$ 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 $A : \mathrm{Set}$ and $B : \mathrm{Set}$, from which it follows $A \times B : \mathrm{Set}$.

Example

- Assuming we have defined the set of genders $\mathrm{Gender} : \mathrm{Set}$ and the set of names $g : \mathrm{Gender} \Rightarrow \mathrm{Name}\ g : \mathrm{Set}$, we can introduce the set

$$\mathrm{NameWithGender} := (g : \mathrm{Gender}) \times \mathrm{Name}\ g : \mathrm{Set}$$

by using the formation rule:

$$\frac{\mathrm{Gender} : \mathrm{Set} \qquad g : \mathrm{Gender} \Rightarrow \mathrm{Name}\ g : \mathrm{Set}}{(g : \mathrm{Gender}) \times \mathrm{Name}\ g : \mathrm{Set}} \, (\times\text{-I})$$

Example

- Furthermore we can introduce

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

  as follows:

$$\frac{g{:}\mathrm{Gender}{\Rightarrow}\mathrm{Name}\ g{:}\mathrm{Set} \qquad \mathrm{male}{:}\mathrm{Gender} \qquad \mathrm{tom}{:}\mathrm{Name\ male}}{\langle \mathrm{male}, \mathrm{tom} \rangle : (g : \mathrm{Gender}) \times \mathrm{Name}\ g} (\times\text{-I})$$

- Note that we need the premise

$$g : \mathrm{Gender} \Rightarrow \mathrm{Name}\ g : \mathrm{Set}$$

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

Jump to the elimination rules for the product.

Example

- Note that we **don't** have

$$\langle \mathrm{female}, \mathrm{tom} \rangle : \mathrm{NameWithGender}$$

since we **don't** have

$$\mathrm{tom} : \mathrm{Name}\ \mathrm{female}$$

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

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

but not

$$\langle \mathrm{female}, \mathrm{tom} \rangle : \mathrm{NameWithGender}$$

Rules of the Dependent Product

Elimination Rules

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

Equality Rules

$$\frac{x : A \Rightarrow B : \mathrm{Set} \qquad a : A \qquad b : B[x := a]}{\pi_0(\langle a, b\rangle) = a : A}\,(\times\text{-Eq}_0)$$

$$\frac{x : A \Rightarrow B : \mathrm{Set} \qquad a : A \qquad b : B[x := a]}{\pi_1(\langle a, b\rangle) = b : B[x := a]}\,(\times\text{-Eq}_1)$$

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

Example

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

$$\frac{a : (g : \mathrm{Gender}) \times \mathrm{Name}\ g}{\pi_0(a) : \mathrm{Gender}}\ (\times\text{-}\mathrm{El}_0)$$

$$\frac{a : (g : \mathrm{Gender}) \times \mathrm{Name}\ g}{\pi_1(a) : \mathrm{Name}\ \pi_0(a)}\ (\times\text{-}\mathrm{El}_1)$$

Example

- Furthermore

$$\pi_0(\langle \mathrm{male}, \mathrm{tom} \rangle) = \mathrm{male} : \mathrm{Gender}$$

therefore

$$\mathrm{Name}\ \pi_0(\langle \mathrm{male}, \mathrm{tom} \rangle) = \mathrm{Name\ male}$$

$$\pi_1(\langle \mathrm{male}, \mathrm{tom} \rangle) = \mathrm{tom} : \mathrm{Name}\ \pi_0(\langle \mathrm{male}, \mathrm{tom} \rangle)$$

therefore as well

$$\pi_1(\langle \mathrm{male}, \mathrm{tom} \rangle) = \mathrm{tom} : \mathrm{Name\ male}$$

Rules of the Dependent Product

We have the following **η-rule**:

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

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

Equality Versions of the above

Equality Version of the Formation Rule

$$\frac{A = A' : \mathrm{Set} \qquad x : A \Rightarrow B = B' : \mathrm{Set}}{(x : A) \times B = (x : A') \times B' : \mathrm{Set}} \,(\times\text{-F}^{=})$$

Equality Version of the Introduction Rule

$$\frac{x : A \Rightarrow B : \mathrm{Set} \qquad a = a' : A \qquad b = b' : B[x := a]}{\langle a, b \rangle = \langle a', b' \rangle : (x : A) \times B} \,(\times\text{-I}^{=})$$

Equality Versions of the Elimination Rules

$$\frac{c = c' : (x : A) \times B}{\pi_0(c) = \pi_0(c') : A} \,(\times\text{-El}_0^{=}) \qquad \frac{c = c' : (x : A) \times B}{\pi_1(c) = \pi_1(c') : B[x := \pi_0(c)]} \,(\times\text{-El}_1^{=})$$

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 Abbre

- More precisely this can be seen as follows:
  - From $A : \mathrm{Set}$ and $B : \mathrm{Set}$ we can derive $x : A \Rightarrow B : \mathrm{Set}$ 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 $A : \mathrm{Set}$ (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 η-rule.

The Dependent Product in Agda

- In Agda, the record type allows already dependencies of later sets on previous ones:
  - Assume $A : \mathrm{Set}$, and $B : \mathrm{Set}$, possibly depending on $a : A$.
  - Then we can form

$$
\begin{array}{l}
\mathrm{record\ AB : Set\ where} \\
\quad \mathrm{field} \\
\qquad\qquad \begin{array}{lcl} a & : & A \\ b & : & B \end{array}
\end{array}
$$

The Dependent Product in Agda

$$\begin{array}{l} \text{record AB : Set where} \\ \quad \text{field} \\ \qquad\qquad a \quad : \quad A \\ \qquad\qquad b \quad : \quad B \end{array}$$

- 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

$$\text{record } \{a : A = a'; b : B = b'\} : \mathrm{AB} \quad .$$

- Note that $b' : B[a := a']$, so the type of b' depends on a'.
- Furthermore, if $ab : \mathrm{AB}$, then
  $\mathrm{AB}.a \; ab : A$,
  $\mathrm{AB}.b \; ab : B[a := \mathrm{AB}.a \; ab]$.
  dependentProduct1.agda

The Dependent Product in Agda

- The same applies to the dependent product using data.
  - Assume $A : \mathrm{Set}$, and $B : \mathrm{Set}$, possibly depending on $a : A$.
  - Then we can form

$$\begin{aligned}&\mathrm{data}\ \mathrm{AB} : \mathrm{Set}\ \mathrm{where}\\ &\quad \mathrm{prod} : (a' : A) \to B[a := a'] \to \mathrm{AB}\end{aligned}$$

  - 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

$$\mathrm{prod}\ a'\ b' : \mathrm{AB} \quad .$$

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

The Dependent Product in Agda

- Furthermore, we can define the projections:

$$
\begin{array}{l}
\pi_0 \quad : \quad \mathrm{AB} \longrightarrow A \\
\pi_0 \; (\mathrm{p} \; a \; b) = a \\
\pi_1 \quad : \quad (ab : \mathrm{AB}) \longrightarrow B[a := \pi_0 \; ab] \\
\pi_1 \; (\mathrm{p} \; a \; b) = b
\end{array}
$$

dependentProduct1.agda

The “Name”-Example in Agda

- Remember:

$$
\begin{array}{l}
\text{data Gender} : \text{Set where} \\
\quad \begin{array}{lcl} \text{female} & : & \text{Gender} \\ \text{male} & : & \text{Gender} \end{array}
\end{array}
$$

$$
\begin{array}{l}
\text{data FemaleName} : \text{Set where} \\
\quad \begin{array}{lcl} \text{jill} & : & \text{FemaleName} \\ \text{sara} & : & \text{FemaleName} \end{array}
\end{array}
$$

$$
\begin{array}{l}
\text{data MaleName} : \text{Set where} \\
\quad \begin{array}{lcl} \text{tom} & : & \text{MaleName} \\ \text{jim} & : & \text{MaleName} \end{array}
\end{array}
$$

The “Name”-Example in Agda

$$
\begin{aligned}
&\mathrm{data}\ \mathrm{MaleName} : \mathrm{Set}\ \mathrm{where}\\
&\qquad \mathrm{tom} \quad : \quad \mathrm{MaleName}\\
&\qquad \mathrm{jim} \quad : \quad \mathrm{MaleName}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{data}\ \mathrm{FemaleName} : \mathrm{Set}\ \mathrm{where}\\
&\qquad \mathrm{jill} \quad : \quad \mathrm{FemaleName}\\
&\qquad \mathrm{sara} \quad : \quad \mathrm{FemaleName}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Name} : \mathrm{Gender} \longrightarrow \mathrm{Set}\\
&\mathrm{Name}\ \mathrm{male} \quad = \quad \mathrm{MaleName}\\
&\mathrm{Name}\ \mathrm{female} \quad = \quad \mathrm{FemaleName}
\end{aligned}
$$

The “Name”-Example in Agda

- Now we define

$$
\begin{array}{l}
\mathrm{record\ NameWithGender : Set\ where} \\
\quad \mathrm{field} \\
\qquad \begin{array}{lcl}
\mathrm{gender} & : & \mathrm{Gender} \\
\mathrm{name} & : & \mathrm{Name\ gender}
\end{array}
\end{array}
$$

  See **exampleAllNames.agda**.

