CS_336/CS_M36 (part 2)/CS_M46 Interactive Theorem Proving

Course Notes
Lent Term Term 2008
Sect. 6 Data Types

Anton Setzer
Dept. of Computer Science, Swansea University

http://www.cs.swan.ac.uk/~csetzer/lectures/intertheo/07/index.html

May 24, 2017

6 (a) The Set of Booleans

Formation Rule

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

Introduction Rules

$$\mathrm{tt} : \mathrm{Bool} \quad (\mathrm{Bool\text{-}I_{tt}}) \qquad\qquad \mathrm{ff} : \mathrm{Bool} \quad (\mathrm{Bool\text{-}I_{ff}})$$

Elimination Rule

$$\frac{C : \mathrm{Bool} \to \mathrm{Set} \quad \mathit{case}_{\mathrm{tt}} : C\ \mathrm{tt} \quad \mathit{case}_{\mathrm{ff}} : C\ \mathrm{ff} \quad b : \mathrm{Bool}}{\mathrm{Case}_{\mathrm{Bool}}\ C\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ b : C\ b}\ (\mathrm{Bool\text{-}El})$$

The Set of Booleans (Cont.)

Equality Rules

$$\frac{C : \mathrm{Bool} \to \mathrm{Set} \qquad \mathit{case}_{\mathrm{tt}} : C\ \mathrm{tt} \qquad \mathit{case}_{\mathrm{ff}} : C\ \mathrm{ff}}{\mathrm{Case}_{\mathrm{Bool}}\ C\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ \mathrm{tt} = \mathit{case}_{\mathrm{tt}} : C\ \mathrm{tt}}\ (\mathrm{Bool\text{-}Eq}_{\mathrm{tt}})$$

$$\frac{C : \mathrm{Bool} \to \mathrm{Set} \qquad \mathit{case}_{\mathrm{tt}} : C\ \mathrm{tt} \qquad \mathit{case}_{\mathrm{ff}} : C\ \mathrm{ff}}{\mathrm{Case}_{\mathrm{Bool}}\ C\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ \mathrm{ff} = \mathit{case}_{\mathrm{ff}} : C\ \mathrm{ff}}\ (\mathrm{Bool\text{-}Eq}_{\mathrm{ff}})$$

Further we have equality versions of the formation-, introduction- and elimination-rules.

Remarks

- $\mathrm{Case}_{\mathrm{Bool}}\ C\ \ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ b$ can be read as

  $$\mathrm{if}\ b\ \mathrm{then}\ \mathit{case}_{\mathrm{tt}}\ \mathrm{else}\ \mathit{case}_{\mathrm{ff}}$$

  where the additional argument C is required in order to determine the type of $\mathit{case}_{\mathrm{tt}}$, of $\mathit{case}_{\mathrm{ff}}$, and of the result of this construct.

Remarks (Cont.)

- The argument $C : \mathrm{Bool} \to \mathrm{Set}$ denotes the set into which we are eliminating.
  - Instead of $C : \mathrm{Set}$, we demand $C : \mathrm{Bool} \to \mathrm{Set}$, since the set into which we are eliminating might depend on the Boolean valued argument.
  - That is necessary in order to define functions $f : (b : \mathrm{Bool}) \to D$ where D depends on b.

Remarks (Cont.)

- If we define

$$\begin{array}{lcl} C & := & \lambda b^{\mathrm{Bool}}.D \\ & : & \mathrm{Bool} \to \mathrm{Set} \\ f & := & \lambda b^{\mathrm{Bool}}.\mathrm{Case}_{\mathrm{Bool}}\ C\ case_{\mathrm{tt}}\ case_{\mathrm{ff}}\ b \\ & : & (b : \mathrm{Bool}) \to C\ b \end{array}$$

where

$$(b : \mathrm{Bool}) \to C\ b = (b : \mathrm{Bool}) \to D$$

we have:

- $f\ \mathrm{tt} : C\ \mathrm{tt}$.
- $f\ \mathrm{ff} : C\ \mathrm{ff}$.
- $f : (b : \mathrm{Bool}) \to C\ b$.

Remarks (Cont.)

- The argument C above has no computational content.
  - It is not needed in order to compute $\mathrm{Case}_{\mathrm{Bool}}\ C\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ \mathrm{tt}$ and $\mathrm{Case}_{\mathrm{Bool}}\ C\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ \mathrm{ff}$.
- C is only needed in order to obtain decidable type checking:
  - In the presence of arguments like this we can decide whether a judgement $a : B$ is derivable.

Remarks (Cont.)

- We can write the elimination rule in a **more compact** but less readable way:
  - $$\begin{aligned}\mathrm{Case}_{\mathrm{Bool}} : (C : \mathrm{Bool} \to \mathrm{Set}) \\ \to (case_{\mathrm{tt}} : C\ \mathrm{tt}) \\ \to (case_{\mathrm{ff}} : C\ \mathrm{ff}) \\ \to (b : \mathrm{Bool}) \\ \to C\ b\end{aligned}$$.
- tt, ff are the **constructors** of Bool.

Remarks (Cont.)

- Notice that we then get for $C : \mathrm{Bool} \to \mathrm{Set}$, $\mathit{case}_{\mathrm{tt}} : C\ \mathrm{tt}, \mathit{case}_{\mathrm{ff}} : C\ \mathrm{ff}$
  - $f := \mathrm{Case}_{\mathrm{Bool}}\ C\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}$,
    $: (b : \mathrm{Bool}) \to C\ b$
  - $f\ \mathrm{tt} = \mathrm{Case}_{\mathrm{Bool}}\ C\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ \mathrm{tt} = \mathit{case}_{\mathrm{tt}} : C\ \mathrm{tt}$,
  - $f\ \mathrm{ff} = \mathrm{Case}_{\mathrm{Bool}}\ C\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ \mathrm{ff} = \mathit{case}_{\mathrm{ff}} : C\ \mathrm{ff}$.
- So we obtain functions from Bool into other sets **without having to write $\lambda \mathbf{b}^{\mathbf{Bool}}. \cdots$**.
- That's why we choose the argument to eliminate from as the **last one**.

Remarks (Cont.)

- This is similar to the definition of for instance $(+)$ in **curried form** in Haskell
  - $(+) : \mathrm{int} \to \mathrm{int} \to \mathrm{int}$.
  - $(+)$ 3 is the function which takes an integer and adds to it 3.
    - **Shorter** than writing $\lambda x^{\mathrm{int}}.3 + x$.

Remarks (Cont.)

- Note that we have the following **order of the arguments** of $\mathrm{Case}_{\mathrm{Bool}}$:
  - First we have the **set into which we eliminate**.
  - Then follow the **cases**, one for each constructor.
  - Finally we put the **element which we are eliminating**.
- In some sense $\mathrm{Case}_{\mathrm{Bool}}$ is a "then _else _if " – the **condition** (if . . .) **is the last one**.

Select Example

- Assume we have introduced in type theory

$$
\begin{array}{lcl}
\mathrm{Name} & : & \mathrm{Bool} \to \mathrm{Set} \quad , \\
\mathrm{Name}\ \mathrm{tt} & = & \mathrm{FemaleName} \quad , \\
\mathrm{Name}\ \mathrm{ff} & = & \mathrm{MaleName} \quad .
\end{array}
$$

Select Example

- Then we can define the function

$$\begin{array}{rcl} \mathrm{SelectBool} & : & (b : \mathrm{Bool}) \to \mathrm{Name}\ b \\ \mathrm{SelectBool}\ \mathrm{tt} & = & \mathrm{sara} \\ \mathrm{SelectBool}\ \mathrm{ff} & = & \mathrm{tom} \end{array}$$

as follows:

$$\mathrm{SelectBool} = \mathrm{Case}_{\mathrm{Bool}}\ \mathrm{Name}\ \mathrm{sara}\ \mathrm{tom}$$

  - Note that by using twice the η-rule we get that

$$\begin{array}{l} \mathrm{SelectBool} \\ \quad = \ \lambda b^{\mathrm{Bool}}.\mathrm{Case}_{\mathrm{Bool}}\ (\lambda d^{\mathrm{Bool}}.\mathrm{Name}\ d)\ \mathrm{sara}\ \mathrm{tom}\ b \end{array}$$

Select Example

- We verify the correctness of $\mathrm{SelectBool}$:

$$\begin{array}{rcl} \mathrm{SelectBool}\ \mathrm{tt} & = & \mathrm{Case}_{\mathrm{Bool}}\ \mathrm{Name}\ \mathrm{sara}\ \mathrm{tom}\ \mathrm{tt} = \mathrm{sara}\ \ , \\ \mathrm{SelectBool}\ \mathrm{ff} & = & \mathrm{Case}_{\mathrm{Bool}}\ \mathrm{Name}\ \mathrm{sara}\ \mathrm{tom}\ \mathrm{ff} = \mathrm{tom}\ \ . \end{array}$$

Jump over $\wedge_{\mathrm{Bool}}$

Example: $\wedge_{\mathrm{Bool}}$

- We want to introduce conjunction

$$\wedge_{\mathrm{Bool}} : \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool} \ .$$

- This will be of the form

$$\wedge_{\mathrm{Bool}} = \ \lambda(b, c : \mathrm{Bool}).t$$

  for some term t.

- t will be defined by case distinction on b, so we get

$$\wedge_{\mathrm{Bool}} = \ \lambda(b, c : \mathrm{Bool}).\mathrm{Case}_{\mathrm{Bool}} \ C \ e \ f \ b$$

  for some e, f.

Example: $\wedge_{\mathrm{Bool}}$

$\wedge_{\mathrm{Bool}} = \lambda(b, c : \mathrm{Bool}).\mathrm{Case}_{\mathrm{Bool}}\ C\ e\ f\ b$

- C will be the set into which we are eliminating, depending on a Boolean value.
  - It need to be an element of $\mathrm{Bool} \to \mathrm{Set}$.
  - Therefore we have $C = \lambda d^{\mathrm{Bool}}.D$ for some D which might depend on d.
  - The set, into which we are eliminating, is always the same, namely Bool.
  - So $D = \mathrm{Bool}$ and therefore we have

$$C = \lambda d^{\mathrm{Bool}}.\mathrm{Bool}\ .$$

Example: $\wedge_{\mathrm{Bool}}$

- Note that in
$$\lambda d^{\mathrm{Bool}}.\mathrm{Bool}$$
Bool occurs in two different meanings:
  - The first occurrence is that of a set.
    - d is chosen here as an element of that set.
  - The second occurrence is that as an element of another type, namely Set.
    - So here Bool is a term.

Two Meanings of Elements of Set

- All elements A of Set have these two meanings:
  - They can be used as terms, which are elements of the type Set.
    - The corresponding judgements are $A : \mathrm{Set}$, $A = A' : \mathrm{Set}$.
  - And they can be used as sets, which have elements.
    - The corresponding judgements are $a : A$ and $a = a' : A$.

Example: $\wedge_{\mathrm{Bool}}$

- So
$$\wedge_{\mathrm{Bool}} = \lambda(b, c : \mathrm{Bool}).\mathrm{Case}_{\mathrm{Bool}}\ (\lambda d^{\mathrm{Bool}}.\mathrm{Bool})\ e\ f\ b$$
for some e, f.
- For conjunction we have:
  - If b is true then
  $$b \wedge c = \mathrm{tt} \wedge c = c$$
    - So the if-case e above is c.
  - If c is false then
  $$b \wedge c = \mathrm{ff} \wedge c = \mathrm{ff}$$
    - So the else-case f above is ff.

Example: $\wedge_{\mathrm{Bool}}$

- In total we define therefore

$$\begin{array}{rcl} \wedge_{\mathrm{Bool}} & = & \lambda(b, c : \mathrm{Bool}).\mathrm{Case}_{\mathrm{Bool}}\ (\lambda d^{\mathrm{Bool}}.\mathrm{Bool})\ c\ \mathrm{ff}\ b \\ & : & \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool} \end{array}$$

- We verify the correctness of this definition:
  - $\wedge_{\mathrm{Bool}}\ \mathrm{tt}\ c = \mathrm{Case}_{\mathrm{Bool}}\ (\lambda d^{\mathrm{Bool}}.\mathrm{Bool})\ c\ \mathrm{ff}\ \mathrm{tt} = c$.
    as desired.
  - $\wedge_{\mathrm{Bool}}\ \mathrm{ff}\ c = \mathrm{Case}_{\mathrm{Bool}}\ (\lambda d^{\mathrm{Bool}}.\mathrm{Bool})\ c\ \mathrm{ff}\ \mathrm{ff} = \mathrm{ff}$.
    Correct as desired.

Jump over derivation of $\wedge_{\mathrm{Bool}}$

Derivation of $\wedge_{\mathrm{Bool}}$

- We derive in the following $\wedge_{\mathrm{Bool}}$: **Bool** $\to$ **Bool** $\to$ **Bool**.
- We write Bool, if it
  - is a type in **boldface red**,
  - and if it is a term, in *italic blue*.