The “Name”-Example in Agda

- Note that we have

$$\text{record } \{\text{gender} = \text{male}; \text{name} = \text{tom}\} : \text{NameWithGender}$$

whereas we **don’t** have

$$\text{record } \{\text{gender} = \text{male}; \text{name} = \text{jill}\} : \text{NameWithGender}$$

- This is different from the dependent record type which occurs for instance in Pascal or Ada, where the second example doesn’t result in a type error.

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 $\langle \mathbf{a}, \mathbf{p} \rangle$ consisting of an element $\mathbf{a} : \mathbf{A}$ and a proof p of $\mathbf{B}[\mathbf{x} := \mathbf{a}]$**.
- Therefore the set of proofs of $\exists x^A.B$ is the **dependent product $(\mathbf{x} : \mathbf{A}) \times \mathbf{B}$**.
- We can **identify $\exists \mathbf{x}^{\mathbf{A}}.\mathbf{B}$** with $(\mathbf{x} : \mathbf{A}) \times \mathbf{B}$.

$\exists$ in Agda

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

$$
\begin{array}{l}
\mathrm{record\ Version1 : Set\ where} \\
\quad \mathrm{field} \\
\qquad \begin{array}{lcl} a & : & A \\ b & : & B[x := a] \end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{data\ Version2 : Set\ where} \\
\quad \mathrm{exists} : (a : A) \to B[x := a] \to \mathrm{Version2}
\end{array}
$$

- Here $B[x := a]$ is the result of substituting in B for x the variable a.

$\exists$ in Agda

- A generic version, depending on $A : \mathrm{Set}$ and $B : A \longrightarrow \mathrm{Set}$ can be defined as follows
(The symbol $\exists$ can be obtained by typing in "\exists"):

$$
\begin{array}{l}
\mathrm{record}\ \exists\mathrm{r}\ (A : \mathrm{Set})\ (B : A \longrightarrow \mathrm{Set}) : \mathrm{Set\ where} \\
\quad \mathrm{field} \\
\qquad \begin{array}{lcl} a & : & A \\ b & : & B\ a \end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{data}\ \exists\mathrm{d}\ (A : \mathrm{Set})\ (B : A \longrightarrow \mathrm{Set}) : \mathrm{Set\ where} \\
\quad \mathrm{exists} : (a : A) \longrightarrow B\ a \longrightarrow \exists\mathrm{d}\ A\ B
\end{array}
$$

existentialQuantification.agda

Example ($\exists$)

- As an example,
  - we define negation $\neg\mathrm{Bool}$ on Bool,
  - define an equality $==$ on Bool,
  - and show $\forall a^{\mathrm{Bool}}.\exists b^{\mathrm{Bool}}.a == \neg\mathrm{Bool}\ b$.
- See **exampleproofproplogic11.agda**.

Example ($\exists$, Cont.)

- $\neg\mathrm{Bool}$ is defined as follows:

$$\begin{aligned} \neg\mathrm{Bool} &: \mathrm{Bool} \to \mathrm{Bool} \\ \neg\mathrm{Bool} \quad \mathrm{tt} &= \mathrm{ff} \\ \neg\mathrm{Bool} \quad \mathrm{ff} &= \mathrm{tt} \end{aligned}$$

Example (∃)

- A Boolean valued equality on Bool is defined as follows:

$$
\begin{array}{l}
\_{==}\mathrm{Bool}\_ : \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool} \\
\begin{array}{lllll}
\mathrm{tt} & {==}\mathrm{Bool} & b & = & b \\
\mathrm{ff} & {==}\mathrm{Bool} & b & = & \neg\mathrm{Bool}\ b
\end{array}
\end{array}
$$

- This corresponds to the following truth table:

$==\mathrm{Bool}$	ff	tt
ff	tt	ff
tt	ff	tt

Example (∃)

- Then we define

$$
\begin{aligned}
\_{==}\_ &: \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Set} \\
b \;==\; b' &= \mathrm{Atom}\ (b ==_{\mathrm{Bool}} b')
\end{aligned}
$$

Example (∃, Cont.)

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

$$
\begin{array}{l}
\mathrm{record\ Lemma5aux}\ (a : \mathrm{Bool}) : \mathrm{Set\ where} \\
\quad \mathrm{field} \\
\qquad \begin{array}{lcl}
b & : & \mathrm{Bool} \\
ab & : & a == \neg\mathrm{Bool}\ b
\end{array}
\end{array}
$$

- The statement $\forall a^{\mathrm{Bool}}.\exists b^{\mathrm{Bool}}.a == \neg\mathrm{Bool}\ b$ is now as follows:

$$
\begin{array}{lcl}
\mathrm{Lemma5} & : & \mathrm{Set} \\
\mathrm{Lemma5} & = & (a : \mathrm{Bool}) \rightarrow \mathrm{Lemma5aux}\ a
\end{array}
$$

Example (∃, Cont.)

- A proof of Lemma5 is an element

$$\mathrm{lemma5} : \mathrm{Lemma5}$$

and we get the goal

$$\begin{array}{lcl} \mathrm{lemma5} & : & \mathrm{Lemma5} \\ \mathrm{lemma5} & = & \{!\ \ !\} \end{array}$$

- The type of goal is

$$\mathrm{Lemma5} \quad = \quad (a : \mathrm{Bool}) \to \mathrm{Lemma5aux}\ a$$

- This goal is solved by applying $\mathrm{lemma5}$ to $a : \mathrm{Bool}$.

Example ($\exists$, Cont.)

$$
\begin{array}{lcl}
\mathrm{Lemma5} & : & \mathrm{Set} \\
\mathrm{Lemma5} & = & (a : \mathrm{Bool}) \to \mathrm{Lemma5aux}\ a
\end{array}
$$

- We get

$$
\begin{array}{lcl}
\mathrm{lemma5} & : & \mathrm{Lemma5} \\
\mathrm{lemma5}\ a & = & \{!\ \ !\}
\end{array}
$$

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

$$\mathrm{Lemma5aux}\ a = \mathrm{record}\ \{b : \mathrm{Bool}; ab : a == \neg\mathrm{Bool}\ b\}$$

Example (∃, Cont.)

$$\begin{array}{lcl} \mathrm{lemma5} & : & \mathrm{Lemma5} \\ \mathrm{lemma5}\ a & = & \{!\ \ !\} \end{array}$$

Type of goal is

$$\mathrm{record}\ \{b : \mathrm{Bool}; ab : a == \neg\mathrm{Bool}\ b\}$$

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

Example (∃, Cont.)

- We get

$$\mathrm{lemma5} : \mathrm{Lemma5}$$
$$\begin{array}{lllll} \mathrm{lemma5} & \mathrm{ff} & = & \{! \quad !\} \\ \mathrm{lemma5} & \mathrm{tt} & = & \{! \quad !\} \end{array}$$

Example (∃, Cont.)

$$\mathrm{lemma5} : \mathrm{Lemma5}$$
$$\begin{array}{llll}\mathrm{lemma5} & \mathrm{ff} & = & \{!\ \ !\} \\ \mathrm{lemma5} & \mathrm{tt} & = & \{!\ \ !\}\end{array}$$

- In case of $a = \mathrm{ff}$, the type of goal is

$$\begin{array}{lllll}\mathrm{Lemma5aux}\ \mathrm{ff} & = & \mathrm{record}\ \{ & b & : & \mathrm{Bool}; \\ & & & ab & : & \mathrm{ff} == \neg\mathrm{Bool}\ b\}\end{array}$$

- This goal can be solved as follows

$$\mathrm{lemma5}\ \mathrm{ff} = \mathrm{record}\ \{b = \mathrm{tt}; ab = \mathrm{true}\}$$

(Note that $(\mathrm{ff} == \neg\mathrm{Bool}\ \mathrm{tt}) = \top$, so $\mathrm{true} : (\mathrm{ff} == \neg\mathrm{Bool}\ \mathrm{tt})$).

Example (∃, Cont.)

$$\begin{array}{l}\mathrm{lemma5} : \mathrm{Lemma5} \\ \mathrm{lemma5} \quad \mathrm{ff} \quad = \quad \mathrm{record}\ \{b = \mathrm{tt}; ab = \mathrm{true}\} \\ \mathrm{lemma5} \quad \mathrm{tt} \quad = \quad \{!\ \ !\}\end{array}$$

- The second goal can be solved as follows

$$\mathrm{lemma5}\ \mathrm{tt} = \mathrm{record}\ \{b = \mathrm{ff}; ab = \mathrm{true}\}$$

- So we get the complete proof:

$$\begin{array}{l}\mathrm{lemma5} : \mathrm{Lemma5} \\ \mathrm{lemma5} \quad \mathrm{ff} \quad = \quad \mathrm{record}\ \{b = \mathrm{tt}; ab = \mathrm{true}\} \\ \mathrm{lemma5} \quad \mathrm{tt} \quad = \quad \mathrm{record}\ \{b = \mathrm{ff}; ab = \mathrm{true}\}\end{array}$$

Complex Example

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

$$
\begin{array}{llll}
\mathrm{postulate} & \mathrm{A} & : & \mathrm{Set} \\
\mathrm{postulate} & \_{==}\mathrm{A}\_ & : & \mathrm{A} \to \mathrm{A} \to \mathrm{Set} \\
\mathrm{postulate} & \mathrm{B} & : & \mathrm{Set} \\
\mathrm{postulate} & \_{==}\mathrm{B}\_ & : & \mathrm{B} \to \mathrm{B} \to \mathrm{Set}
\end{array}
$$

- We will introduce
  - the product AB of A and B
  - an equality $==\mathrm{AB}$ on AB
  - and show that if $==\mathrm{A}$ and $==\mathrm{B}$ are symmetric, so is $==\mathrm{AB}$.
- See **exampleProductEqual.agda**.

Equality Sets

- $==\mathrm{A}$ (and $==\mathrm{B}$) could be decidable equalities,
  - i.e. $==\mathrm{A} = \lambda(a, b : A) \rightarrow \mathrm{Atom}\ (\mathrm{eqboolA}\ a\ b)$, where $\mathrm{eqboolA} : A \rightarrow A \rightarrow \mathrm{Bool}$,
- Or an undecidable equality.
  - E.g. the equality on $\mathbb{N} \rightarrow \mathbb{N}$ is in standard logic

$$f = g :\Leftrightarrow \forall n^{\mathbb{N}}.f(n) = g(n)$$

which reads in Agda as follows:

$$\begin{array}{l} \_{==}\mathbb{N}{\rightarrow}\_ : (f\ g : \mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathrm{Set} \\ f\ \ {==}\mathbb{N}{\rightarrow}\ \ g = (n : \mathbb{N}) \rightarrow f\ n == g\ n \end{array}$$

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

Undecidable Equalities

- The last equality is undecidable, since in order to check whether $f \;==_{\mathbb{N}\rightarrow}\; g$ holds we have to check **for all** $n : \mathbb{N}$ whether $f\ n = g\ n$ holds

Complex Example (Cont.)

- The formation of $\mathrm{AB} = \mathrm{A} \times \mathrm{B}$ is straightforward:

$$
\begin{array}{l}
\mathrm{data}\ \_ \times \_ (A\ B : \mathrm{Set}) : \mathrm{Set}\ \mathrm{where} \\
\quad \mathrm{p} : A \to B \to A \times B
\end{array}
$$

$$
\begin{array}{l}
\mathrm{AB} : \mathrm{Set} \\
\mathrm{AB} = \mathrm{A} \times \mathrm{B}
\end{array}
$$

Complex Example (Cont.)

- We define the equality $==\mathrm{AB}$ 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. $==\mathrm{A}$ and their second projections are equal w.r.t. $==\mathrm{B}$.
  - So we get

$$
\begin{array}{l}
\_{==}\mathrm{AB}\_ : \mathrm{AB} \to \mathrm{AB} \to \mathrm{Set} \\
(\mathrm{p}\ a\ b)\ \ {==}\mathrm{AB}\ \ (\mathrm{p}\ a'\ b') = (a\ \ {==}\mathrm{A}\ \ a') \wedge (b\ \ {==}\mathrm{B}\ \ b')
\end{array}
$$

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:

$$\begin{array}{l}
\mathrm{Sym} : (A : \mathrm{Set}) \rightarrow (A \rightarrow A \rightarrow \mathrm{Set}) \rightarrow \mathrm{Set} \\
\mathrm{Sym} \;\; A \;\; _{==}_ \;\; = (a \; a' : A) \rightarrow a == a' \rightarrow a' == a
\end{array}$$

Specialisation of Sym

- We create instances of Sym for symmetry on A, B, AB:

$$
\begin{array}{lcl}
\mathrm{SymA} & : & \mathrm{Set} \\
\mathrm{SymA} & = & \mathrm{Sym}\ \ \mathrm{A}\ \ \_{==}\mathrm{A}\_ \\
\\
\mathrm{SymB} & : & \mathrm{Set} \\
\mathrm{SymB} & = & \mathrm{Sym}\ \ \mathrm{B}\ \ \_{==}\mathrm{B}\_ \\
\\
\mathrm{SymAB} & : & \mathrm{Set} \\
\mathrm{SymAB} & = & \mathrm{Sym}\ \ \mathrm{AB}\ \ \_{==}\mathrm{AB}\_
\end{array}
$$

Formulae vs. Proofs

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

$$\mathrm{symA} : \mathrm{SymA}$$

- 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

$$\begin{aligned} \mathrm{symAB} \quad &: \quad \mathrm{SymA} \rightarrow \mathrm{SymB} \rightarrow \mathrm{SymAB} \\ \mathrm{symAB} \quad &= \quad \{!\ \ !\} \end{aligned}$$

- We apply symAB to elements $symA : \mathrm{SymA}$, $symB : \mathrm{SymB}$ and obtain

$$\begin{aligned} \mathrm{symAB} \quad &: \quad \mathrm{SymA} \rightarrow \mathrm{SymB} \rightarrow \mathrm{SymAB} \\ \mathrm{symAB}\ symA\ symB \quad &= \quad \{!\ \ !\} \end{aligned}$$

Complex Example

$$\begin{array}{lcl} \mathrm{symAB} & : & \mathrm{SymA} \to \mathrm{SymB} \to \mathrm{SymAB} \\ \mathrm{symAB}\ \mathit{symA}\ \mathit{symB} & = & \{!\ \ !\} \end{array}$$

- The type of the goal is SymAB which is

$$(ab\ ab' : \mathrm{AB}) \ \to\ ab\ ==_{\mathrm{AB}}\ ab' \ \to\ ab'\ ==_{\mathrm{AB}}\ ab$$

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

$$\begin{array}{l} \mathrm{symAB} : \mathrm{SymA} \to \mathrm{SymB} \to \mathrm{SymAB} \\ \mathrm{symAB}\ \mathit{symA}\ \mathit{symB}\ ab\ ab'\ abab' = \{!\ \ !\} \end{array}$$

Complex Example

$$\mathrm{symAB} : \mathrm{SymA} \rightarrow \mathrm{SymB} \rightarrow \mathrm{SymAB}$$
$$\mathrm{symAB}\ symA\ symB\ ab\ ab'\ abab' = \{!\ \ !\}$$

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

$$\mathrm{symAB} : \mathrm{SymA} \rightarrow \mathrm{SymB} \rightarrow \mathrm{SymAB}$$
$$\mathrm{symAB}\ symA\ symB\ (\mathrm{p}\ a\ b)\ (\mathrm{p}\ a'\ b')\ abab' = \{!\ \ !\}$$

Complex Example

$\mathrm{symAB} : \mathrm{SymA} \to \mathrm{SymB} \to \mathrm{SymAB}$
$\mathrm{symAB}\ symA\ symB\ (\mathrm{p}\ a\ b)\ (\mathrm{p}\ a'\ b')\ abab' = \{!\ \ !\}$

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

$$\mathrm{symAB} : \mathrm{SymA} \to \mathrm{SymB} \to \mathrm{SymAB}$$
$$\mathrm{symAB}\ symA\ symB\ (\mathrm{p}\ a\ b)\ (\mathrm{p}\ a'\ b')\ (\mathrm{and}\ aa'\ bb') = \{!\ \ !\}$$

Complex Example

$$\mathrm{symAB} : \mathrm{SymA} \to \mathrm{SymB} \to \mathrm{SymAB}$$
$$\mathrm{symAB}\ \mathit{symA}\ \mathit{symB}\ (\mathrm{p}\ a\ b)\ (\mathrm{p}\ a'\ b')\ (\mathrm{and}\ aa'\ bb') = \{!\ \ !\}$$

- The Type of the goal is

$$(a' \ ==_{\mathrm{A}}\ a) \wedge (b' \ ==_{\mathrm{B}}\ b)$$

- Elements of it are of the form $\mathrm{p}\ ab\ ab'$ with $a'a : a' \ ==_{\mathrm{A}}\ a$ and $b'b : b' \ ==_{\mathrm{B}}\ b$.
- So we insert into the goal p and use intro.
  We obtain

$$\mathrm{symAB} : \mathrm{SymA} \to \mathrm{SymB} \to \mathrm{SymAB}$$
$$\mathrm{symAB}\ \mathit{symA}\ \mathit{symB}\ (\mathrm{p}\ a\ b)\ (\mathrm{p}\ a'\ b')\ (\mathrm{and}\ aa'\ bb')$$
$$= \mathrm{p}\ \{!\ \ !\}\ \{!\ \ !\}$$

Complex Example

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

- The type of the first goal is $a' \ ==_{\mathrm{A}}\ a$.
- We have $aa' : a \ ==_{\mathrm{A}}\ a'$ and
  $symA : (a\ a' : A) \rightarrow a \ ==_{\mathrm{A}}\ a' \rightarrow a' \ ==_{\mathrm{A}}\ a$.
- So
  $$symA\ a\ a'\ aa' : a' \ ==_{\mathrm{A}}\ a$$
  and this term can be used in order to solve the first goal:

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

Complex Example

$$\mathrm{symAB} : \mathrm{SymA} \to \mathrm{SymB} \to \mathrm{SymAB}$$
$$\mathrm{symAB}\ \mathit{symA}\ \mathit{symB}\ (\mathrm{p}\ a\ b)\ (\mathrm{p}\ a'\ b')\ (\mathrm{and}\ aa'\ bb')$$
$$= \mathrm{p}\ (\mathit{symA}\ a\ a'\ aa')\ \{!\ \ !\}$$

- The type of the second goal is $b' \ \ {==}\mathrm{B}\ \ b$ which can be solved by $\mathit{symB}\ b\ b'\ bb'$.
- We obtain

$$\mathrm{symAB} : \mathrm{SymA} \to \mathrm{SymB} \to \mathrm{SymAB}$$
$$\mathrm{symAB}\ \mathit{symA}\ \mathit{symB}\ (\mathrm{p}\ a\ b)\ (\mathrm{p}\ a'\ b')\ (\mathrm{and}\ aa'\ bb')$$
$$= \mathrm{p}\ (\mathit{symA}\ a\ a'\ aa')\ (\mathit{symB}\ b\ b'\ bb')$$

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

$$\lambda(\mathbf{a} : \mathbf{A}).\mathbf{a} : \mathbf{A} \to \mathbf{A}$$

  (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:

$$\mathrm{postulate}\ A : \mathrm{Set}$$

Example 1 (Cont.)