Derivation of $\wedge_{\mathrm{Bool}}$

- First we derive $b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \lambda(d^{\mathbf{Bool}}).\mathit{Bool} : \mathbf{Bool} \rightarrow \mathrm{Set}$:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathit{Bool} : \mathrm{Set}}{b : \mathbf{Bool} \Rightarrow \mathrm{Context}}\,(\mathrm{Context}_1)}{b : \mathbf{Bool} \Rightarrow \mathit{Bool} : \mathrm{Set}}\,(\text{Bool-F})}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{Context}}\,(\mathrm{Context}_1)}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathit{Bool} : \mathrm{Set}}\,(\text{Bool-F})}{b : \mathbf{Bool}, c : \mathbf{Bool}, d : \mathbf{Bool} \Rightarrow \mathrm{Context}}\,(\mathrm{Context}_1)}{b : \mathbf{Bool}, c : \mathbf{Bool}, d : \mathbf{Bool} \Rightarrow \mathit{Bool} : \mathrm{Set}}\,(\text{Bool-F})}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \lambda d^{\mathbf{Bool}}.\mathit{Bool} : \mathbf{Bool} \rightarrow \mathrm{Set}}\,(\rightarrow\text{-I})
$$

Derivation of $\wedge_{\mathrm{Bool}}$

- We derive

$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathbf{Bool} = (\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ \mathrm{tt} : \mathrm{Set}$$

(using part of the derivation above):

$$\dfrac{\dfrac{\dfrac{\cdots}{b{:}\mathbf{Bool},c{:}\mathbf{Bool},d{:}\mathbf{Bool}\Rightarrow\mathrm{Context}}}{b{:}\mathbf{Bool},c{:}\mathbf{Bool},d{:}\mathbf{Bool}\Rightarrow\mathit{Bool}{:}\mathrm{Set}}\,(\mathrm{Bool\text{-}F}) \quad \dfrac{\dfrac{\cdots}{b{:}\mathbf{Bool},c{:}\mathbf{Bool}\Rightarrow\mathrm{Context}}}{b{:}\mathbf{Bool},c{:}\mathbf{Bool}\Rightarrow\mathrm{tt}{:}\mathbf{Bool}}\,(\mathrm{Bool\text{-}I_{tt}})}{\dfrac{b{:}\mathbf{Bool},c{:}\mathbf{Bool}\Rightarrow(\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ \mathrm{tt}{=}\mathit{Bool}{:}\mathrm{Set}}{b{:}\mathbf{Bool},c{:}\mathbf{Bool}\Rightarrow\mathbf{Bool}{=}(\lambda d^{\mathrm{Bool}}.\mathit{Bool})\ \mathrm{tt}{:}\mathrm{Set}}\,(\mathrm{Sym_{Elem}})}\,(\to\text{-}\mathrm{Eq})$$

Derivation of $\wedge_{\mathrm{Bool}}$

- Similarly follows

$$b : \textbf{Bool}, c : \textbf{Bool} \Rightarrow \textbf{Bool} = (\lambda d^{\textbf{Bool}}.\mathit{Bool})\ \mathrm{ff} : \mathrm{Set}$$

Derivation of $\wedge_{\mathrm{Bool}}$

- Using part of the proof above, we derive

$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow c : (\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ \mathrm{tt}$$

$$\dfrac{\dfrac{\dfrac{\cdots}{b{:}\mathbf{Bool},c{:}\mathbf{Bool}\Rightarrow\mathrm{Context}}}{b{:}\mathbf{Bool},c{:}\mathbf{Bool}\Rightarrow c{:}\mathbf{Bool}}\,(\mathrm{Ass}) \qquad \dfrac{\cdots}{b{:}\mathbf{Bool},c{:}\mathbf{Bool}\Rightarrow\mathbf{Bool}{=}(\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ \mathrm{tt}{:}\mathrm{Set}}}{b{:}\mathbf{Bool},c{:}\mathbf{Bool}\Rightarrow c{:}(\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ \mathrm{tt}}\,(\mathrm{Transfer}_0)$$

- We derive using $(\mathrm{Transfer}_0)$

$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{ff} : (\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ \mathrm{ff}$$

$$\dfrac{\dfrac{\dfrac{\cdots}{b{:}\mathbf{Bool},c{:}\mathbf{Bool}\Rightarrow\mathrm{Context}}}{b{:}\mathbf{Bool},c{:}\mathbf{Bool}\Rightarrow\mathrm{ff}{:}\mathbf{Bool}}\,(\mathrm{Bool\text{-}I_{ff}}) \qquad \dfrac{\cdots}{b{:}\mathbf{Bool},c{:}\mathbf{Bool}\Rightarrow\mathbf{Bool}{=}(\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ \mathrm{ff}{:}\mathrm{Set}}}{b{:}\mathbf{Bool},c{:}\mathbf{Bool}\Rightarrow\mathrm{ff}{:}(\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ \mathrm{ff}}\,(\mathrm{Transfer}_0)$$

Derivation of $\wedge_{\mathrm{Bool}}$

- We derive $b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow b : \mathbf{Bool}$ using part of the proof above:

$$\dfrac{\dfrac{\cdots}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{Context}}}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow b : \mathbf{Bool}}\,(\mathrm{Ass})$$

Derivation of $\wedge_{\mathrm{Bool}}$

- Finally we obtain our judgement (we stack the premises of the rule because of lack of space):

$$
\cfrac{
\cfrac{
\cfrac{
\begin{array}{c}
b{:}\mathbf{Bool}, c{:}\mathbf{Bool} \Rightarrow \lambda d^{\mathbf{Bool}}.\mathit{Bool} : \mathbf{Bool} \to \mathrm{Set} \\
b{:}\mathbf{Bool}, c{:}\mathbf{Bool} \Rightarrow c : (\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ \mathrm{tt} \\
b{:}\mathbf{Bool}, c{:}\mathbf{Bool} \Rightarrow \mathrm{ff} : (\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ \mathrm{ff} \\
b{:}\mathbf{Bool}, c{:}\mathbf{Bool} \Rightarrow b : \mathbf{Bool}
\end{array}
}{
b{:}\mathbf{Bool}, c{:}\mathbf{Bool} \Rightarrow \mathrm{Case}_{\mathrm{Bool}}\ (\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ c\ \mathrm{ff}\ b : \mathbf{Bool}
}\ (\mathrm{Bool\text{-}El})
}{
b{:}\mathbf{Bool} \Rightarrow \lambda c^{\mathbf{Bool}}.\mathrm{Case}_{\mathrm{Bool}}\ (\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ c\ \mathrm{ff}\ b : \mathbf{Bool} \to \mathbf{Bool}
}\ (\to\text{-}\mathrm{I})
}{
\lambda(b, c{:}\mathbf{Bool}).\mathrm{Case}_{\mathrm{Bool}}\ (\lambda d^{\mathbf{Bool}}.\mathit{Bool})\ c\ \mathrm{ff}\ b : \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}
}\ (\to\text{-}\mathrm{I})
$$

Elimination into Type

We can extend add elimination and equality rules, having as result Type:

Elimination Rule into Type

$$\frac{C{:}\mathrm{Bool}{\rightarrow}\mathrm{Type} \qquad case_{\mathrm{tt}}{:}C\ \mathrm{tt} \qquad case_{\mathrm{ff}}{:}C\ \mathrm{ff} \qquad b{:}\mathrm{Bool}}{\mathrm{Case}^{\mathrm{Type}}_{\mathrm{Bool}}\ C\ case_{\mathrm{tt}}\ case_{\mathrm{ff}}\ b : C\ b}\ (\mathrm{Bool\text{-}El}^{\mathrm{Type}})$$

Equality Rules into Type

$$\frac{C : \mathrm{Bool} \rightarrow \mathrm{Type} \qquad case_{\mathrm{tt}} : C\ \mathrm{tt} \qquad case_{\mathrm{ff}} : C\ \mathrm{ff}}{\mathrm{Case}^{\mathrm{Type}}_{\mathrm{Bool}}\ C\ case_{\mathrm{tt}}\ case_{\mathrm{ff}}\ \mathrm{tt} = case_{\mathrm{tt}} : C\ \mathrm{tt}}\ (\mathrm{Bool\text{-}Eq}^{\mathrm{Type}}_{\mathrm{ff}})$$

$$\frac{C : \mathrm{Bool} \rightarrow \mathrm{Type} \qquad case_{\mathrm{tt}} : C\ \mathrm{tt} \qquad case_{\mathrm{ff}} : C\ \mathrm{ff}}{\mathrm{Case}^{\mathrm{Type}}_{\mathrm{Bool}}\ C\ case_{\mathrm{tt}}\ case_{\mathrm{ff}}\ \mathrm{ff} = case_{\mathrm{ff}} : C\ \mathrm{ff}}\ (\mathrm{Bool\text{-}Eq}^{\mathrm{Type}}_{\mathrm{tt}})$$

Example Select

- Assume we have introduced

$$
\begin{array}{lcl}
\mathrm{FemaleName} & : & \mathrm{Set} \\
 & = & \{\mathrm{jill}, \mathrm{sara}\} \\
\mathrm{MaleName} & : & \mathrm{Set} \\
 & = & \{\mathrm{tom}, \mathrm{jim}\}
\end{array}
$$

- Then we can define

$$
\begin{array}{lcl}
\mathrm{Name} & : & \mathrm{Bool} \to \mathrm{Set} \\
 & := & \lambda x^{\mathrm{Bool}}.\mathrm{Case}_{\mathrm{Bool}}^{\mathrm{Type}}\ (\lambda y.\mathrm{Set}) \\
 & & \qquad \mathrm{FemaleName}\ \mathrm{MaleName}\ x \\
 & : & \mathrm{Bool} \to \mathrm{Set}
\end{array}
$$

Elimination into Type (Cont.)

We can extend this into an elimination rule
into Kind or other higher types.

6 (b) The Finite Sets

Bool can be generalised to sets having n elements (n a fixed natural number):

Formation Rule

$$\mathrm{Fin}_n : \mathrm{Set} \quad (\mathrm{Fin}_n\text{-F})$$

Introduction Rules

$$\mathrm{A}^n_k : \mathrm{Fin}_n \quad (\mathrm{Fin}_n\text{-I}_k)$$

(for $k = 0, \ldots, n - 1$)

Rules for Fin_n

Elimination Rule

$$\frac{\begin{array}{c} C : \mathrm{Fin}_n \to \mathrm{Set} \\ s_0 : C\ \mathrm{A}^n_0 \\ s_1 : C\ \mathrm{A}^n_1 \\ \cdots \\ s_{n-1} : C\ \mathrm{A}^n_{n-1} \\ a : \mathrm{Fin}_n \end{array}}{\mathrm{Case}_n\ C\ s_0\ \ldots\ s_{n-1}\ a : C\ a}\ (\mathrm{Fin}_n\text{-El})$$

The Finite Sets (Cont)

Equality Rules

$$\frac{\begin{array}{c} C : \mathrm{Fin}_n \to \mathrm{Set} \\ s_0 : C\ \mathrm{A}^n_0 \\ s_1 : C\ \mathrm{A}^n_1 \\ \cdots \\ s_{n-1} : C\ \mathrm{A}^n_{n-1} \end{array}}{\mathrm{Case}_n\ C\ s_0\ \ldots\ s_{n-1}\ \mathrm{A}^n_k = s_k : C\ \mathrm{A}^n_k}\ (\mathrm{Fin}_n\text{-}\mathrm{Eq}_k)$$

(for $k = 0, \ldots, n-1$).

We add as well **equality versions** of the formation-, introduction-, and elimination rules.

Remark: Note that we have just introduced infinitely many rules (for each $n \in \mathbb{N}$ and $k = 0, \ldots, n-1$).

Omitting Premises in Equality Rules

- Since the premises of the equality rule can in most cases be determined from the introduction and elimination rules, we will **usually omit them**, when writing down equality rules.
- So we write for instance for the previous rule:

$$\mathrm{Case}_n\ C\ s_0\ \ldots\ s_{n-1}\ \mathrm{A}_k^n = s_k : C\ \mathrm{A}_k^n$$

- We sometimes even **omit the type**:

$$\mathrm{Case}_n\ C\ s_0\ \ldots\ s_{n-1}\ \mathrm{A}_k^n = s_k$$

More Compact Elimination Rules

- $$\begin{aligned}\mathrm{Case}_n : (C : \mathrm{Fin}_n \to \mathrm{Set}) \\ \to (s_0 : C\ \mathrm{A}^n_0) \\ \to \cdots \\ \to (s_{n-1} : C\ \mathrm{A}^n_{n-1}) \\ \to (a : \mathrm{Fin}_n) \\ \to C\ a\end{aligned}$$.

Elimination into Type

- Similarly as for Bool we can write down **elimination rules**, where $\mathbf{C} : \mathbf{Fin_n} \to \mathbf{Type}$ (instead of $C : \mathrm{Fin}_n \to \mathrm{Set}$).
- This can be done for all sets defined later as well.

Rules for $\top$

$\top$ is the special case Fin_n for $n = 1$ (we write true for A_0^1):

Formation Rule

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

Introduction Rules

$$\mathrm{true} : \top \quad (\top\text{-I})$$

Elimination Rule

$$\frac{C : \top \to \mathrm{Set} \qquad c : C\ \mathrm{true} \qquad t : \top}{\mathrm{Case}_\top\ c\ t : C\ t}\ (\top\text{-El})$$

Rules for $\top$

Equality Rule

$$\mathrm{Case}_{\top}\ c\ \mathrm{true} = c$$

We add as well **equality versions** of the formation-, introduction-, and elimination rules.

Jump over next slide (advanced material)

Rules for $\top$ (Cont.)

- $\mathrm{Case}_\top$ is **computationally not very interesting**.
  - $\mathrm{Case}_\top\ c$ is the constant function $\lambda x^\top.c$.
  - However, in Agda we might not be able to derive

$$\lambda t^\top.c : (t : \top) \to C\ t$$

- From a **logic point of view**, it expresses:
  From an element of $C\ \mathrm{true}$ we obtain an element of $C\ t$
  for every t : $\top$.
  - So there is no $C : \top \to \mathrm{Set}$ s.t. $C\ \mathrm{true}$ is inhabited, but $C\ x$ is not inhabited for some other $x : \top$.
  - This means that all elements of x of type $\top$ are **indistinguishable from true**, i.e. they are **identical to true**.
  - This equality is called **Leibnitz equality**.

Rules for $\bot$

$\bot$ is the special case Fin_n for $n = 0$:

Formation Rule

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

There is no Introduction Rule

Elimination Rule

$$\frac{C : \bot \to \mathrm{Set} \qquad f : \bot}{\mathrm{Case}_\bot\ f : C\ f} (\bot\text{-El})$$

There is no Equality Rule

We add as well **equality versions** of the formation- and elimination rule.