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

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

Example 1 (Cont.)

- **Step 3:**
  - We want to derive an element of function type $A \to A$.
  - Elements of the function type $A \to A$ are introduced by using **λ-terms**.
  - If introduced as a λ-term, the term in question will be of the form $\lambda(a : A) \to \mathbf{something}$.
  - So we insert into the goal $\lambda(a : A) \to \{! \ \ !\}$, use **agda-give** and obtain

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

(The precise Agda code uses $\backslash$ instead of λ, and $->$ instead of $\to$).

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 λ-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**.

Example 1 (Cont.)

- **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.

Example 1 (Cont.)

- **Step 4 (Cont.)**
  - Now everything with **result type** A (i.e. which has at the right side of the arrow A) can be used in order to solve the goal.
    - f would result in black-hole recursion.
    - So we take a.
  - We type in a into the goal and then use the command **Refine**
  - We obtain:

$$\begin{aligned} f &: A \to A \\ &= \lambda(a : A) \to a \end{aligned}$$

and are done.
derivationsagdacode1.agda

Example 1, Using Rules

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

$$f\colon A \longrightarrow A$$
$$= \{!\ \ !\}$$

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

$$d_0 : A \longrightarrow A$$

Example 1, Using Rules (Cont.)

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

$$
\begin{aligned}
f &: A \to A \\
&= \lambda(a : A) \to \{!\ \ !\}
\end{aligned}
$$

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

$$
\frac{a : A \Rightarrow d_1 : A}{\lambda a^A.d_1 : A \to A}\,(\to\text{-I})
$$

Example 1, Using Rules (Cont.)

- In **Agda step 4** we replaced $\{! \ \ !\}$ in $\lambda(a : A) \to \{! \ \ !\}$ by a:

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

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

$$\frac{a : A \Rightarrow a : A}{\lambda a^A.a : A \to A} (\to \text{-I})$$

- The assumption rule will be discussed later.
  - Essentially it allows to derive if $x : B$ occurs in the context that $x : B$ holds.

Example 2

- We consider a derivation of

$$\lambda(a{-}a{-}a : (A \to A) \to A).a{-}a{-}a\ (\lambda(a : A) \to a)$$
$$: ((A \to A) \to A) \to A$$

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

- **Step 1:**
  - We postulate A:

$$\text{postulate } A : \text{Set}$$

  - We state our goal:

$$f : ((A \to A) \to A) \to A$$
$$f = \{!\ \ !\}$$

Example 2 (Cont.)

- **Step 2:**
  - The type of the goal is a function type.
    We therefore insert into the goal
    $\lambda(a{-}a{-}a : (A \to A) \to A) \to \{!\ \ !\}$, use goal
    command **Refine** and obtain
  - We obtain

$$
\begin{aligned}
f &: ((A \to A) \to A) \to A \\
f &= \ \lambda(a{-}a{-}a : (A \to A) \to A) \to \{!\ \ !\}
\end{aligned}
$$

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 $a{-}a{-}a : (A \to A) \to A$.
  - We can as well use f (for recursive definitions) and A for solving the goal.
  - $a{-}a{-}a$ 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.

Example 2 (Cont.)

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

$$
\begin{aligned}
&f : ((A \longrightarrow A) \longrightarrow A) \longrightarrow A \\
&f = \ \lambda(a{-}a{-}a : (A \longrightarrow A) \longrightarrow A) \longrightarrow a{-}a{-}a \ \{! \ \ !\}
\end{aligned}
$$

Example 2 (Cont.)

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

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

Example 2 (Cont.)

- **Step 5**
  - The new goal has type A.
    - The complete expression $\lambda(a : A) \to \{! \quad !\}$ should have type $A \to A$, so $\{! \quad !\}$ must have type A.
  - The context contains $a{-}a{-}a$ and a; we can use as well f, A.
    - Both $a{-}a{-}a$ 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.

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

$$f = \ \lambda(a{-}a{-}a : (A \to A) \to A) \to a{-}a{-}a\ (\lambda(a : A) \to a)$$

Example 2, Using Rules

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

Example 2, Using Rules

- 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:

$$\frac{aaa : (A \to A) \to A \Rightarrow d_1 : A}{\lambda aaa^{((A \to A) \to A)}.d_1 : ((A \to A) \to A) \to A}\,(\to\text{-I})$$

Example 2, Using Rules

- The next step in the Agda-derivation used refine. $\{!\ \ !\}$ was replaced by $aaa\ \{!\ \ !\}$.
- This corresponds to replacing d_1 by $aaa\ d_2$, and using one elimination rule in order to derive it:

$$\dfrac{\dfrac{aaa{:}(A{\rightarrow}A){\rightarrow}A \Rightarrow aaa{:}(A{\rightarrow}A){\rightarrow}A \qquad aaa{:}(A{\rightarrow}A){\rightarrow}A \Rightarrow d_2{:}A{\rightarrow}A}{aaa{:}(A{\rightarrow}A){\rightarrow}A \Rightarrow aaa\ d_2{:}A}\,(\rightarrow\text{-El})}{\lambda aaa^{(A\rightarrow A)\rightarrow A}.aaa\ d_2{:}((A{\rightarrow}A){\rightarrow}A){\rightarrow}A}\,(\rightarrow\text{-I})$$

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

$$\dfrac{aaa{:}(A{\to}A){\to}A{\Rightarrow}aaa{:}(A{\to}A){\to}A \qquad \dfrac{aaa{:}(A{\to}A){\to}A,a{:}A{\Rightarrow}d_3{:}A}{aaa{:}(A{\to}A){\to}A{\Rightarrow}\lambda a^A.d_3{:}A{\to}A}\,(\to\text{-I})}{\dfrac{aaa{:}(A{\to}A){\to}A{\Rightarrow}aaa\ (\lambda a^A.d_3){:}A}{(\lambda aaa^{(A\to A)\to A}.aaa)\ (\lambda a^A.d_3){:}((A{\to}A){\to}A){\to}A}\,(\to\text{-I})}\,(\to\text{-El})$$

Example 2, Using Rules

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

$$
\dfrac{aaa{:}(A\to A)\to A \Rightarrow aaa{:}(A\to A)\to A \qquad \dfrac{aaa{:}(A\to A)\to A, a{:}A \Rightarrow a{:}A}{aaa{:}(A\to A)\to A \Rightarrow \lambda a^{A}.a{:}A\to A}\,(\to\text{-I})}{\dfrac{aaa{:}(A\to A)\to A \Rightarrow aaa\ (\lambda a^{A}.a){:}A}{(\lambda aaa^{(A\to A)\to A}.aaa)\ (\lambda a^{A}.a){:}((A\to A)\to A)\to A}\,(\to\text{-I})}\,(\to\text{-El})
$$

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

Example 3

- We derive an element of type

$$A \to B \to A \times B$$

(See **exampleProductIntro.agda**).

Example 3 (Cont.)

- **Step 1:**
  - We postulate types A, B:

$$\begin{array}{l} \mathrm{postulate}\ A : \mathrm{Set} \\ \mathrm{postulate}\ B : \mathrm{Set} \end{array}$$

  - We introduce the product type:

$$\begin{array}{l} \mathrm{record}\ \_\times\_\ (A\ B : \mathrm{Set}) : \mathrm{Set}\ \mathrm{where} \\ \quad \mathrm{field} \\ \qquad \begin{array}{lcl} \mathrm{first} & : & A \\ \mathrm{second} & : & B \end{array} \end{array}$$

Example 3 (Cont.)

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

$$f : A \longrightarrow B \longrightarrow A \times B$$
$$f = \{! \quad !\}$$

Example 3 (Cont.)

- **Step 3:**
  - An element of $A \rightarrow B \rightarrow A \times B$ will be of the form

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

  - We insert this into our goal and use **Refine** and obtain

$$\begin{aligned} f &: A \rightarrow B \rightarrow A \times B \\ f &= \ \lambda(a : A) \rightarrow \lambda(b : B) \rightarrow \{! \quad !\} \end{aligned}$$

Example 3 (Cont.)

- **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

$$\begin{array}{rcl} \mathrm{record}\ \{\mathrm{first} & = & \{!\ \ !\}; \\ \mathrm{second} & = & \{!\ \ !\}\} \end{array}$$

Example 3 (Cont.)

- **Step 4 (Cont):**
  - We insert this into the goal and obtain:

$$
\begin{aligned}
&f\colon A \longrightarrow B \longrightarrow A \times B \\
&\quad = \ \lambda(a : A) \longrightarrow \lambda(b : B) \longrightarrow \mathrm{record}\ \{\begin{array}{rcl} \mathrm{first} & = & \{!\ \ !\}; \\ \mathrm{second} & = & \{!\ \ !\}\} \end{array}
\end{aligned}
$$

Example 3 (Cont.)

- **Step 5:**
  - The first goal has as context:
    - $a : A$,
    - $b : B$
  - We could use as well
    - $A, B : \mathrm{Set}$,
    - $A \times B : \mathrm{Set}$,
    - $f : A \to B \to A \times B$.

Example 3 (Cont.)

- **Step 5 (Cont)**
  - We insert a, use refine and solve the first goal:

$$
\begin{array}{l}
f : A \rightarrow B \rightarrow A \times B \\
f = \ \lambda(a : A) \rightarrow \lambda(b : B) \rightarrow \mathrm{record}\ \{\begin{array}[t]{rcl}\mathrm{first} & = & a; \\ \mathrm{second} & = & \{!\ \ !\}\}\end{array}
\end{array}
$$

Example 3 (Cont.)

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

$$
\begin{array}{l}
f : A \to B \to A \times B \\
f = \ \lambda(a : A) \to \lambda(b : B) \to \mathrm{record}\ \{\begin{array}{lll}\mathrm{first} & = & a; \\ \mathrm{second} & = & b\}\end{array}
\end{array}
$$

Example 3, Using Rules

- $A \times B$ is formed as follows (assuming the global assumptions $A : \mathrm{Set}$, $B : \mathrm{Set}$):

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set}}{A \times B : \mathrm{Set}}\,(\times\text{-F})$$

- We won’t use this however, since it is required for the assumption rules only, the treatment of which will be delayed until later.

Example 3, Using Rules (Cont.)

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

$$d_0 : A \rightarrow B \rightarrow (A \times B)$$

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

$$\dfrac{\dfrac{a : A, b : B \Rightarrow d_1 : A \times B}{a : A \Rightarrow \lambda b^B.d_1 : B \rightarrow (A \times B)} (\rightarrow \text{-I})}{\lambda a^A.\lambda b^B.d_1 : A \rightarrow B \rightarrow (A \times B)} (\rightarrow \text{-I})$$

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:

$$
\dfrac{\dfrac{\dfrac{a : A, b : B \Rightarrow d_2 : A \qquad a : A, b : B \Rightarrow d_3 : B}{a : A, b : B \Rightarrow \langle d_2, d_3 \rangle : A \times B}\,(\times\text{-I})}{a : A \Rightarrow \lambda b^B.\langle d_2, d_3 \rangle : B \to (A \times B)}\,(\to\text{-I})}{\lambda a^A.\lambda b^B.\langle d_2, d_3 \rangle : A \to B \to (A \times B)}\,(\to\text{-I})
$$

Example 3, Using Rules (Cont.)

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

$$\dfrac{\dfrac{\dfrac{a : A, b : B \Rightarrow a : A \qquad a : A, b : B \Rightarrow b : B}{a : A, b : B \Rightarrow \langle a, b \rangle : A \times B}\,(\times\text{-I})}{a : A \Rightarrow \lambda b^B.\langle a, b \rangle : B \rightarrow (A \times B)}\,(\rightarrow\text{-I})}{\lambda a^A.\lambda b : B.\langle a, b \rangle : A \rightarrow B \rightarrow (A \times B)}\,(\rightarrow\text{-I})$$

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

Example 4

- We derive an element of type

$$(A \rightarrow B \times C) \rightarrow A \rightarrow B$$

(See **exampleProductElim.agda**).

Example 4 (Cont.)

- **Step 1:**
  - We postulate types A, B, C:

$$\begin{array}{l} \mathrm{postulate}\ A : \mathrm{Set} \\ \mathrm{postulate}\ B : \mathrm{Set} \\ \mathrm{postulate}\ C : \mathrm{Set} \end{array}$$

  - The product is introduced as before:

$$\begin{array}{l} \mathrm{record}\ \_\times\_\ (A\ B : \mathrm{Set}) : \mathrm{Set}\ \mathrm{where} \\ \quad \mathrm{field} \\ \qquad \begin{array}{lcl} \mathrm{first} & : & A \\ \mathrm{second} & : & B \end{array} \end{array}$$

Example 4 (Cont.)

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

$$
\begin{aligned}
&f : (A \rightarrow B \times C) \rightarrow A \rightarrow B \\
&f = \{!\ \ !\}
\end{aligned}
$$

Example 4 (Cont.)

- **Step 3:**
  - We insert a λ-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:

$$
\begin{array}{l}
f : (A \to B \times C) \to A \to B \\
f = \lambda(a\text{–}bc : A \to B \times C) \\
\qquad \to \lambda(a : A) \\
\qquad \to \mathrm{let}\ \begin{array}{lcl} bc & : & B \times C \\ bc & = & \{!\ \ !\} \end{array} \\
\qquad\quad\ \mathrm{in}\ \{!\ \ !\}
\end{array}
$$

Example 4 (Cont.)

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

$$
\begin{array}{l}
f : (A \to B \times C) \to A \to B \\
f = \lambda(a{-}bc : A \to B \times C) \\
\quad\quad \to \lambda(a : A) \\
\quad\quad \to \mathrm{let}\ \begin{array}{rcl} bc & : & B \times C \\ bc & = & a{-}bc\ \{!\ \ !\} \end{array} \\
\quad\quad\quad \mathrm{in}\ \{!\ \ !\}
\end{array}
$$

Example 4 (Cont.)

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

$$
\begin{aligned}
&f : (A \to B \times C) \to A \to B \\
&f = \lambda(a\text{–}bc : A \to B \times C) \\
&\qquad \to \lambda(a : A) \\
&\qquad \to \mathrm{let}\ \begin{array}{rcl} bc & : & B \times C \\ bc & = & a\text{–}bc\ a \end{array} \\
&\qquad\quad \mathrm{in}\ \{!\ \ !\}
\end{aligned}
$$

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 $\_\times\_.\mathrm{first}$.
  - We obtain

$$
\begin{array}{l}
f : (A \rightarrow B \times C) \rightarrow A \rightarrow B \\
f = \lambda(a\text{--}bc : A \rightarrow B \times C) \\
\qquad \rightarrow \lambda(a : A) \\
\qquad \rightarrow \mathrm{let}\ \begin{array}{rcl} bc & : & B \times C \\ bc & = & a\text{--}bc\ a \end{array} \\
\qquad\quad \mathrm{in}\ \_\times\_.\mathrm{first}\ bc
\end{array}
$$

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 ($a\text{--}bc\ a$).
- We get

$$
\begin{aligned}
& f : (A \to B \times C) \to A \to B \\
& f = \lambda(a\text{--}bc : A \to B \times C) \\
& \qquad \to \lambda(a : A) \\
& \qquad \to \_\times\_.\mathrm{first}\ (a\text{--}bc\ a)
\end{aligned}
$$

Example 4, Using Rules

- Using rules we make the global assumptions $A : \mathrm{Set}, B : \mathrm{Set}, C : \mathrm{Set}$.
- Then we start with our goal

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

Example 4, Using Rules (Cont.)

- The use of a λ-expression amounts to replacing d_0 by

$$\lambda a\!-\!bc^{A\rightarrow(B\times C)}.\lambda a^{A}.d_1$$

introduced by two applications of an introduction rule:

$$\dfrac{\dfrac{a\!-\!bc : A \longrightarrow (B \times C), a : A \Rightarrow d_1 : A}{a\!-\!bc : A \longrightarrow (B \times C) \Rightarrow \lambda a^{A}.d_1 : A \longrightarrow B}\,(\longrightarrow\text{-I})}{\lambda a\!-\!bc^{A\rightarrow(B\times C)}.\lambda a^{A}.d_1 : (A \longrightarrow (B \times C)) \longrightarrow A \longrightarrow B}\,(\longrightarrow\text{-I})$$

Example 4, Using Rules (Cont.)

- In Agda, we then replace the goal corresponding to d_1 by _×_.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:

$$
\dfrac{\dfrac{\dfrac{\dfrac{a\!-\!bc{:}A\to(B\times C),a{:}A\Rightarrow a\!-\!bc{:}A\to(B\times C) \qquad a\!-\!bc{:}A\to(B\times C),a{:}A\Rightarrow a{:}A}{a\!-\!bc{:}A\to(B\times C),a{:}A\Rightarrow a\!-\!bc\ a{:}B\times C}\,(\to\text{-E}}{a\!-\!bc{:}A\to(B\times C),a{:}A\Rightarrow \pi_0(a\!-\!bc\ a){:}B}\,(\times\text{-El})}{a\!-\!bc{:}A\to(B\times C)\Rightarrow \lambda a^{A}.\pi_0(a\!-\!bc\ a){:}A\to B}\,(\to\text{-I})}{\lambda a\!-\!bc^{A\to(B\times C)}.\lambda a^{A}.\pi_0(a\!-\!bc\ a){:}(A\to(B\times C))\to A\to B}\,(\to\text{-I})
$$

- The two initial judgements can be introduced by assumption rules.