6 (c) Atomic formulae and the Traffic Light Example

- Atom can be defined as follows:

$$\begin{array}{lcl} \mathrm{Atom} & : & \mathrm{Bool} \rightarrow \mathrm{Set} \\ \mathrm{Atom} & = & \mathrm{Case}_{\mathrm{Bool}}^{\mathrm{Type}}\ (\lambda b^{\mathrm{Bool}}.\mathrm{Set})\ \top\ \bot \end{array}$$

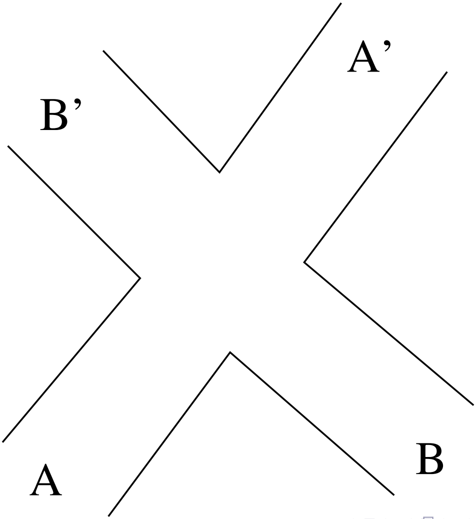
- So we have

$$\begin{array}{rcl} \mathrm{Atom}\ \mathrm{tt} & = & \top \\ \mathrm{Atom}\ \mathrm{ff} & = & \bot \end{array}$$

Jump over Traffic Light Example.

The Traffic Light Example

- Assume a **road crossing**, controlled by **traffic lights**:



The Traffic Light Example

- Assume from each direction A, A', B, B' there is one traffic light,
  - but A and A' always coincide, similarly B and B'.

The Set of Physical States

- For simplicity assume that **each traffic light is either red or green**:

$$\begin{array}{l} \text{data Colour : Set where} \\ \quad \begin{array}{lcl} \text{red} & : & \text{Colour} \\ \text{green} & : & \text{Colour} \end{array} \end{array}$$

- The set of **physical states of the system** is given by a pair, determining the colour of A (and therefore as well A') and of B (and B')

$$\begin{array}{l} \text{record PhysState : Set where} \\ \quad \text{field} \\ \qquad \begin{array}{lcl} \text{sigA} & : & \text{Colour} \\ \text{sigB} & : & \text{Colour} \end{array} \end{array}$$

The Set of Control States

- The set of **control states** is a set of states of the system, a controller of the system can choose.
  - Each of these states **should be safe**.
  - In our example, **all safe states will be captured** (this can usually be only achieved in small examples).
- A **complete set of control states** consists of:
  - allRed – all signals are red.
  - $\mathrm{onlyAGreen}$ – signal A (and A') is green, signal B is red.
  - $\mathrm{onlyBGreen}$ – signal B is green, signal A is red.

The Set of Control States (Cont.)

- We therefore define

$$
\begin{array}{l}
\text{data ControlState} : \text{Set where} \\
\quad \begin{array}{lcl}
\text{allRed} & : & \text{ControlState} \\
\text{onlyAGreen} & : & \text{ControlState} \\
\text{onlyBGreen} & : & \text{ControlState}
\end{array}
\end{array}
$$

Control States to Physical States

- We define the **state of signals A, B depending on a control state**:

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\mathrm{toSigA} : \mathrm{ControlState} \to \mathrm{Colour}} \\
\mathrm{toSigA} & \mathrm{allRed} & = & \mathrm{red} \\
\mathrm{toSigA} & \mathrm{onlyAGreen} & = & \mathrm{green} \\
\mathrm{toSigA} & \mathrm{onlyBGreen} & = & \mathrm{red}
\end{array}
$$

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\mathrm{toSigB} : \mathrm{ControlState} \to \mathrm{Colour}} \\
\mathrm{toSigB} & \mathrm{allRed} & = & \mathrm{red} \\
\mathrm{toSigB} & \mathrm{onlyAGreen} & = & \mathrm{red} \\
\mathrm{toSigB} & \mathrm{onlyBGreen} & = & \mathrm{green}
\end{array}
$$

Control States to Physical States

- Now we can define the **physical state corresponding to a control state**:

$$
\begin{array}{l}
\mathrm{toPhysState} : \mathrm{ControlState} \to \mathrm{PhysState} \\
\mathrm{toPhysState}\ c = \mathrm{record}\{\begin{array}{lcl}\mathrm{sigA} & = & \mathrm{toSigA}\ c\ ; \\ \mathrm{sigB} & = & \mathrm{toSigB}\ c\ \}\end{array}
\end{array}
$$

Safety Predicate

- We define now **when a physical state is safe**:
  - It is **safe iff not both signals are green**.
  - We define now a corresponding predicate **directly**, without defining first a Boolean function.
  - We first define a predicate depending on two signals:

$$
\begin{array}{llll}
\mathrm{CorAux} : \mathrm{Colour} \to \mathrm{Colour} \to \mathrm{Set} & & & \\
\mathrm{CorAux} \quad \mathrm{red} & \_ & = & \top \\
\mathrm{CorAux} \quad \mathrm{green} & \mathrm{red} & = & \top \\
\mathrm{CorAux} \quad \mathrm{green} & \mathrm{green} & = & \bot
\end{array}
$$

Safety Predicate (Cont.)

- Now we define

$$\begin{array}{l}\mathrm{Cor} : \mathrm{PhysState} \to \mathrm{Set} \\ \mathrm{Cor}\ s = \mathrm{CorAux}\ (\mathrm{PhysState.sigA}\ s)\ (\mathrm{PhysState.sigB}\ s)\end{array}$$

- **Remark:** In some cases in order to define a function from a **record type** into some other set, it is better first to **introduce an auxiliary function**, depending on the components of that product.

Safety of the System

- Now we show that **all control states are safe**:

$$
\begin{array}{llcl}
\mathrm{corProof} : (s : \mathrm{ControlState}) \to \mathrm{Cor}\ (\mathrm{toPhysState}\ s) & & & \\
\mathrm{corProof} & \mathrm{allRed} & = & \mathrm{true} \\
\mathrm{corProof} & \mathrm{onlyAGreen} & = & \mathrm{true} \\
\mathrm{corProof} & \mathrm{onlyBGreen} & = & \mathrm{true}
\end{array}
$$

See **exampleTrafficLight1.agda**

Safety of the System (Cont.)

- The first element true was an element of **Cor (phys_state Allred)**, which reduces to $\top$.
- Similarly for the other two elements.
- This works only because **each control state corresponds to a correct physical state**.
  - If this hadn't been the case, we would have gotten instances where the goal to solve is $\bot$, which we can't solve.

Safety of the System (Cont.)

- If one makes a **mistake** which results in an unsafe situation
  - e.g. sets $\mathrm{toSigB\ onlyAGreen} = \mathrm{green}$,

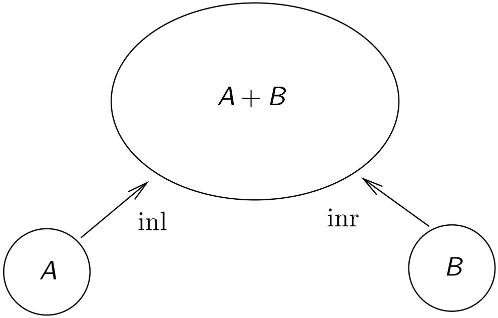
  then in the last step we obtain one goal of type $\perp$.
  - Then we can't solve this goal directly and **cannot prove the correctness**.
  - (We could in Agda solve this goal by using **full recursion**,
    - e.g. solve this goal as **corProof Agreen**,

    but this would be rejected by the termination checker.)

6 (d) The Disjoint Union of Sets

- The **disjoint union** $A + B$ of two sets A and B is the union of A and B,
  - but defined in such a way that we can decide whether an element of this union is originally from A or B.
  - This is distinguished by having constructors $\mathrm{inl} : A \to A + B$ and $\mathrm{inr} : B \to A + B$.
    - Elements from $a : A$ are inserted into $A + B$ as $\mathrm{inl}\ a : A + B$.
    - elements from $b : B$ are inserted into $A + B$ as $\mathrm{inr}\ b : A + B$.
    - inl stands for "in-left", inr for "in-right".
  - If we have $a : A$ and $a : B$, then a is represented both as $\mathrm{inl}\ a$ and $\mathrm{inr}\ a$ in $A + B$.

Visualisation $(A + B)$



Disjoint Union

- Informally, if

  $$A = \{1, 2\}$$

  and

  $$B = \{1, 2, 3\} \ ,$$

  then

  $$A + B = \{\mathrm{inl}(1), \mathrm{inl}(2), \mathrm{inr}(1), \mathrm{inr}(2), \mathrm{inr}(3)\}$$

  - Each element of $A + B$ is
    - either of the form $\mathrm{inl}(a)$ for some $a : A$
    - or of the form $\mathrm{inr}(b)$ for $b : B$.

Jump over Comparison with Product

Comparison with the Product

- Note that if we have again

$$A = \{1, 2\}$$

and

$$B = \{1, 2, 3\} \ ,$$

then for the product we have informally

$$A \times B = \{\mathrm{p}(1,1), \mathrm{p}(1,2), \mathrm{p}(1,3), \mathrm{p}(2,1), \mathrm{p}(2,2), \mathrm{p}(2,3)\}$$

  - Each element of $A \times B$ is of the form $\mathrm{p}(a, b)$ where $a : A$ and $b : B$.
  - So each element of $A \times B$ contains both an element of A and an element of B.

Disjoint Union vs. Product

- Note that, if A is empty, then
  - $A + B = \{\mathrm{inr}(b) \mid b : B\}$, which has a copy of each element of B,
  - $A \times B$ is empty, since we cannot form a pair $\mathrm{p}(a, b)$ where $a : A$, $b : B$, since there is no element $a : A$.

Rules for $A + B$

Formation Rule

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

Introduction Rules

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set} \qquad a : A}{\mathrm{inl}\ A\ B\ a : A + B} \, (+\text{-I}_{\mathrm{inl}})$$

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set} \qquad b : B}{\mathrm{inr}\ A\ B\ b : A + B} \, (+\text{-I}_{\mathrm{inr}})$$

Rules for $A + B$

Elimination Rules

$$\frac{\begin{array}{c} A : \mathrm{Set} \\ B : \mathrm{Set} \\ C : (A + B) \to \mathrm{Set} \\ case_{inl} : (a : A) \to C\ (\mathrm{inl}\ A\ B\ a) \\ case_{inr} : (b : B) \to C\ (\mathrm{inr}\ A\ B\ b) \\ d : A + B \end{array}}{\mathrm{Case}_{+}\ A\ B\ C\ case_{inl}\ case_{inr}\ d : C\ d}\ (+\text{-El})$$

($case_{inl}$, $case_{inr}$ stand for "case left", "case right").

Rules for $A + B$

Equality Rules

$$\begin{array}{ll} \mathrm{Case}_{+}\ A\ B\ C\ case_{inl}\ case_{inr}\ (\mathrm{inl}\ A\ B\ a) & \\ \quad = case_{inl}\ a : C\ (\mathrm{inl}\ A\ B\ a) & (+\text{-}\mathrm{Eq}_{\mathrm{inl}}) \end{array}$$

$$\begin{array}{ll} \mathrm{Case}_{+}\ A\ B\ C\ case_{inl}\ case_{inr}\ (\mathrm{inr}\ A\ B\ b) & \\ \quad = case_{inr}\ b : C\ (\mathrm{inr}\ A\ B\ b) & (+\text{-}\mathrm{Eq}_{\mathrm{inr}}) \end{array}$$

Additionally, we have the **equality versions** of the formation-, introduction and elimination rules.

Logical Framework Version

- A **more compact notation** for the formation, introduction and elimination rules is:
  - $\_+\_ : \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set}$, written infix.
  - $\mathrm{inl} : (A, B : \mathrm{Set}) \to A \to (A + B)$.
  - $\mathrm{inr} : (A, B : \mathrm{Set}) \to B \to (A + B)$.
  - $$\begin{aligned} \mathrm{Case}_+ : (A, B : \mathrm{Set}) \\ &\to (C : (A + B) \to \mathrm{Set}) \\ &\quad \to ((a : A) \to C\ (\mathrm{inl}\ A\ B\ a)) \\ &\quad\quad \to ((b : B) \to C\ (\mathrm{inr}\ A\ B\ b)) \\ &\quad\quad\quad \to (d : A + B) \\ &\quad\quad\quad\quad \to C\ d\ . \end{aligned}$$
  - Equality rule as before.

Disjoint Union in Agda

- The disjoint union can be defined as a "data"-set having **two constructors**
  - inl (in-left for left injection) and
  - inr (in-right for right injection):

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

Disjoint Union in Agda (Cont.)

- Elimination is represented by pattern matching.
  So if want to define for $A, B : \mathrm{Set}$ for instance

$$\begin{array}{l} f : A + B \to \mathrm{Bool} \\ f\ \ x = \{!\ \ !\} \end{array}$$

we can define $f\ \ x$ by case distinction on x:

$$\begin{array}{l} f : A + B \to \mathrm{Bool} \\ f\ \ (\mathrm{inl}\ \ a) = \mathrm{tt} \\ f\ \ (\mathrm{inr}\ \ b) = \mathrm{ff} \end{array}$$

Use of Concrete Disjoint Sets

- It is usually **more convenient** to define concrete disjoint unions **directly** with more intuitive names for constructors, e.g.

$$\begin{array}{l}\mathrm{data}\ \mathrm{Plant} : \mathrm{Set}\ \mathrm{where}\\ \quad \begin{array}{lcl}\mathrm{tree} & : & \mathrm{Tree} \to \mathrm{Plant}\\ \mathrm{flower} & : & \mathrm{Flower} \to \mathrm{Plant}\end{array}\end{array}$$

- Now one can define for instance

$$\begin{array}{l}\mathrm{isFlower} : \mathrm{Plant} \to \mathrm{Bool}\\ \begin{array}{lcl}\mathrm{isFlower}\ (\mathrm{tree}\ t) & = & \mathrm{ff}\\ \mathrm{isFlower}\ (\mathrm{flower}\ f) & = & \mathrm{tt}\end{array}\end{array}$$

Disjunction

- $A \vee B$ is true iff A is true or B is true.
- Therefore a **proof of A $\vee$ B consists of a proof of A or a proof of B, plus the information which one.**
  - It is therefore an element $\mathrm{inl}\ p$ for a proof $p : A$ or an element $\mathrm{inr}\ q$ for a proof $q : B$.
- Therefore the set of proofs of $A \vee B$ is the **disjoint union of A and B**, i.e. **A + B**.
- We can **identify** $A \vee B$ with $A + B$.

Disjunction in Agda

- Or is represented as disjoint union in type theory.
- In Agda we can type in the symbol for $\vee$ using Leim as \vee.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{data}\ \_\vee\_\ (A\ B : \mathrm{Set}) : \mathrm{Set}\ \mathrm{where}} \\
\quad \mathrm{or1} & : & A \to A \vee B \\
\quad \mathrm{or2} & : & B \to A \vee B
\end{array}
$$

- See **exampleproofproplogic7.agda**.
- On the blackboard $A \to A \vee B$ and $A \vee A \to A$ will now be shown in Agda.

Example (Disjunction)

- The following derives $(A \vee B) \to (B \vee A)$:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathrm{lemma3} : A \vee B \to B \vee A} \\
\mathrm{lemma3}\ (\mathrm{or1}\ a) & = & \mathrm{or2}\ a \\
\mathrm{lemma3}\ (\mathrm{or2}\ b) & = & \mathrm{or1}\ b
\end{array}
$$

- See **exampleproofproplogic9.agda**.

Disjunction with more Args.

- As for conjunction, it is useful to introduce special ternary versions of the disjunction (and versions with higher arities):

$$
\begin{array}{l}
\mathrm{data}\ \mathrm{OR3}\ (A\ B\ C : \mathrm{Set}) : \mathrm{Set}\ \mathrm{where} \\
\quad \begin{array}{lcl}
\mathrm{or1} & : & A \to \mathrm{OR3}\ A\ B\ C \\
\mathrm{or2} & : & B \to \mathrm{OR3}\ A\ B\ C \\
\mathrm{or3} & : & C \to \mathrm{OR3}\ A\ B\ C
\end{array}
\end{array}
$$

- See **exampleproofproplogic8.agda**.

Jump over Σ-Type.

6 (e) The **Σ**-Set

- The Σ-set is a second version of the **dependent product** of two sets.
- It depends on
  - a set A,
  - and a second set B depending on A, i.e. on $B : A \to \mathrm{Set}$.
- Similar to the standard product $(x : A) \times (B\ x)$.
- In Agda
  - $(x : A) \times (B\ x)$ is a in Agda a builtin construct,
  - the Σ-set is introduced by the user using a constructor, similar to the previous sets.
- The Σ-set behaves sometimes better than the standard product.

Rules for Σ

Formation Rule

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

Introduction Rule

$$\frac{\begin{array}{c} A : \mathrm{Set} \\ B : A \to \mathrm{Set} \\ a : A \\ b : B\ a \end{array}}{\mathrm{p}\ A\ B\ a\ b : \Sigma\ A\ B}\ (\Sigma\text{-I})$$

Rules for Σ

Elimination Rule

$$\frac{\begin{array}{c} A : \mathrm{Set} \\ B : A \to \mathrm{Set} \\ C : (\Sigma\ A\ B) \to \mathrm{Set} \\ c : (a : A) \to (b : B\ a) \to C\ (\mathrm{p}\ A\ B\ a\ b)) \\ d : \Sigma\ A\ B \end{array}}{\mathrm{Case}_{\Sigma}\ A\ B\ C\ c\ d : C\ d}\ (\Sigma\text{-El})$$

Equality Rule

$$\mathrm{Case}_{\Sigma}\ A\ B\ C\ c\ (\mathrm{p}\ A\ B\ a\ b) = c\ a\ b : C\ (\mathrm{p}\ A\ B\ a\ b) \quad (\Sigma\text{-Eq})$$

Additionally we have the **Equality versions** of the formation-, introduction- and elimination-rules.

The Σ-Set using the Log. Framew.

- The more compact notation is:
  - $$\begin{array}{l}\Sigma : (A : \mathrm{Set}) \\ \quad \to (A \to \mathrm{Set}) \\ \qquad \to \mathrm{Set} \quad .\end{array}$$
  - $$\begin{array}{l}\mathrm{p} : (A : \mathrm{Set}) \\ \quad \to (B : A \to \mathrm{Set}) \\ \qquad \to (a : A) \\ \qquad\quad \to (B\ a) \\ \qquad\qquad \to \Sigma\ A\ B \quad .\end{array}$$

The Σ-Set using the Log. Framew.

- Case_{Σ} :
  $$\begin{array}{l} (A : \mathrm{Set}) \\ \quad \to (B : A \to \mathrm{Set}) \\ \qquad \to (C : (\Sigma\ A\ B) \to \mathrm{Set}) \\ \qquad\quad \to ((a : A, b : B\ a) \to C\ (\mathrm{p}\ A\ B\ a\ b)) \\ \qquad\qquad \to (d : \Sigma\ A\ B) \\ \qquad\qquad\quad \to C\ d\ \ . \end{array}$$
- Equality rule as before.

The Σ-Set and the Dep. Prod.

- Both the Σ-set and the dep. product have similar introduction rules.
  - For the Σ-set, the constructors have additional arguments A, B necessary for bureaucratic reasons only.
- One can define the projections π_0, π_1 using Case_Σ:

$$\begin{array}{rcl}\pi_0 & = & \mathrm{Case}_\Sigma\ A\ B\ (\lambda x^{(\Sigma\ A\ B)}.A)\ (\lambda x^A.\lambda y^{(B\ x)}.x) \\ \pi_1 & = & \mathrm{Case}_\Sigma\ A\ B\ (\lambda x^{(\Sigma\ A\ B)}.B\ \pi_0(x))\ (\lambda x^A.\lambda y^{(B\ x)}.y)\end{array}$$

- On the other hand, from π_0, π_1 we can define Case_Σ as follows:

$$\begin{array}{l}\lambda A^{\mathrm{Set}}.\lambda B^{A\to\mathrm{Set}}.\lambda C^{(\Sigma\ A\ B)\to\mathrm{Set}}. \\ \quad \lambda s^{(a:A)\to(b:B\ a)\to C\ (\mathrm{p}\ a\ b)}.\lambda d^{(\Sigma\ A\ B)}.s\ \pi_0(d)\ \pi_1(d)\ .\end{array}$$

The **Σ**-Set and the Dep. Prod.

- However the dependent product has the **η-rule** (which is however not implemented in Agda).
- Because of the lack of η-rule, Σ works usually **better than the dependent product** in Agda.
  - I personally **don't use the dependent product** of Agda much.

The Σ-Set in Agda

- Σ can be defined as a "data"-set with a constructor, e.g. p:

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

- Elimination uses **case-distinction**:

$$\begin{array}{l} f : \Sigma\ A\ B \to D \\ f\ (\mathrm{p}\ a\ b) = \{!\ \ !\} \end{array}$$

sigmaset.agda

The Σ-Set in Agda (Cont.)

- Again one usually defines concrete Σ-sets more directly.
- **Example:** Assume we have defined
  - a set $\mathrm{PlantGroup}$ for **groups of plants** (e.g. "tree", "flower"),
  - depending on $g : \mathrm{PlantGroup}$, sets $(\mathrm{PlantsInGroup}\ g)$ for **plants in that group**.
- The **set of plants** can then be defined as

$$\begin{array}{l} \mathrm{data\ Plant : Set\ where} \\ \mathrm{plant} : (g : \mathrm{PlantGroup}) \to \mathrm{PlantsInGroup}\ g \to \mathrm{Plant} \end{array}$$

The Σ-Set in Agda (Cont.)

- Not surprisingly, for **elimination** we use **pattern matching**, e.g.:

$$\begin{aligned} &f : \mathrm{Plant} \to \mathrm{PlantGroup} \\ &f\ (\mathrm{plant}\ g\ \_) = g \end{aligned}$$

6 (f) Natural Deduction and Dependent Type Theory

- In this section we study, how derivations in dependent type theory correspond to derivations in natural deduction. (Omitted 2008)
- We will as well introduce constructive logic.
  Jump to constructive logic.

Conjunction

- We have seen before that we can identify in type theory conjunction with the non-dependent product.
- With this interpretation, the **introduction rule** for the product allows to form a proof of $A \wedge B$ from a proof of A and a proof of B:

$$\frac{p : A \qquad q : B}{\langle p, q \rangle : A \wedge B}\,(\times\text{-I})$$

- This means that we can **derive A $\wedge$ B from A and B**.

Conjunction and Natural Ded.

- In so called natural deduction, one has rules for deriving and eliminating formulas formed using the standard connectives.
- There the rule for introducing proofs of $A \wedge B$ is

$$\frac{A \qquad B}{A \wedge B}\,(\wedge\text{-I})$$

- The type theoretic introduction rule corresponds exactly to this rule.

Omit Example1

Example 1

- For instance, assume we want to prove that a function sort from lists to lists is a sorting algorithm.
- Then we have to show that for every list l the application of sort to l is sorted, and has the same elements of l.
- In order to show this, one would assume a list l and show
  - first that $\mathrm{sort}\ l$ is sorted,
  - then, that $\mathrm{sort}\ l$ has the same elements as l
  - and finally conclude that it fulfils the conjunction of both properties.
  - The last operation uses the introduction rule for $\wedge$.

Conjunction (Cont.)

- The **elimination rule** for $\wedge$ allows to project a proof of $A \wedge B$ to a proof of A and a proof of B:

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

- This means that we can **derive from A $\wedge$ B both A and B**.
- This corresponds to the **natural deduction elimination rule for** $\wedge$:

$$\frac{A \wedge B}{A}\,(\wedge\text{-El}_0) \qquad \frac{A \wedge B}{B}\,(\wedge\text{-El}_1)$$

Omit Example 2

Example 2

- Assume we have defined a function f, which takes a list of natural numbers l, a proof that l is sorted, and a natural number n, and returns the Boolean value tt or ff indicating whether n is in this list or not.
- Assume now a sorting function sort from lists of natural numbers to natural numbers, plus a proof that it is a sorting function, i.e. that $\mathrm{sort}\ l$ is sorted and has the same elements as l for every list l.
- We want to apply f to $\mathrm{sort}\ l$ and need therefore a proof that $\mathrm{sort}\ l$ is sorted.
- We have that the conjunction of "$\mathrm{sort}\ l$ is sorted" and "$\mathrm{sort}\ l$ has the same elements as l" holds.
- Using the elimination rule for $\wedge$ one can conclude the desired property, that $\mathrm{sort}\ l$ is sorted.

Example 3

- Assume a proof of $A \wedge B$.
- We want to show $B \wedge A$.
  - By $\wedge$-elimination we obtain from $A \wedge B$ that B holds.
  - Similarly we conclude that A holds.
  - Using $\wedge$-introduction we conclude $B \wedge A$.
  - In natural deduction, this proof is as follows:

$$\dfrac{\dfrac{A \wedge B}{B}\,(\wedge\text{-El}_0) \qquad \dfrac{A \wedge B}{A}\,(\wedge\text{-El}_1)}{B \wedge A}\,(\wedge\text{-I})$$

- We have seen in the previous section how to derive this in Agda.

Disjunction

- We have seen before that we can identify in type theory disjunction can be identified with the disjoint union.
- With this identification, the **introduction rules** for $+$ allows to form a proof of $A \vee B$ from a proof of A or from a proof of B.

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set} \qquad p : A}{\mathrm{inl}\ A\ B\ p : A + B}\ (+\text{-}\mathrm{I}_{\mathrm{inl}})$$

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set} \qquad p : B}{\mathrm{inr}\ A\ B\ p : A + B}\ (+\text{-}\mathrm{I}_{\mathrm{inr}})$$

Disjunction (Cont.)

- Omitting the premises $A, B : \mathrm{Set}$ and omitting them as arguments of inl and inr (which is needed only for type checking purposes in the presence of the identity type – this type is not treated in this module) we get:

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set} \qquad p : A}{\mathrm{inl}\ p : A + B}\ (+\text{-}\mathrm{I}_{\mathrm{inl}})$$

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set} \qquad p : B}{\mathrm{inr}\ p : A + B}\ (+\text{-}\mathrm{I}_{\mathrm{inr}})$$

Disjunction (Cont.)

- This means that we can **derive A $\vee$ B from A and from B**.
- This is what is expressed by the **natural deduction introduction rules for $\vee$**:

$$\frac{A}{A \vee B}\,(\vee\text{-}\mathrm{I}_{\mathrm{inl}}) \qquad\qquad \frac{B}{A \vee B}\,(\vee\text{-}\mathrm{I}_{\mathrm{inr}})$$

Omit Example 1

Example 1

- Assume we want to show that every prime number is equal to 2 or odd.
- In order to show this one assumes a prime number.
  - If it is 2, it is trivially equal to 2.
    - Using the introduction rule for $\vee$ one concludes that it is equal to 2 or odd.
  - Otherwise, one argues (using some proof) that it is odd.
    - Using the introduction rule for $\vee$ one concludes again that it is equal to 2 or odd.

Disjunction (Cont.)

- The **elimination rule** for $+$ allows to form from an element of $A + B$ an element of any set C provided we can compute such an element from A and from B:

$$\frac{\begin{array}{c} A : \mathrm{Set} \\ B : \mathrm{Set} \\ C : (A \vee B) \to \mathrm{Set} \\ sl : (a : A) \to C\ (\mathrm{inl}\ A\ B\ a) \\ sr : (b : B) \to C\ (\mathrm{inr}\ A\ B\ b) \\ d : A \vee B \end{array}}{\mathrm{Case}_{+}\ A\ B\ C\ sl\ sr\ d : C\ d}\ (+\text{-El})$$

Disjunction (Cont.)

- Omitting the dependency of C on $A \vee B$, the premises A, B and C, and the arguments A, B and C, we get:

$$\frac{d : A \vee B \qquad sl : A \to C \qquad sr : B \to C}{\mathrm{Case}_{+}\ sl\ sr\ d : C}\,(+\text{-El})$$

- This means that we can **derive from A $\vee$ B a formula C, if we can derive C from A and from B**.