(h) Presuppositions

- In order to derive $x : A, y : B \Rightarrow C : \mathrm{Set}$ we need to show:
  - $A : \mathrm{Set}$.
  - $x : A \Rightarrow B : \mathrm{Set}$
- So the judgement

$$x : A, y : B \Rightarrow C : \mathrm{Set}$$

implicitly contains the judgements

$$A : \mathrm{Set} \quad ,$$

$$x : A \Rightarrow B : \mathrm{Set} \quad .$$

Presuppositions (Cont.)

- $A : \mathrm{Set}$ and $x : A \Rightarrow B : \mathrm{Set}$ are **presuppositions** of the judgement

$$x : A, y : B \Rightarrow C : \mathrm{Set} \quad .$$

Presuppositions (Cont.)

- $A : \mathrm{Set}$ and $B : \mathrm{Set}$ are **presuppositions** of the judgement
$$A \to B : \mathrm{Set} \quad .$$
and of the judgement
$$A \times B : \mathrm{Set} \quad .$$
- The next slide shows the presuppositions of judgements.

Presuppositions

Judgement	Presuppositions
$\Gamma, x : A \Rightarrow \mathrm{Context}$	$\Gamma \Rightarrow A : \mathrm{Set}$.
$\Gamma \Rightarrow A : \mathrm{Set}$	$\Gamma \Rightarrow \mathrm{Context}$
$\Gamma \Rightarrow A = B : \mathrm{Set}$	$\Gamma \Rightarrow A : \mathrm{Set}$, $\Gamma \Rightarrow B : \mathrm{Set}$.

Presuppositions

Judgement	Presuppositions
$\Gamma \Rightarrow a : A$	$\Gamma \Rightarrow A : \mathrm{Set}$.
$\Gamma \Rightarrow a = b : A$	$\Gamma \Rightarrow a : A$, $\Gamma \Rightarrow b : A$.

Presuppositions

Judgement	**Presuppositions**
$\Gamma \Rightarrow (x : A) \to B : \mathrm{Set}$	$\Gamma, x : A \Rightarrow B : \mathrm{Set}$.
$\Gamma \Rightarrow (x : A) \times B : \mathrm{Set}$	$\Gamma, x : A \Rightarrow B : \mathrm{Set}$.

Presuppositions

- Furthermore, **presuppositions of presuppositions** of

$$\Gamma \Rightarrow \theta$$

are as well presuppositions of

$$\Gamma \Rightarrow \theta \quad .$$

Example of Presuppositions

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

Remark on $\mathrm{A} \to \mathrm{B}$, $\mathrm{A} \times \mathrm{B}$

- Note that $A \to B$ is an **abbreviation** for $(x : A) \to B$ for some fresh x.
- Similarly $A \times B$ is an **abbreviation** for $(x : A) \times B$ for some fresh x.
- Therefore the presupposition of $A \to B : \mathrm{Set}$ (which abbreviates $\emptyset \Rightarrow A \to B : \mathrm{Set}$) are:
  - $\emptyset \Rightarrow \mathrm{Context}$,
  - $A : \mathrm{Set}$,
  - $x : A \Rightarrow \mathrm{Context}$,
  - $x : A \Rightarrow B : \mathrm{Set}$.

(i) The Full Logical Framework

- We would like to **add operations on types**, such as

$$\mathrm{prod} : \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set}$$

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

- The problem is that for this we need

$$\mathrm{Set} \to \mathrm{Set} \to \mathrm{Set} : \mathrm{Set}$$

and our rules allow this only if we had

Set: Set

Set

- Adding

$$\mathrm{Set} : \mathrm{Set}$$

  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 $A : \mathrm{Set}$ we have as well judgements of the form

$$A : \mathrm{Type}$$

  - One rule will especially express

$$\mathrm{Set} : \mathrm{Type}$$

  - Elements of Type are **types**, elements of Set are **small types**.

Set (Cont.)

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

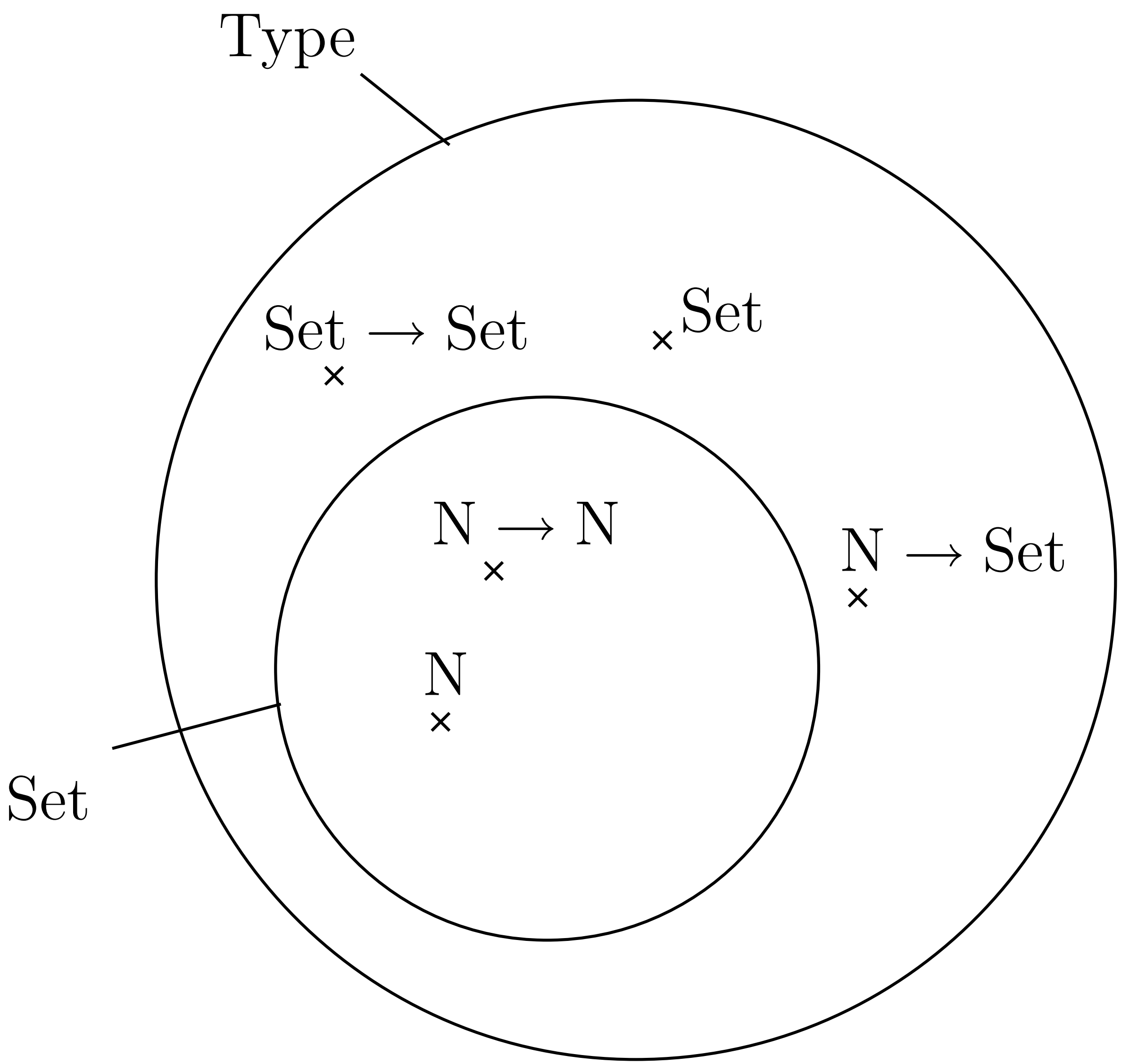
$$\mathrm{Set} \rightarrow \mathrm{Set} \rightarrow \mathrm{Set} : \mathrm{Type}$$

and we can **assign to prod above the type**

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

(The definition of prod will be given later.)

Set and Type



Set (Cont.)

- However, we **cannot use prod in order to form the product of two sets**, ie. we cannot introduce

$$\mathrm{prod}\ \mathrm{Set}\ \mathrm{Set} : \mathrm{Set} \quad ,$$

since $\mathrm{Set} : \mathrm{Set}$ does not hold.