Disjunction (Cont.)

- This is what is expressed by the **natural deduction elimination rules for** $\vee$:

$$\frac{A \vee B \qquad A \vdash C \qquad B \vdash C}{C}\,(\vee\text{-El})$$

- In the above rule we have written

$$A \vdash C$$

for

from assumption A we can derive C.

  - This is written sometimes in the following form

$$\begin{array}{c} A \\ \vdots \\ C \end{array}$$

Disjunction (Cont.)

- Note that in natural deduction, from the premise $A \vdash C$ we obtain $A \to C$, which is the premise used in the corresponding rule in dependent type theory.

Omit Example 2

Example 2

- Assume we want to show that every prime number is equal to 2, equal to 3, or ≥ 5.
- We want to make use of the proof above that every prime number is equal to 2 or odd.
- We assume a prime number.
  - We know that it is equal to 2 or odd.
  - In case it is equal to 2 we conclude that it is equal to 2, equal to 3, or ≥ 5.
  - In case it is odd, we conclude using the fact that it is prime and 1 is not prime, that it is equal to 3 or ≥ 5.
    Therefore it is equal to 2, equal to 3, or ≥ 5.
  - Now from the elimination rule for $\vee$ we conclude that the prime number chosen is equal to 2, equal to 3, or ≥ 5.

Example 3

- Assume a proof of $A \lor B$.
- We want to show $B \lor A$.
  - We have $A \lor B$.
  - From assumption A we obtain A and therefore by $\lor$-introduction $B \lor A$.
  - From assumption B we obtain B and therefore by $\lor$-introduction $B \lor A$.
  - By $\lor$-elimination we obtain from these three premises $B \lor A$ without any premises.

Example 3 (Cont.)

- In natural deduction, this proof is as follows (we write $A_1, \ldots, A_n \vdash B$ for B follows under assumptions $A_1, \ldots, A_n$):

$$\dfrac{A \vee B \qquad \dfrac{A \vdash A}{A \vdash B \vee A}\,(\vee\text{-I}_{\text{inr}}) \qquad \dfrac{B \vdash B}{B \vdash B \vee A}\,(\vee\text{-I}_{\text{inr}})}{B \vee A}\,(\vee\text{-El})$$

- We have seen in the previous section how to derive this in Agda.

Implication

- We have seen before that we can identify in type theory implication with the non-dependent function type.
- In order to distinguish between the function type and the logical implication we will write in this subsection $\supset$ instead of $\to$ for logical implication.

Implication (Cont.)

- With this identification of logical implication and the function type, the **introduction rule for $\rightarrow$** allows to form a proof of $A \supset B$ from a proof of B depending on a proof p of A:

$$\frac{p : A \Rightarrow q : B}{\lambda p^A.q : A \supset B} \ (\rightarrow\text{-I})$$

- This means that, if we, **from assumptions p:A can prove B**
  - (i.e. we can make use of a context $p : A$ for proving $q : B$)

  then we can derive A $\supset$ B without assuming p:A.

Implication (Cont.)

- This is what is expressed by the **introduction rule for $\supset$ in natural deduction**:

$$\frac{A \vdash B}{A \supset B}\,(\supset\text{-I})$$

Example

- We extend the proof that, if we have $A \vee B$, then we have $B \vee A$, to a proof of
$$(A \vee B) \supset (B \vee A)$$
- The previous proof can be easily transformed into a proof of $A \vee B \vdash B \vee A$.
- By $\supset$-introduction, it follows $(A \vee B) \supset (B \vee A)$.

Example

- The complete proof in natural deduction is as follows is as follows.

$$\dfrac{\dfrac{A \lor B \vdash A \lor B \qquad \dfrac{A \vdash A}{A \vdash B \lor A}\,(\lor\text{-I}_{\text{inr}}) \qquad \dfrac{B \vdash B}{B \vdash B \lor A}\,(\lor\text{-I}_{\text{inl}})}{A \lor B \vdash B \lor A}\,(\lor\text{-El})}{(A \lor B) \supset (B \lor A)}\,(\supset\text{-I})$$

Implication (Cont.)

- The **elimination rule for $\supset$** allows to apply a proof p of $A \supset B$ to a proof of q of A in order to obtain a proof of B:

$$\frac{p : A \supset B \qquad q : A}{p\ q : B}\ (\rightarrow\text{-El})$$

- This means that we can **derive from A $\supset$ B and A that B holds**.
- This is what is expressed by the **natural deduction elimination rule for $\supset$**:

$$\frac{A \supset B \qquad A}{B}\ (\supset\text{-El})$$

Example

- Assume we want to show $A \supset (A \supset B) \supset B$.
- We can show this as follows:
  - From assumptions A and $A \supset B$ we can conclude $A \supset B$.
  - From assumptions A and $A \supset B$ we can conclude as well A.
  - Using the elimination rule for $\supset$, we conclude that under the same assumptions we get B.
  - Using the introduction rule for $\supset$ we conclude from assumption A that $(A \supset B) \supset B$ holds.
  - Using again the introduction rule for $\supset$ we conclude that $A \supset (A \supset B) \supset B$ holds without any assumptions.

Example

- A proof in natural deduction is as follows:

$$\dfrac{\dfrac{\dfrac{A, A \supset B \vdash A \supset B \qquad A, A \supset B \vdash A}{A, A \supset B \vdash B}\,(\supset\text{-El})}{A \vdash (A \supset B) \supset B}\,(\supset\text{-I})}{A \supset (A \supset B) \supset B}\,(\supset\text{-I})$$

Universal Quantification

- We have seen before that we can identify in type theory universal quantification with the dependent function type.
- With this identification, the **introduction rule** for the dependent function type allows to form a proof of $\forall x^A.B$ from a proof of B depending on an element $x : A$:

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

- This means that, if we, **from x:A can prove B**, then we get a proof of $\forall x^A.B$ which doesn't depend on $x : A$.

Universal Quantification (Cont.)

- This is what is expressed by the **natural deduction introduction rule for** $\forall$:

$$\frac{x : A \vdash B}{\forall x^A . B}\ (\forall\text{-I})$$

where

- **x might not occur free in any assumption of the proof**.
  - This is guaranteed in type theory, since $x : A$ must be the last element of the context, so any other assumptions must be located before it and can therefore **not depend on x:A.**

Universal Quantification (Cont.)

- Note that we have written

$$x : A \vdash B$$

for

we can derive B from variable $x : A$.

- This is usually not mentioned as such in natural deduction.
- We prefer this notation, since it
  - makes the variable x explicit,
  - and allows to deal with more complex types A.

Universal Quantification (Cont.)

- The **conclusion** of the introduction rule **will no longer depend on free variables x.**
  - This is made explicit by mentioning free variables $x : A$ in our notation.
  - In type theory this corresponds to the fact that **x:A does no longer occur in the context of the conclusion.**

Example

- Assume one wants to show that for every natural number n we have $n + 0 == n$.
- In order to show this one assumes a natural number n and shows then that $n + 0 == n$.
- then using the introduction rule for $\forall$ one concludes $\forall n^{\mathbb{N}}.n + 0 == n$.
- In natural deduction, this proof is as follows (where the prove of $n + 0 == n$ is not carried out):

$$\frac{n + 0 == n}{\forall n^{\mathbb{N}}.n + 0 == n}\,(\forall\text{-I})$$

Universal Quantification (Cont.)

- The **elimination rule** for the dependent function type allows to apply a proof p of $\forall x^A.B$ to an element $a : A$ in order to obtain a proof of $B[x := a]$:

$$\frac{p : \forall x^A.B \qquad a : A}{p\ a : B[x := a]}\,(\to\text{-El})$$

- This means that we can **derive from $\forall x^A.B$ and an element of a:A that B[x:=a] holds**.

Universal Quantification (Cont.)

- This is what is expressed by the **natural deduction elimination rule for $\forall$**
  - For the simple languages used in natural deduction, there is no need to derive that $a : A$;
    in more **complex type theories we have to carry out this derivation**.

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

Example

- Assume a proof of $\forall n^{\mathbb{N}}.0 + n == n$.
- We want to conclude that $\forall n, m : \mathbb{N}.0 + (n + m) == (n + m)$.
- This can be done as follows:
  - One assumes $n, m : \mathbb{N}$.
  - Then one can conclude $n + m : \mathbb{N}$.
  - Using $\forall n^{\mathbb{N}}.0 + n == n$ and the elimination rule for $\forall$ one concludes $0 + (n + m) == (n + m)$ under assumption $n, m : \mathbb{N}$.
  - Now using the introduction rule for $\forall$ twice it follows $\forall n, m : \mathbb{N}.0 + (n + m) == (n + m)$.

Example

- In natural deduction, this proof is written as follows:

$$
\dfrac{\forall n^{\mathbb{N}}.0 + n == n \qquad \dfrac{n:\mathbb{N}, m:\mathbb{N} \vdash n:\mathbb{N} \qquad n:\mathbb{N}, m:\mathbb{N} \vdash m:\mathbb{N}}{n : \mathbb{N}, m : \mathbb{N} \vdash n + m : \mathbb{N}}\,(\mathbb{N}\text{-El}_{+})}{\dfrac{n : \mathbb{N}, m : \mathbb{N} \vdash 0 + (n + m) == (n + m)}{\dfrac{n : \mathbb{N} \vdash \forall m^{\mathbb{N}}.0 + (n + m) == (n + m)}{\forall n, m : \mathbb{N}.0 + (n + m) == (n + m)}\,(\forall\text{-I})}\,(\forall\text{-I})}\,(\forall\text{-El})
$$

Existential Quantification

- We have seen before that we can identify in type theory existential quantification with the dependent product.
- With this identification, the **introduction rule** for the dependent product allows to form a proof of $\exists x^A.B$ from an element $a : A$ and a proof $p : B[x := a]$:

$$\frac{a : A \qquad p : B[x := a]}{\langle a, p\rangle : \exists x^A.B}\ (\times\text{-I})$$

- This is what is expressed by the **natural deduction introduction rule for $\exists$**:

$$\frac{a : A \qquad B[x := a]}{\exists x^A.B}\ (\exists\text{-I})$$

Example

- Assume we want to show $\forall n^{\mathbb{N}}.\exists m^{\mathbb{N}}.m > n$.
  - In order to prove this one assumes first $n : \mathbb{N}$.
  - Then one concludes $\mathrm{S}\ n : \mathbb{N}$ and $\mathrm{S}\ n > n$.
  - Using the introduction rule for $\exists$ one concludes $\exists m^{\mathbb{N}}.m > n$ under the assumption $n : \mathbb{N}$.
  - Using the introduction rule for $\forall$ one concludes $\forall n^{\mathbb{N}}.\exists m^{\mathbb{N}}.m > n$.

Example

- In natural deduction, this proof reads as follows:

$$\dfrac{\dfrac{n:\mathbb{N}\vdash n:\mathbb{N}}{n:\mathbb{N}\vdash \mathrm{S}\;n:\mathbb{N}}\,(\mathbb{N}\text{-}\mathrm{I}_{\mathrm{S}}) \qquad n:\mathbb{N}\vdash \mathrm{S}\;n>n}{\dfrac{n:\mathbb{N}\vdash \exists m^{\mathbb{N}}.m>n}{\forall n^{\mathbb{N}}.\exists m^{\mathbb{N}}.m>n}\,(\forall\text{-}\mathrm{I})}\,(\exists\text{-}\mathrm{I})$$

Existential Quantification (Cont.)

- The **elimination rule** for the dependent product allows to project a proof p of $\exists x^A.B$ to an element $\pi_0(p) : A$ and proof $\pi_1(p) : B[x := \pi_0(p)]$.
- This kind of rule works only if we have **explicit proofs**.
- From this we can derive a rule which is essentially that used in natural deduction (in which one doesn't have explicit proofs):
  - Assume:
    - $C : \mathrm{Set}$, which does not depend on $x : A$,
    - $p : \exists x^A.B$ and
    - $x : A, y : B \Rightarrow c : C$.
  - Then we have $\mathbf{c[x := \pi_0(p), y := \pi_1(p)] : C}$, **not depending on x:A or y:B**.

Existential Quantification (Cont.)

- Therefore the **rule in natural deduction** follows from the type theoretic rules:

$$\frac{\exists x^A.B \qquad x^A, B \vdash C}{C}\ (\exists\text{-El})$$

where C does not depend on $x : A$ and B.

- Here $x : A, B \vdash C$ means that from $x : A$ and assumption B we can derive C.
  - As in the introduction rule for natural deduction, $x : A$ is usually not mentioned explicitly, since the type structure there is very simple.

Example

- Assume we have shown $\forall n^{\mathbb{N}}.\exists m^{\mathbb{N}}.m > n \wedge \mathrm{Prime}(m)$.
- We want to show that for all n there exist two primes above it, i.e.

$$\forall n^{\mathbb{N}}.\exists m, k : \mathbb{N}.m > k \wedge k > n \wedge \wedge \mathrm{Prime}(m) \wedge \mathrm{Prime}(k) \ .$$

- We can derive this as follows:
  - Assume $n : \mathbb{N}$.
  - We have $\exists m^{\mathbb{N}}.m > n \wedge \mathrm{Prime}(m)$.
  - So assume $m : \mathbb{N}$ and $m > n \wedge \mathrm{Prime}(m)$.
  - We have as well $\exists k^{\mathbb{N}}.k > m \wedge \mathrm{Prime}(k)$.
  - So assume $k : \mathbb{N}$ and $k > m \wedge \mathrm{Prime}(k)$.

Example

- Then we can conclude

$$m > k \wedge k > n \wedge \mathrm{Prime}(m) \wedge \mathrm{Prime}(k)$$

and therefore as well

$$\exists m, k : \mathbb{N}.m > k \wedge k > n \wedge \mathrm{Prime}(m) \wedge \mathrm{Prime}(k)$$

- Now by $\exists$-elimination twice follows

$$n : \mathbb{N} \vdash \exists m, k : \mathbb{N}.m > k \wedge k > n \wedge \mathrm{Prime}(m) \wedge \mathrm{Prime}(k)$$

without assuming m, k as above.
- By $\forall$-introduction follows

$$\forall n^{\mathbb{N}}.\exists m, k : \mathbb{N}.m > k \wedge k > n \wedge \mathrm{Prime}(m) \wedge \mathrm{Prime}(k)$$

Example

- The formal proof in natural deduction is as follows (some of the premises can be shown easily in natural deduction):

Example

- First step: Under the global assumption

$$n : \mathbb{N}, m : \mathbb{N}, m > n \land \mathrm{Prime}(m), k : \mathbb{N}, k > m \land \mathrm{Prime}(k)$$

  we prove the following

$$\dfrac{m : \mathbb{N} \qquad \dfrac{k : \mathbb{N} \qquad m > n \land k > m \land \mathrm{Prime}(m) \land \mathrm{Prime}(k)}{\exists k^{\mathbb{N}}.m > n \land k > m \land \mathrm{Prime}(m) \land \mathrm{Prime}(k)}\,(\exists\text{-I})}{\exists m, k : \mathbb{N}.m > n \land k > m \land \mathrm{Prime}(m) \land \mathrm{Prime}(k)}\,(\exists\text{-I})$$

- So we have shown

$$n : \mathbb{N}, m : \mathbb{N}, m > n \land \mathrm{Prime}(m), k : \mathbb{N}, k > m \land \mathrm{Prime}(k) \vdash \\ \exists m, k : \mathbb{N}.m > n \land k > m \land \mathrm{Prime}(m) \land \mathrm{Prime}(k)$$

Example

- Second step: Under the assumption

$$n : \mathbb{N}, m : \mathbb{N}, m > n \land \mathrm{Prime}(m)$$

we can conclude

$$\exists k^{\mathbb{N}}.k > m \land \mathrm{Prime}(k)$$

and then conclude by $\exists$-elimination and Step 1

$$\frac{\exists k^{\mathbb{N}}.k>m\land\mathrm{Prime}(k) \qquad k{:}\mathbb{N},k>m\land\mathrm{Prime}(k)\vdash\exists m,k{:}\mathbb{N}.m>n\land k>m\land\mathrm{Prime}(m)\land\mathrm{Prime}(k)}{\exists m, k : \mathbb{N}.m > n \land k > m \land \mathrm{Prime}(m) \land \mathrm{Prime}(k)}\ (\exists\text{-I})$$

Example

- Third step: Again we can conclude

$$n : \mathbb{N} \vdash \exists m^{\mathbb{N}}.m > n \wedge \mathrm{Prime}(m)$$

and then conclude by $\exists$-elimination and Step 2

$$\dfrac{\dfrac{\begin{array}{c} n{:}\mathbb{N} \vdash \exists m^{\mathbb{N}}.m > n \wedge \mathrm{Prime}(m) \\ n{:}\mathbb{N}, m{:}\mathbb{N}, m > n \wedge \mathrm{Prime}(m) \vdash \exists m, k{:}\mathbb{N}.m > n \wedge k > m \wedge \mathrm{Prime}(m) \wedge \mathrm{Prime}(k) \end{array}}{n : \mathbb{N} \vdash \exists m, k : \mathbb{N}.m > n \wedge k > m \wedge \mathrm{Prime}(m) \wedge \mathrm{Prime}(k)}\ (\exists\text{-I})}{\forall n^{\mathbb{N}}.\exists m, k : \mathbb{N}.m > n \wedge k > m \wedge \mathrm{Prime}(m) \wedge \mathrm{Prime}(k)}\ (\forall\text{-I})$$

Construct. (or Intuit.) Logic

- From type theoretic proofs we can **directly extract programs**.
- For instance, if $p : \forall x^A.\exists y^B.C[x,y]$, then we have
  - for $x : A$ it follows $b := \pi_0(p\ x) : B$ and $\pi_1(p\ x) : C[x,b]$.
  - Therefore $f := \lambda x^A.\pi_0(p\ x)$ is a **function of type $\mathbf{A} \to \mathbf{B}$**, and we have
  $$\boldsymbol{\lambda x^A.\pi_1(p\ x) : \forall x^A.C[x, f\ x]}$$
  i.e. we have a proof that $\forall \mathbf{x^A}.\mathbf{C}[\mathbf{x},\mathbf{f}\ \mathbf{x}]$ **holds**.
  - Therefore, from a proof of $\forall x^A.\exists y^B.C[x,y]$, we can **extract a function**, which computes the y from the x.

Constructive Logic (Cont.)

- We can derive as well a function which **depending on p : A + B decides whether p= inl(a) or p = inr(b)**.
- Therefore we can decide, from a proof of a disjunction, **which of the disjuncts holds**.
- This has consequences due to the **undecidability of the Turing halting problem.**
  - Before continuing, I introduce briefly this result for those who haven't been in the module on computability theory.

Turing Machines

- A **Turing machine** (in short **TM**) is a program language which is according to **Church's thesis** universal:
    - Every computable function can be computed by a TM.
    - TMs can have one input string, no interaction, and have as output one output string.
        - Both these strings are usually interpreted as natural numbers.
    - To run a TM with no input means to run it with the empty input string.

Turing Complete Languages

- Any programming language, which can simulate a TM, shares this property and is called **Turing complete**.
  - Most standard programming languages, e.g. Java, Pascal, C, C++ are **Turing complete**.
  - **Agda**, restricted to termination checked programs, is **not Turing complete**.
    - No (decidable) language, which allows to write terminating programs only, can be Turing complete.

Turing Halting Problem

- The **Turing halting problem** is the question, whether a TM (with no inputs) terminates.
  - An essentially equivalent form is the question whether a TM with one input terminates.
- One can introduce a predicate **halts x** depending on a TM x (which can be represented as a string, as a natural number, or as a specific data type) expressing that "TM x holds, if given no inputs".
- Therefore the Turing halting problem is the question whether we can decide

$$\text{halts } x \vee \neg\text{halts } x \quad .$$

Unprovability in Type Theory

- It is known that the Turing halting problem is undecidable:
  - We cannot decide in a computable way for every x the Turing halting problem for x.
- Similarly we cannot decide whether a Java program with no input and no interaction terminates or not.
- Because of the undecidability of the Turing halting problem, the following formula is unprovable in Martin-Löf Type Theory and as well in Agda:

$$\forall x^{\mathrm{TM}}.\mathrm{halts}\ x \lor \neg \mathrm{halts}\ x \quad .$$

  - Here TM is a data type which allows to encode all TM in a standard way.

Unprovability in Constructive Logic

- If we could prove it, we could get a function, which determines for $x : \mathrm{TM}$ whether $\mathrm{halts}\ x$ or not.
- But such a function needs to be computable, and such a computable function doesn't exist.

Constructive Logic (Cont.)

- In classical logic we **can prove the above**, since we can derive $\mathbf{A} \lor \neg\mathbf{A}$ (tertium non datur) for any formula A.
- In type theory, this law **cannot hold**, unless we don't want that all programs can be evaluated.
  - The logic of type theory is **intuitionistic (constructive) logic**, in which $A \lor \neg A$ and $\neg\neg A \supset A$ are in general not provable for all formulae A.
- Jump over remaining slides

Constructive Logic (Cont.)

- In **classical logic**,
  - $\exists x^{A}.B$ **is equivalent to** $\neg\forall x^{A}.\neg B$,
  - $A \lor B$ **is equivalent to** $\neg(\neg A \land \neg B)$.
- If we take decidable atomic formulae only and

$$\begin{array}{llll} \text{replace} & \exists x^{A}.B & \text{by} & \neg\forall x^{A}.\neg B \\ \text{replace} & A \lor B & \text{by} & \neg(\neg A \land \neg B) \end{array}$$

  then **all formulae provable in classical logic are derivable** in type theory.
  - All we need is $\neg\neg A \supset A$, which can be shown for all formulae built from decidable atomic formulae using $\neg$, $\supset$, $\land$, $\forall$.

Constructive Logic (Cont.)

- Especially, the tertium non datur formula

$$A \vee \neg A$$

translates into

$$\neg(\neg A \wedge \neg\neg A)$$

which trivially holds, since $\neg A$ and $\neg\neg A$ implies $\perp$.

- In this sense, **type theory contains classical logic**.

Weak vs. Strong Disjunction and Exist-Quantification

- But type theory is **richer**, since it has as well so called **strong disjunction and existential quantification**.
  - **Strong disjunction** and **strong existential quantification** are the formulae
    $$A \vee B \text{ and } \exists x^A.B$$
    whereas **weak disjunction** and **weak existential quantification** are the formulae
    $$\neg(\neg A \wedge \neg B) \text{ and } \neg\forall x^A.\neg B$$

Weak vs. Strong Disjunction and Exist-Quantification

- From a proof $p : \exists x^A.B$ we can extract an element x of A s.t. B holds, namely

  $$\pi_0(x)$$

  This is in general **not possible for weak existential quantification.**
- From a proof $p : A \vee B$ we can determine which one of A or B holds (the other disjunct might hold as well).
  From a proof of **weak disjunction** this is in general **not possible.**

Constructive Logic (Cont.)

- **Remark:** One can always obtain classical logic in Agda for arbitrary formulae by **postulating** tertium non datur for the formulae for which one needs it:

  postulate p : A ∨ ¬A

- Jump over the following proofs.

Constructive Logic (Cont.)

- Proof (using classical logic) of

$$\exists \mathbf{x}^{\mathbf{A}}.\mathbf{B} \leftrightarrow (\neg\forall \mathbf{x}^{\mathbf{A}}.\neg\mathbf{B}) \quad :$$

  - We have classically:

$$\neg\neg\mathbf{A} \supset \mathbf{A} \quad :$$

    - If A is true, then $\neg\neg A \supset A$ holds.
    - If A is false, then $\neg\neg A$ is false, therefore $\neg\neg A \supset A$ holds.

Constructive Logic (Cont.)

- We show intuitionistically $\neg\exists \mathbf{x}^{\mathbf{A}}.\mathbf{B} \leftrightarrow \forall \mathbf{x}^{\mathbf{A}}.\neg\mathbf{B}$:
  - Assume $\neg\exists x^A.B$, $x : A$ and show $\neg B$.
    If we had B, then we had $\exists x^A.B$, contradicting $\neg\exists x^A.B$. Therefore $\neg B$.
  - Assume $\forall x^A.\neg B$. Show $\neg\exists x^A.B$:
    Assume $\exists x^A.B$. Assume x s.t. B holds.
    By $\forall x^A.\neg B$ we get $\neg B$, therefore a contradiction.
- Now it follows (classically):

$$(\exists \mathbf{x}^{\mathbf{A}}.\mathbf{B}) \leftrightarrow (\neg\neg\exists \mathbf{x}^{\mathbf{A}}.\mathbf{B}) \leftrightarrow (\neg\forall \mathbf{x}^{\mathbf{A}}.\neg\mathbf{B})$$

Constructive Logic (Cont.)

- Proof of

$$\mathbf{A} \lor \mathbf{B} \leftrightarrow \neg(\neg\mathbf{A} \land \neg\mathbf{B}) \quad :$$

  - We show intuitionistically $\neg(\mathbf{A} \lor \mathbf{B}) \leftrightarrow (\neg\mathbf{A} \land \neg\mathbf{B}) \quad :$
    - Assume $\neg(A \lor B)$. If A then $A \lor B$, a contradiction, therefore $\neg A$. Similarly we get $\neg B$, therefore $\neg A \land \neg B$.
    - Assume $\neg A \land \neg B$, show $\neg(A \lor B)$. Assume $A \lor B$. If A then a contradiction with $\neg A$, similarly with B.
  - Now it follows (classically):

$$(\mathbf{A} \lor \mathbf{B}) \leftrightarrow \neg\neg(\mathbf{A} \lor \mathbf{B}) \leftrightarrow \neg(\neg\mathbf{A} \land \neg\mathbf{B})$$

Classical Logic for $\exists$, $\vee$-free Formulae

- We show that for formulas A built from $\neg$, $\supset$, $\wedge$, $\forall$ and decidable prime formulae we have
$$\neg\neg A \supset A \ .$$
- The formula $\neg\neg A \supset A$ is called **stability for A**.
- This is done by induction over the buildup of these formulae.

Classical Logic for $\exists$, $\vee$-free Formulae

- Case $A \equiv \mathrm{Atom}\ c$.
  - We make case distinction on c.
  - If $c = \mathrm{tt}$, then we have $A \equiv \top$, A is provable, therefore as well $\neg\neg A \supset A$.
  - If $c = \mathrm{ff}$, then we have $A \equiv \bot$.
    - Assume $\neg\neg A \equiv (\bot \supset \bot) \supset \bot$.
    - $\bot \supset \bot$ is provable.
    - Therefore we obtain $\bot$, which is A.
    - So we have
$$\neg\neg A \vdash A$$
and obtain
$$\neg\neg A \supset A \ .$$

Classical Logic for $\exists$, $\vee$-free Formulae

- Case $A \equiv B \supset C$, and assume we have already shown stability for B and C.
- We have to show that from $\neg\neg A$ we obtain A, which is $B \supset C$.
- So assume $\neg\neg A$, B and show C.
- We show $\neg\neg C$, then by stability of C we obtain C.
- $\neg\neg C \equiv \neg C \supset \bot$.
- Therefore assume $\neg C$ and show $\bot$.
  - We show $\neg A$ which is $A \supset \bot$.
    - So assume A and show $\bot$. $A \equiv B \supset C$, therefore by B we get C, and by $\neg C$ therefore $\bot$.
  - By $\neg\neg A$, which is $\neg A \supset \bot$, we get therefore $\bot$, which completes the proof for this case.