Rules for Set (as an El. of Type)

Formation Rule for Set

$$\mathrm{Set} : \mathrm{Type} \qquad (\mathrm{SetIsType})$$

Every Set is a Type

$$\frac{A : \mathrm{Set}}{A : \mathrm{Type}} \, (\mathrm{Set2Type})$$

Closure of Type

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

Closure of Type under the dependent product

$$\frac{A : \mathrm{Type} \qquad x : A \Rightarrow B : \mathrm{Type}}{(x : A) \times B : \mathrm{Type}} \, (\times\text{-}\mathrm{F}^{\mathrm{Type}})$$

Closure of Type under the dependent function type

$$\frac{A : \mathrm{Type} \qquad x : A \Rightarrow B : \mathrm{Type}}{(x : A) \to B : \mathrm{Type}} \, (\to\text{-}\mathrm{F}^{\mathrm{Type}})$$

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

$$\frac{A : \mathrm{Type} \qquad B : \mathrm{Type}}{A \times B : \mathrm{Type}} \, (\times\text{-F}^{\mathrm{Type}})$$

Closure of Type under the non-dependent function type

$$\frac{A : \mathrm{Type} \qquad B : \mathrm{Type}}{A \to B : \mathrm{Type}} \, (\to\text{-F}^{\mathrm{Type}})$$

Equality Versions of the Rules

Formation Rule for Set

$$\mathrm{Set} = \mathrm{Set} : \mathrm{Type} \qquad (\mathrm{SetIsType}^{=})$$

Every Set is a Type

$$\frac{A = B : \mathrm{Set}}{A = B : \mathrm{Type}} \, (\mathrm{Set2Type}^{=})$$

Equality Versions of the Rules

Closure of Type under the dependent product

$$\frac{A = A' : \mathrm{Type} \qquad x : A \Rightarrow B = B' : \mathrm{Type}}{(x : A) \times B = (x : A') \times B' : \mathrm{Type}} \, (\times\text{-}\mathrm{F}^{=,\mathrm{Type}})$$

Closure of Type under the dependent function type

$$\frac{A = A' : \mathrm{Type} \qquad x : A \Rightarrow B = B' : \mathrm{Type}}{(x : A) \to B = (x : A') \to B' : \mathrm{Type}} \, (\to\text{-}\mathrm{F}^{=,\mathrm{Type}})$$

Similarly for the non-dependent versions of the above.

Definition of prod

- Now $\mathrm{Set} \to \mathrm{Set} \to \mathrm{Set} : \mathrm{Type}$.
- And we can derive

$$\begin{array}{rcl} \mathrm{prod} & := & \lambda(X, Y : \mathrm{Set}).X \times Y \\ & : & \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set} \end{array}$$

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

Context Rules

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

$$\frac{\Gamma \Rightarrow A : \mathrm{Type}}{\Gamma, x : A \Rightarrow \mathrm{Context}}\,(\mathrm{Context}_1^{\mathrm{Type}})$$

Example: prod

We can now introduce $\mathbf{prod} : \mathbf{Set} \to \mathbf{Set} \to \mathbf{Set}$:
First we derive $X : \mathrm{Set}, Y : \mathrm{Set} \Rightarrow X : \mathrm{Set}$:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{Set} : \mathrm{Type}}{X : \mathrm{Set} \Rightarrow \mathrm{Context}}\,(\mathrm{Context}_1)}{X : \mathrm{Set} \Rightarrow \mathrm{Set} : \mathrm{Type}}\,(\mathrm{SetIsType})}{X : \mathrm{Set}, Y : \mathrm{Set} \Rightarrow \mathrm{Context}}\,(\mathrm{Context}_1)}{X : \mathrm{Set}, Y : \mathrm{Set} \Rightarrow X : \mathrm{Set}}\,(\mathrm{Ass})
$$

Similarly we derive $X : \mathrm{Set}, Y : \mathrm{Set} \Rightarrow Y : \mathrm{Set}$.

Example: prod (Cont.)

Now we can derive our desired judgement:

$$\dfrac{\dfrac{\dfrac{X : \mathrm{Set}, Y : \mathrm{Set} \Rightarrow X : \mathrm{Set} \qquad X : \mathrm{Set}, Y : \mathrm{Set} \Rightarrow Y : \mathrm{Set}}{X : \mathrm{Set}, Y : \mathrm{Set} \Rightarrow X \times Y : \mathrm{Set}}\,(\times\text{-F})}{X : \mathrm{Set} \Rightarrow \lambda Y^{\mathrm{Set}}.X \times Y : \mathrm{Set} \rightarrow \mathrm{Set}}\,(\rightarrow\text{-I})}{\lambda(X, Y : \mathrm{Set}).X \times Y : \mathrm{Set} \rightarrow \mathrm{Set} \rightarrow \mathrm{Set}}\,(\rightarrow\text{-I})$$

and define

$$\begin{array}{rcl} \mathrm{prod} & := & \lambda(X, Y : \mathrm{Set}).X \times Y \\ & : & \mathrm{Set} \rightarrow \mathrm{Set} \rightarrow \mathrm{Set} \end{array}$$

Set vs. Type in Agda

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

Hierarchies of Types

- If one wants to form

$$\mathrm{prod}' : \mathrm{Type} \to \mathrm{Type} \to \mathrm{Type} \quad ,$$

one needs to have a further level Kind above Type, s.t.

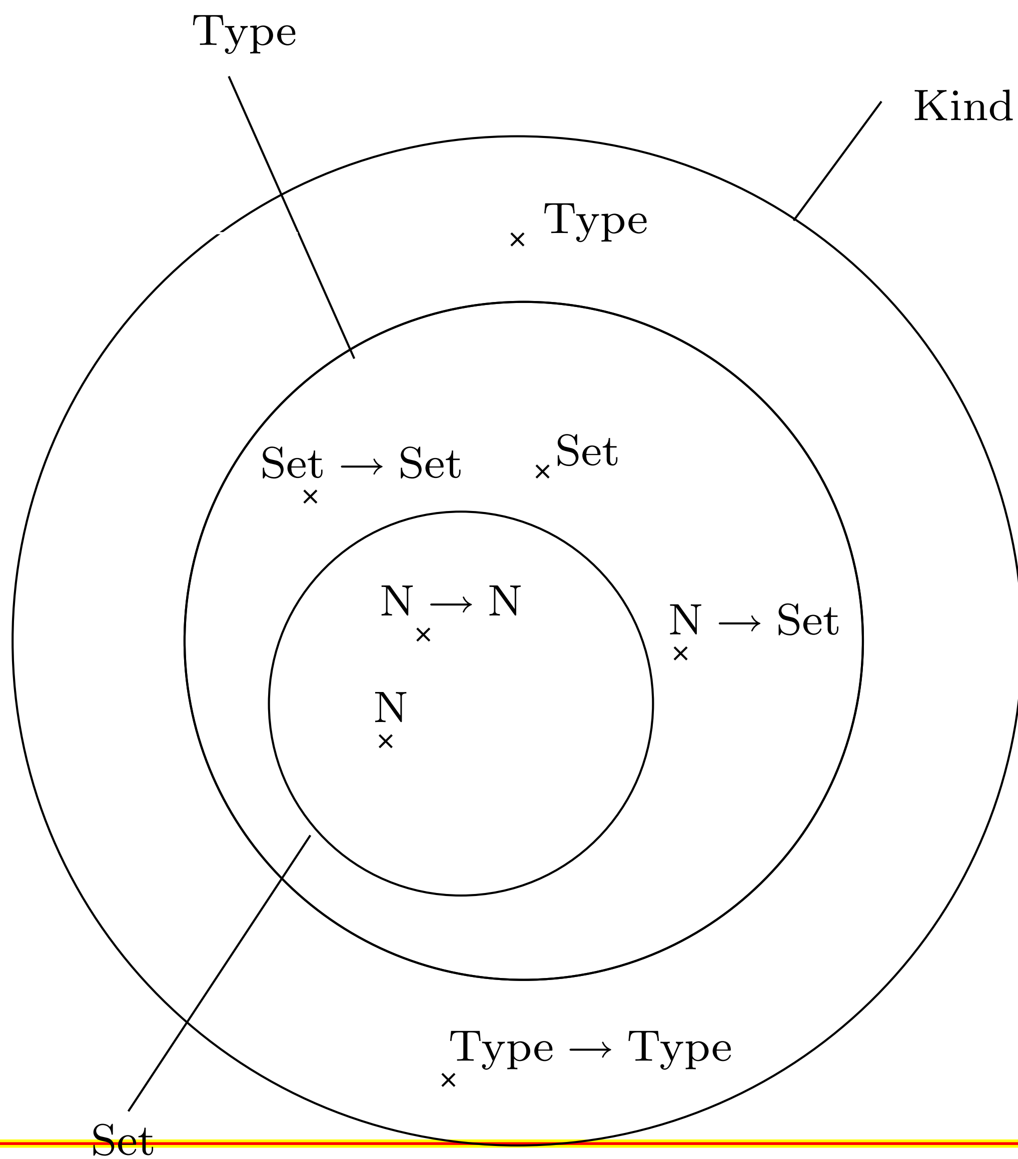
$$\mathrm{Type} : \mathrm{Kind} \quad .$$

  - Then

$$\mathrm{Type} \to \mathrm{Type} \to \mathrm{Type} : \mathrm{Kind} \quad .$$

  - In Agda Kind is written as $\mathrm{Set2}$.

Hierarchy of Types (Set, Type, Kind)



Rules for Type as a Kind

Type is a Kind

$$\mathrm{Type} : \mathrm{Kind}$$

Every Type is a Kind

$$\frac{A : \mathrm{Type}}{A : \mathrm{Kind}}\,(\mathrm{Type2Kind})$$

Closure of Kind

Closure of Kind under the dependent product

$$\frac{A : \mathrm{Kind} \qquad x : A \Rightarrow B : \mathrm{Kind}}{(x : A) \times B : \mathrm{Kind}} \ (\times\text{-F}^{\mathrm{Kind}})$$

Closure of Kind under the dependent function type

$$\frac{A : \mathrm{Kind} \qquad x : A \Rightarrow B : \mathrm{Kind}}{(x : A) \to B : \mathrm{Kind}} \ (\to\text{-F}^{\mathrm{Kind}})$$

Plus **equality versions** of the above rules.

Jump over Context Rule.

Context Rules

- Again, the context rules have to be expanded:

$$\frac{\Gamma \Rightarrow A : \mathrm{Kind}}{\Gamma, x : A \Rightarrow \mathrm{Context}} \, (\mathrm{Context}_1^{\mathrm{Kind}})$$

Definition of prod'

- Now we can define

$$\begin{array}{rcl} \mathrm{prod}' & := & \lambda(X, Y : \mathrm{Type}).X \times Y \\ & : & \mathrm{Type} \to \mathrm{Type} \to \mathrm{Type} \end{array}$$

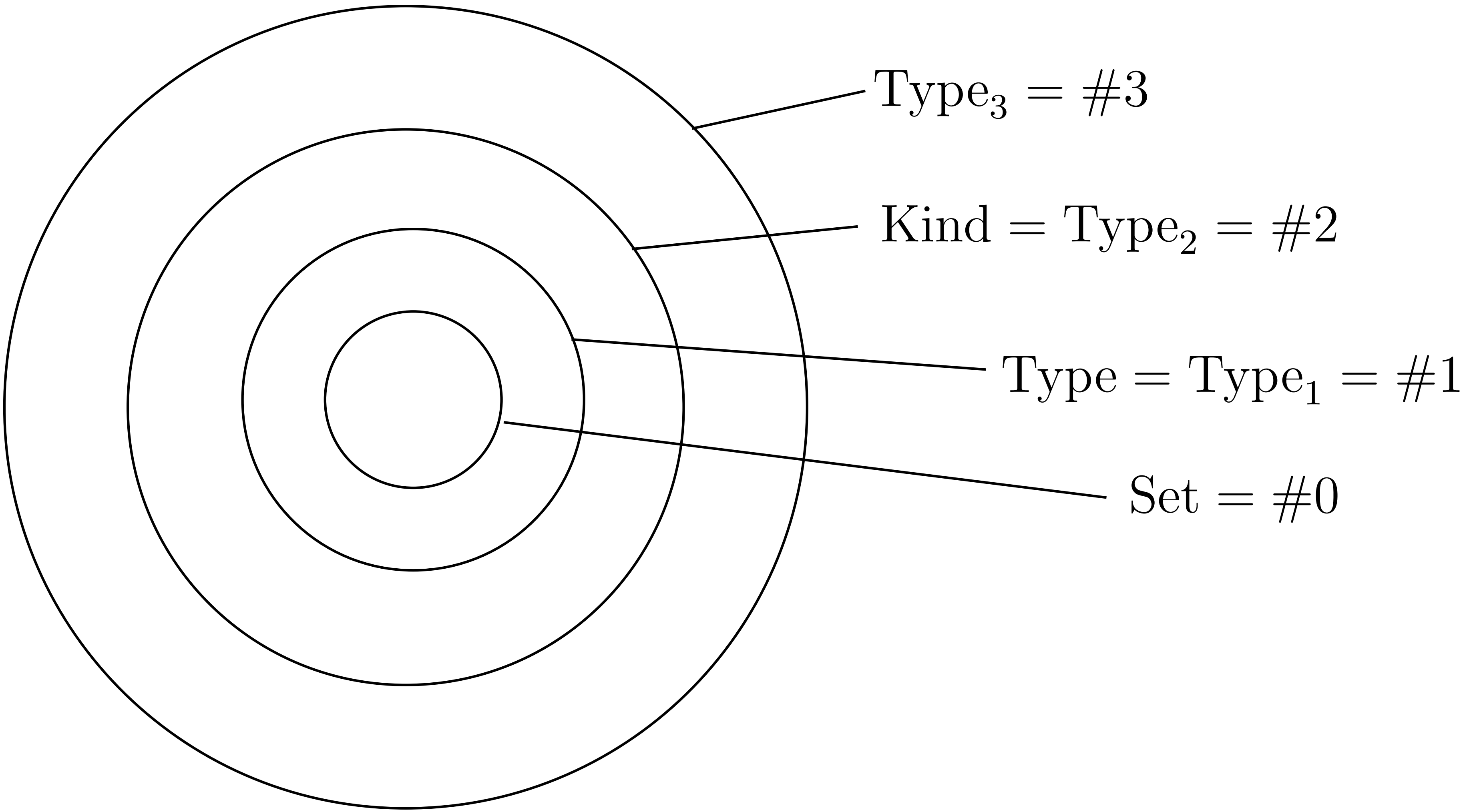
Hierarchies of Types (Cont.)

- This can be iterated further, forming
  $\mathbf{Type} = \mathbf{Type_1}, \mathbf{Kind} = \mathbf{Type_2}, \mathbf{Type_3}, \mathbf{Type_4} \cdots$
- So we have
  - $\mathrm{Set} : \mathrm{Type}$,
  - $\mathrm{Set} : \mathrm{Type}_2$, $\mathrm{Type} = \mathrm{Type}_1 : \mathrm{Type}_2$,
  - $\mathrm{Set} : \mathrm{Type}_3$, $\mathrm{Type} = \mathrm{Type}_1 : \mathrm{Type}_3$, $\mathrm{Type}_2 : \mathrm{Type}_3$,
  - $\mathrm{Set} : \mathrm{Type}_4$, $\mathrm{Type} = \mathrm{Type}_1 : \mathrm{Type}_4$, $\mathrm{Type}_2 : \mathrm{Type}_4$, $\mathrm{Type}_3 : \mathrm{Type}_4$,
  - etc.

Hierarchies of Types (Cont.)

- Agda has a hierarchy of types built in, written as $\mathrm{Set0}$ (which is Set), $\mathrm{Set1}$ (which is Type), $\mathrm{Set2}$ (in the rule calculus called Kind), $\mathrm{Set3}$ etc.
- Again we don't have for instance $\mathrm{Set} : \mathrm{Set}_2$.
- But $(x : A) \to B$, $(x : A) \times B$ belong to the maximum type level of A and B.

Hierarchy of Types (Set0, Set1, Set2



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 $A : \mathrm{Set}$ but $A : \mathrm{Type}$.
  - It might be that the derivation derives actually $A : \mathrm{Set}$, but that implies $A : \mathrm{Type}$.
  - But it might be that we can only derive $A : \mathrm{Type}$.
- Therefore the presuppositions have to be changed as in the following table.

Presuppositions (with Set, Type)

Judgement	Presuppositions
$\Gamma, x : A \Rightarrow \mathrm{Context}$	$\Gamma \Rightarrow A : \mathrm{Type}$.
$\Gamma \Rightarrow A : \mathrm{Set}$	$\Gamma \Rightarrow A : \mathrm{Type}$.
$\Gamma \Rightarrow A : \mathrm{Type}$	$\Gamma \Rightarrow \mathrm{Context}$.

Presuppositions (with Set, Type)

Judgement	Presuppositions
$\Gamma \Rightarrow A = B : \mathrm{Set}$	$\Gamma \Rightarrow A : \mathrm{Set}$, $\Gamma \Rightarrow B : \mathrm{Set}$, $\Gamma \Rightarrow A = B : \mathrm{Type}$.
$\Gamma \Rightarrow A = B : \mathrm{Type}$	$\Gamma \Rightarrow A : \mathrm{Type}$, $\Gamma \Rightarrow B : \mathrm{Type}$.
$\Gamma \Rightarrow a : A$	$\Gamma \Rightarrow A : \mathrm{Type}$.

Presuppositions (with Set, Type)

Judgement	Presuppositions
$\Gamma \Rightarrow a = b : A$	$\Gamma \Rightarrow a : A$, $\Gamma \Rightarrow b : A$.
$\Gamma \Rightarrow (x : A) \times B : \mathrm{Set}$	$\Gamma \Rightarrow A : \mathrm{Set}$, $\Gamma, x : A \Rightarrow B : \mathrm{Set}$.
$\Gamma \Rightarrow (x : A) \times B : \mathrm{Type}$	$\Gamma, x : A \Rightarrow B : \mathrm{Type}$.

Presuppositions (with Set, Type)

Judgement	**Presuppositions**
$\Gamma \Rightarrow (x : A) \to B : \mathrm{Set}$	$\Gamma \Rightarrow A : \mathrm{Set}$, $\Gamma, x : A \Rightarrow B : \mathrm{Set}$.
$\Gamma \Rightarrow (x : A) \to B : \mathrm{Type}$	$\Gamma, x : A \Rightarrow B : \mathrm{Type}$.

Changes To Presuppositions

- If we have more levels (Kind or $\mathrm{Set}i$), then the presuppositions have to be changed again.
  - E.g., if we have levels Set, Type, Kind, the presupposition
    - of $\Gamma \Rightarrow A : \mathrm{Set}$ is $\Gamma \Rightarrow A : \mathrm{Type}$,
    - of $\Gamma \Rightarrow A : \mathrm{Type}$ is $\Gamma \Rightarrow A : \mathrm{Kind}$,
    - of $\Gamma \Rightarrow A : \mathrm{Kind}$ is $\Gamma \Rightarrow \mathrm{Context}$.