Classical Logic for $\exists$, $\vee$-free Formulae

- Case $A \equiv B \wedge C$, and assume we have already shown stability for B and C.
- Assume $\neg\neg A$ and show A.
  - We show $\neg\neg B$, which implies by the stability of B that B holds.
    - Since $\neg\neg B \equiv \neg B \supset \bot$, we assume $\neg B$ and have to show $\bot$.
    - We show $\neg A$, i.e. show that A implies $\bot$:

      Assume A, which is $B \wedge C$. Then we get B, and by $\neg B$ therefore $\bot$.
    - By $\neg\neg A$ we obtain $\bot$.
  - Therefore we have shown B.
  - A similar proof shows C, and therefore we get $B \wedge C$, i.e. A.

Classical Logic for $\exists$, $\vee$-free Formulae

- Case $A \equiv \forall x^B.C$, and assume we have already shown stability for C.
- Assume $\neg\neg A$ and show A.
- So assume $x : B$, and show C.
- We show $\neg\neg C$, which by the stability of C implies C.
  - So assume $\neg C$ and show $\perp$.
  - We show $\neg A$.
    - Assume A, which is $\forall x^B.C$.
    - Then we obtain C, and by $\neg C$ therefore $\perp$.
  - By $\neg\neg A$ we therefore get $\perp$, and are done.

Classical Logic for $\exists$, $\vee$-free Formulae

- Case $A \equiv \neg B$, and we have stability for B.
- $\neg B \equiv B \supset \bot$.
- $\bot \equiv \bot = \mathrm{Atom\ false}$.
- By stability for decidable prime formulae we get stability for $\bot$.
- Together with the stability for B we obtain by case $\supset$ the stability for $B \supset \bot \equiv \neg B$.

6 (g) The Set of Natural Numbers

- The set $\mathbb{N}$ is the type theoretic representation of the set $\mathbb{N} := \{0, 1, 2, \ldots, \}$.
- $\mathbb{N}$ can be generated by
  - starting with the empty set,
  - adding 0 to it, and
  - adding, whenever we have x in it $x + 1$ to it.

The Set of Natural Numbers (Cont.)

- Let S be a type theoretic notation for the operation $x \mapsto x + 1$.
- Then the type theoretic rules are

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

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

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

Primitive Recursion

- **Primitive Recursion expresses:**
  Assume we have
  - $a : \mathbb{N}$.
  - and, if $n : \mathbb{N}$, $x : \mathbb{N}$ then $g\ n\ x : \mathbb{N}$.

  Then we can define f $: \mathbb{N} \to \mathbb{N}$, s.t.
  - $f\ 0 = a$,
  - $f\ (\mathrm{S}\ n) = g\ n\ (f\ n)$.

Primitive Recursion (Cont.)

- The **computation of f n** proceeds now as follows:
  - Compute n.
  - If $n = 0$, then the result is a.
  - Otherwise $n = \mathrm{S}\ n'$.
    - We assume that we have determined already how to compute $f\ n'$.
    - Now $f\ n$ reduces to $g\ n'\ (f\ n')$.
    - $g\ n'\ (f\ n')$ can be computed, since we know how to compute

      - g
      - $f\ n'$.

Example

- The function $f : \mathbb{N} \to \mathbb{N}$ with $f\ n = 2 \cdot n$ can be defined **primitive recursively** by:
  - $f\ 0 = 0$.
  - $f\ (\mathrm{S}\ n) = \mathrm{S}\ (\mathrm{S}\ (f\ n))$.
- Therefore take in the definition above:
  - $a = 0$,
  - $g\ n\ x = \mathrm{S}\ (\mathrm{S}\ x)$.

Generalised Primitive Recursion

- We can **generalise primitive recursion** as follows:
  - First we can **replace the range of f by an arbitrary set C**
    - i.e. we allow for any set C

$$f : \mathbb{N} \to C$$

  - Further, C can now **depend on** $\mathbb{N}$.
- We obtain the following set of rules:

Rules for the Natural Numbers

Formation Rule

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

Introduction Rules

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

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

Rules for the Natural Numbers

Elimination Rule

$$\frac{\begin{array}{c} C : \mathbb{N} \to \mathrm{Set} \\ a : C\ 0 \\ g : (x : \mathbb{N}) \to C\ x \to C\ (\mathrm{S}\ x) \\ n : \mathbb{N} \end{array}}{\mathrm{P}\ C\ a\ g\ n : C\ n}\ (\mathbb{N}\text{-El})$$

Equality Rules

$$\begin{array}{rcll} \mathrm{P}\ C\ a\ g\ 0 & = & a & (\mathbb{N}\text{-Eq}_0) \\ \mathrm{P}\ C\ a\ g\ (\mathrm{S}\ n) & = & g\ n\ (\mathrm{P}\ C\ a\ g\ n) & (\mathbb{N}\text{-Eq}_\mathrm{S}) \end{array}$$

Additionally we have the **Equality versions** of the formation-, introduction- and elimination-rules.

Jump over Elimination into Type

Elimination into Type

- In order to define predicates on the natural numbers by prim. recursion, we need sometimes elimination into type:
  Strong elimination Rule

$$\frac{\begin{array}{c} n : \mathbb{N} \Rightarrow C[n] : \mathrm{Type} \\ a : C[0] \\ g : (x : \mathbb{N}) \to C[x] \to C[\mathrm{S}\ x] \\ n : \mathbb{N} \end{array}}{\mathrm{P}_C^{\mathrm{Type}}\ a\ g\ n : C[n]} \ (\mathbb{N}\text{-El}^{\mathrm{Type}})$$

Strong Equality Rules

$$\begin{array}{rcll} \mathrm{P}_C^{\mathrm{Type}}\ a\ g\ 0 & = & a & (\mathbb{N}\text{-Eq}_0^{\mathrm{Type}}) \\ \mathrm{P}_C^{\mathrm{Type}}\ a\ g\ (\mathrm{S}\ n) & = & g\ n\ (\mathrm{P}_C^{\mathrm{Type}}\ a\ g\ n) & (\mathbb{N}\text{-Eq}_{\mathrm{S}}^{\mathrm{Type}}) \end{array}$$

Rules for the Natural Numbers

- Note that if we define in the elimination rule $f := \mathrm{P}\ C\ a\ g$ (which is η-equal to $\lambda n^{\mathbb{N}}.\mathrm{P}\ C\ g\ a\ n$) then
  - The conclusion of the elimination rule reads:

$$f\ n : C\ n$$

    which means that

$$f : (n : \mathbb{N}) \to C\ n\ .$$

  - The equality rules read:

$$\begin{array}{rcl} f\ 0 & = & a \\ f\ (\mathrm{S}\ n) & = & g\ n\ (f\ n) \end{array}$$

Logical Framework Rules for $\mathbb{N}$

- The more compact notation is:
  - $\mathbb{N} : \mathrm{Set}$,
  - $0 : \mathbb{N}$,
  - $S : \mathbb{N} \to \mathbb{N}$,
  - $$\begin{aligned}\mathrm{P} \;\; : & \; (C : \mathbb{N} \to \mathrm{Set}) \\ & \to C\ 0 \\ & \quad \to ((x : \mathbb{N}) \to C\ x \to C\ (S\ x)) \\ & \qquad \to (n : \mathbb{N}) \\ & \qquad\quad \to C\ n \ \ .\end{aligned}$$
  - The same equality rules as before.

Natural Numbers in Agda

- $\mathbb{N}$ is defined using **data**:

$$
\begin{array}{l}
\text{data } \mathbb{N} : \text{Set where} \\
\quad \text{Z} : \mathbb{N} \\
\quad \text{S} : \mathbb{N} \to \mathbb{N}
\end{array}
$$

  Here $\mathbb{N}$ can be typed in using Leim as \Bbb{N}.
  (We cannot use 0 for zero, since this denotes the builtin native natural number 0 in Agda).

- Therefore we have

$$
\begin{array}{lcl}
\text{Z} & : & \mathbb{N} \\
\text{S} & : & \mathbb{N} \to \mathbb{N}
\end{array}
$$

Elimination Rules for $\mathbb{N}$ in Agda

- Elimination is represented in Agda as before via case distinction.
- Assume we want to define

$$\begin{array}{l} f : (n : \mathbb{N}) \to A \\ f\ n = \{!\ \ !\} \end{array}$$

  - A possibly depending on n,
- Then we can distinguish the cases $n = \mathrm{Z}$ and $n = \mathrm{S}\ m$ and obtain:

$$\begin{array}{llll} f : (n : \mathbb{N}) \to A & & & \\ f & \mathrm{Z} & = & \{!\ \ !\} \\ f & (\mathrm{S}\ n) & = & \{!\ \ !\} \end{array}$$

Elimination Rules for $\mathbb{N}$ in Agda

- For solving the goals, we can now **make use of f**.
  That will be **accepted by the type checker.**
- However, if we use of full f, and then type check the file, the termination checker will complain, and we obtain for instance

$$
\begin{array}{l}
\boxed{f} : (n : \mathbb{N}) \to A \\
f\ n = \boxed{f}\ n
\end{array}
$$

exampleNat1.agda

Elimination Rules for $\mathbb{N}$ in Agda

- If we, in

$$
\begin{array}{llcl}
g : (n : \mathbb{N}) \to A & & & \\
g & \mathrm{Z} & = & \{!\ \ !\} \\
g & (\mathrm{S}\ n) & = & \{!\ \ !\}
\end{array}
$$

  - **do not make use of g when defining** $g\ \mathrm{Z}$ and
  - **only use of g n when defining** $g\ (\mathrm{S}\ n)$

  then the termination check succeeds (once the definition is complete).

Elimination Rules for $\mathbb{N}$ in Agda

- If we haven't completed the definition of g, the termination checker might complain, as long as not all details are known.
  - For instance, if we have the following we get an error:

$$
\begin{array}{lll}
\boxed{g} : \mathbb{N} \to \mathbb{N} & & \\
g \quad Z & = & Z \\
g \quad (S\ n) & = & \boxed{g}\ \{!\ \ !\}
\end{array}
$$

  - If we complete it as follows the error vanishes (one might need to load the agda code again):

$$
\begin{array}{lll}
g : \mathbb{N} \to \mathbb{N} & & \\
g \quad Z & = & Z \\
g \quad (S\ n) & = & g\ n
\end{array}
$$

Elimination Rules for $\mathbb{N}$ in Agda

- If **check-termination succeeds**, the definition should be **correct.**
  - (The lecturer hasn't checked the algorithm).
- However, **if check-termination fails**, the **definition might still be correct**.

  Jump over Limitations of Termination Checker.

Power of Termination Check

- The following definition of the **Fibonacci numbers** can't be defined this way directly using the rules of type theory, but it **can be defined in Agda** as follows and **check-termination accepts it**:
  (one := S Z):

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\mathrm{fib} : \mathbb{N} \to \mathbb{N}} \\
\mathrm{fib} & \mathrm{Z} & = & \mathrm{one} \\
\mathrm{fib} & (\mathrm{S}\ \mathrm{Z}) & = & \mathrm{one} \\
\mathrm{fib} & (\mathrm{S}\ (\mathrm{S}\ n)) & = & \mathrm{fib}\ n + \mathrm{fib}\ (\mathrm{S}\ n)
\end{array}
$$

  fib1.agda

Limitations of Termination Checker

- Assume we define the **predecessor function**

$$
\begin{array}{lll}
\mathrm{pred} : \mathbb{N} \to \mathbb{N} & & \\
\mathrm{pred} \quad \mathrm{Z} & = & \mathrm{Z} \\
\mathrm{pred} \quad (\mathrm{S}\ n) & = & n
\end{array}
$$

i.e.

$$
\mathrm{pred}(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise.} \end{cases}
$$

Limitations of Termination Checker

- Then the function

$$\begin{array}{llll} \boxed{f} : \mathbb{N} \to \mathbb{N} & & & \\ f & \mathrm{Z} & = & \mathrm{Z} \\ f & (\mathrm{S}\ n) & = & \boxed{f}\ (\mathrm{pred}\ n) \end{array}$$

terminates always
  - (it returns for all $n : \mathbb{N}$ the value Z).
- However, **check-termination fails**.
  terminationnat1.agda

Limitations of Termination Checker

- Because of the **undecidability of the Turing halting problem**
  - it is undecidable, whether a recursively defined function terminates or not,
- therefore there is no **extension of check-termination**, **which accepts exactly all in Agda definable functions, which terminate for all inputs**.

Example: Addition

- Definition of $+$ in Agda:

$$
\begin{array}{l}
\mathrm{infixr}\ 10\ \_+\_ \\
\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\begin{array}{lllll}
n & + & \mathrm{Z} & = & n \\
n & + & \mathrm{S}\ m & = & \mathrm{S}\,(n+m)
\end{array}
\end{array}
$$

- The definitition is correct, since when defining $n + \mathrm{S}\ m$, $n + m$ is defined before $n + \mathrm{S}\ m$.
- Because of the line

$$\mathrm{infixr}\ 10\ \_+\_\ ,$$

$n + m + k$ is interpreted as $n + (m + k)$.

Example: Multiplication

- Definition

$$\begin{array}{l} \mathrm{infixr}\ 20\ \_*\_ \\ \_*\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ \begin{array}{lllll} n & * & \mathrm{Z} & = & \mathrm{Z} \\ n & * & \mathrm{S}\ m & = & n * m + n \end{array} \end{array}$$

- Because of the line

$$\mathrm{infixr}\ 20\ \_*\_\ ,$$

  $\_*\_$ binds more than $\_+\_$
  - Remember we had $\mathrm{infixr}\ 10\ \_+\_$.
- We can use in the definition of $\_*\_$ $+$, and can refer in case of $n * \mathrm{S}\ m$ to $n * m$, which is defined before $n * \mathrm{S}\ m$.

Equality on $\mathbb{N}$

- We can define a Boolean valued equality on $\mathbb{N}$ as follows:

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\_{==}\mathrm{Bool}\_ : \mathbb{N} \to \mathbb{N} \to \mathrm{Bool}} \\
\mathrm{Z} & {==}\mathrm{Bool} & \mathrm{Z} & = & \mathrm{tt} \\
\mathrm{S}\ n & {==}\mathrm{Bool} & \mathrm{S}\ m & = & n \ {==}\mathrm{Bool} \ m \\
\_ & {==}\mathrm{Bool} & \_ & = & \mathrm{ff}
\end{array}
$$

- Note that the third case expresses: in all other cases (i.e. when defining $n \,{==}\mathrm{Bool}\, m$ and neither both n, m are Z nor both are of the form $\mathrm{S}\ \_$) we obtain the result ff.

Equality on $\mathbb{N}$

- Then we can define equality $\_==\_$ on $\mathbb{N}$ as follows

$$\begin{aligned} \_==\_ &: \mathbb{N} \to \mathbb{N} \to \mathrm{Set} \\ n == m &= \mathrm{Atom}\,(n \;\; ==\mathrm{Bool} \;\; m) \end{aligned}$$

Equality on $\mathbb{N}$ (Cont.)

- Alternatively we could have defined $\_==\_$ directly (this uses in fact large elimination on $\mathbb{N}$):

$$
\begin{array}{llllll}
\multicolumn{6}{l}{\_==\_ : \mathbb{N} \to \mathbb{N} \to \mathrm{Set}} \\
\mathrm{Z} & == & \mathrm{Z} & = & \top \\
\mathrm{S}\ n & == & \mathrm{S}\ m & = & n == m \\
\_ & == & \_ & = & \bot
\end{array}
$$

nat1.agda

Reflexivity of ==

- **Reflexivity** of == is the formula:

$$\forall n^{\mathbb{N}}. n == n$$

- **Type theoretically** this means that we have to prove

$$\begin{aligned}&\mathrm{refl} : \mathrm{Refl}\\&\mathrm{refl} = \{!\ \ !\}\end{aligned}$$

where

$$\begin{aligned}&\mathrm{Refl} : \mathrm{Set}\\&\mathrm{Refl} = (n : \mathbb{N}) \to n == n\end{aligned}$$

Reflexivity of ==

$\mathrm{Refl} : \mathrm{Set}$
$\mathrm{Refl} = (n : \mathbb{N}) \to n == n$

$\mathrm{refl} : \mathrm{Refl}$
$\mathrm{refl} = \{!\ \ !\}$

- Since refl is an element of a function type, we replace the definition of refl by

$$\begin{array}{l} \mathrm{refl} : \mathrm{Refl} \\ \mathrm{refl}\ n = \{!\ \ !\} \end{array}$$

where the type of the goal is $n == n$.

Reflexivity of == (Cont.)

$\mathrm{Refl} : \mathrm{Set}$
$\mathrm{Refl} = (n : \mathbb{N}) \to n == n$

$\mathrm{refl} : \mathrm{Refl}$
$\mathrm{refl}\ n = \{!\ \ !\}$

- This can now be shown using **pattern matching**:

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\mathrm{refl} : \mathrm{Refl}} \\
\mathrm{refl} & \mathrm{Z} & = & \{!\ \ !\} \\
\mathrm{refl} & (\mathrm{S}\ n) & = & \{!\ \ !\}
\end{array}
$$

Reflexivity of $==$ (Cont.)

- In order to prove $\mathrm{refl}\ \mathrm{Z}$, we observe

$$\begin{array}{rcl}(\mathrm{Z} == \mathrm{Z}) & = & \mathrm{Atom}\,(\mathrm{Z}\ \ ==_{\mathrm{Bool}}\ \ \mathrm{Z}) \\ & = & \mathrm{Atom}\ \mathrm{tt} \\ & = & \top\end{array}$$

  - Therefore the goal can be solved by taking $\mathrm{true} : \top$.

Reflexivity of == (Cont.)

- In order to prove $\mathrm{refl}\ (\mathrm{S}\ n)$, we observe

$$\begin{array}{rcl}(\mathrm{S}\ n == \mathrm{S}\ n) & = & \mathrm{Atom}\ (\mathrm{S}\ n\ ==_{\mathrm{Bool}}\ \mathrm{S}\ n) \\ & = & \mathrm{Atom}\ (n\ ==_{\mathrm{Bool}}\ n) \\ & = & (n == n)\end{array}$$

  - Therefore the goal can be solved by taking $\mathrm{refl}\ n : (n == n)$.

Reflexivity of $==$ (Cont.)

- The complete proof is as follows:

$$
\begin{array}{lll}
\mathrm{refl} : \mathrm{Refl} & & \\
\mathrm{refl} \quad \mathrm{Z} & = & \mathrm{true} \\
\mathrm{refl} \quad (\mathrm{S}\ n) & = & \mathrm{refl}\ n
\end{array}
$$

- Note that this is not a black hole recursion, since in the second equation $\mathrm{refl}\ n$ is defined before $\mathrm{refl}\ (\mathrm{S}\ n)$.
  reflnat.agda

Symmetry of ==

- **Symmetry** of == is the formula:

$$\forall n, m : \mathbb{N}. n == m \to m == n$$

- **Type theoretically** this means that we have to prove

$$\begin{array}{l}\mathrm{Sym} : \mathrm{Set} \\ \mathrm{Sym} = (n\ \ m : \mathbb{N}) \to n == m \to m == n\end{array}$$

In Agda this is shown by defining

$$\begin{array}{l}\mathrm{sym} : \mathrm{Sym} \\ \mathrm{sym}\ n\ \ m\ \ nm = \{!\ \ !\}\end{array}$$

Symmetry of == (Cont.)

$\mathrm{Sym} : \mathrm{Set}$
$\mathrm{Sym} = (n\ m : \mathbb{N}) \to n == m \to m == n$

- This can now be shown using **case distinction** on both n and m:

$$
\begin{array}{llllll}
\mathrm{sym} : \mathrm{Sym} \\
\mathrm{sym} & \mathrm{Z} & \mathrm{Z} & nm & = & \{!\ \ !\} \\
\mathrm{sym} & \mathrm{Z} & (\mathrm{S}\ m) & nm & = & \{!\ \ !\} \\
\mathrm{sym} & (\mathrm{S}\ n) & \mathrm{Z} & nm & = & \{!\ \ !\} \\
\mathrm{sym} & (\mathrm{S}\ n) & (\mathrm{S}\ m) & nm & = & \{!\ \ !\}
\end{array}
$$

- For convenience we spell out the type of sym in the following.

Symmetry of $==$ (Cont.)

$$
\begin{array}{llllll}
\mathrm{sym} : (n\ m : \mathbb{N}) \to n == m \to m == n \\
\mathrm{sym} & \mathrm{Z} & \mathrm{Z} & nm & = & \{!\ \ !\} \\
\mathrm{sym} & \mathrm{Z} & (\mathrm{S}\ m) & nm & = & \{!\ \ !\} \\
\mathrm{sym} & (\mathrm{S}\ n) & \mathrm{Z} & nm & = & \{!\ \ !\} \\
\mathrm{sym} & (\mathrm{S}\ n) & (\mathrm{S}\ m) & nm & = & \{!\ \ !\}
\end{array}
$$

- In case $\mathrm{sym}\ \mathrm{Z}\ \mathrm{Z}\ nm$, the goal is

$$(\mathrm{Z} == \mathrm{Z}) = \top$$

  which can be solved by using true.
  - The argument nm is irrelevant and can be replaced by _.

Symmetry of == (Cont.)

$$
\begin{array}{llllll}
\mathrm{sym} : (n\ m : \mathbb{N}) \to n == m \to m == n \\
\mathrm{sym} & \mathrm{Z} & \mathrm{Z} & \_ & = & \mathrm{true} \\
\mathrm{sym} & \mathrm{Z} & (\mathrm{S}\ m) & nm & = & \{!\ \ !\} \\
\mathrm{sym} & (\mathrm{S}\ n) & \mathrm{Z} & nm & = & \{!\ \ !\} \\
\mathrm{sym} & (\mathrm{S}\ n) & (\mathrm{S}\ m) & nm & = & \{!\ \ !\}
\end{array}
$$

- In case $\mathrm{sym}\ \mathrm{Z}\ (\mathrm{S}\ m)\ nm$, we have

$$nm : \mathrm{Z} == \mathrm{S}\ m = \perp$$

so there is no element in nm, we can solve it as

$$\mathrm{sym}\ \mathrm{Z}\ (\mathrm{S}\ m)\ ()$$

Symmetry of $==$ (Cont.)

$$
\begin{array}{llllll}
\mathrm{sym} : (n\ m : \mathbb{N}) \to n == m \to m == n \\
\mathrm{sym} & \mathrm{Z} & \mathrm{Z} & \_ & = & \mathrm{true} \\
\mathrm{sym} & \mathrm{Z} & (\mathrm{S}\ m) & () & & \\
\mathrm{sym} & (\mathrm{S}\ n) & \mathrm{Z} & nm & = & \{!\ \ !\} \\
\mathrm{sym} & (\mathrm{S}\ n) & (\mathrm{S}\ m) & nm & = & \{!\ \ !\}
\end{array}
$$

- In case $\mathrm{sym}\ (\mathrm{S}\ n)\ \mathrm{Z}\ nm$, we have

$$nm : \mathrm{S}\ m == \mathrm{Z} = \bot$$

  so there is no element in nm, we can solve it as

$$\mathrm{sym}\ (\mathrm{S}\ n)\ \mathrm{Z}\ ()$$

Symmetry of == (Cont.)

$$
\begin{array}{llllll}
\mathrm{sym} : (n\ m : \mathbb{N}) \to n == m \to m == n \\
\mathrm{sym} & \mathrm{Z} & \mathrm{Z} & \_ & = & \mathrm{true} \\
\mathrm{sym} & \mathrm{Z} & (\mathrm{S}\ m) & () \\
\mathrm{sym} & (\mathrm{S}\ n) & \mathrm{Z} & () \\
\mathrm{sym} & (\mathrm{S}\ n) & (\mathrm{S}\ m) & nm & = & \{!\ \ !\}
\end{array}
$$

- In case $\mathrm{sym}\ (\mathrm{S}\ n)\ (\mathrm{S}\ m)\ nm$, we have that the type of the goal is

$$(\mathrm{S}\ m == \mathrm{S}\ n) = (m == n)$$

- This goal can be solved by

$$\mathrm{sym}\ n\ m\ nm : m == n$$

which is type correct since $nm : (\mathrm{S}\ n == \mathrm{S}\ m) = (n == m)$

Symmetry of $==$ (Cont.)

- The complete proof is as follows:

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\mathrm{sym} : (n\ \ m : \mathbb{N}) \to n == m \to m == n} \\
\mathrm{sym} & \mathrm{Z} & \mathrm{Z} & \_ & = \ \mathrm{true} \\
\mathrm{sym} & \mathrm{Z} & (\mathrm{S}\ m) & () & \\
\mathrm{sym} & (\mathrm{S}\ n) & \mathrm{Z} & () & \\
\mathrm{sym} & (\mathrm{S}\ n) & (\mathrm{S}\ m) & nm & = \ \mathrm{sym}\ n\ m\ nm
\end{array}
$$

- Note that this code termination checks, since in the last equation $\mathrm{sym}\ n\ m\ nm$ is defined before $\mathrm{sym}\ (\mathrm{S}\ n)\ (\mathrm{S}\ m)\ nm$.
  symnat.agda

Symmetry of == (Cont.)

$$
\begin{array}{lllll}
\mathrm{sym} : (n\ m : \mathbb{N}) \to n == m \to m == n \\
\mathrm{sym} & \mathrm{Z} & \mathrm{Z} & \_ & = \mathrm{true} \\
\mathrm{sym} & \mathrm{Z} & (\mathrm{S}\ m) & () & \\
\mathrm{sym} & (\mathrm{S}\ n) & \mathrm{Z} & () & \\
\mathrm{sym} & (\mathrm{S}\ n) & (\mathrm{S}\ m) & nm & = \mathrm{sym}\ n\ m\ nm
\end{array}
$$

- In the cases

$$
\begin{array}{lllll}
\mathrm{sym} & \mathrm{Z} & (\mathrm{S}\ m) & nm & \text{and} \\
\mathrm{sym} & (\mathrm{S}\ n) & \mathrm{Z} & nm &
\end{array}
$$

  we have that nm is an element of $\perp$, and the goal is $\perp$.
- So we can, instead of using empty case distinction on nm, return the proof nm and obtain the following:

Symmetry of == (Cont.)

$$
\begin{array}{llllll}
\multicolumn{6}{l}{\mathrm{sym} : (n\ m : \mathbb{N}) \to n == m \to m == n} \\
\mathrm{sym} & \mathrm{Z} & \mathrm{Z} & \_ & = & \mathrm{true} \\
\mathrm{sym} & \mathrm{Z} & (\mathrm{S}\ m) & nm & = & nm \\
\mathrm{sym} & (\mathrm{S}\ n) & \mathrm{Z} & nm & = & nm \\
\mathrm{sym} & (\mathrm{S}\ n) & (\mathrm{S}\ m) & nm & = & \mathrm{sym}\ n\ m\ nm
\end{array}
$$

symnat2.agda

Example: $<$ on $\mathbb{N}$

- The following introduces $<$ on $\mathbb{N}$:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\_{<}\mathrm{Bool}\_ : \mathbb{N} \to \mathbb{N} \to \mathrm{Bool}} \\
\_ & {<}\mathrm{Bool} \;\; \mathrm{Z} & = & \mathrm{ff} \\
\mathrm{Z} & {<}\mathrm{Bool} \;\; \mathrm{S}\ m & = & \mathrm{tt} \\
\mathrm{S}\ n & {<}\mathrm{Bool} \;\; \mathrm{S}\ m & = & n \;\; {<}\mathrm{Bool} \;\; m
\end{array}
$$

$$
\begin{array}{l}
\_{<}\_ : \mathbb{N} \to \mathbb{N} \to \mathrm{Set} \\
n < m = \mathrm{Atom}\,(n \;\; {<}\mathrm{Bool} \;\; m)
\end{array}
$$

lessnat1.agda

Example: $<$ on $\mathbb{N}$

- Alternatively, we can define $<$ using large elimination:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\_{<}\_ : \mathbb{N} \to \mathbb{N} \to \mathrm{Set}} \\
\_ & < \; \mathrm{Z} & = & \bot \\
\mathrm{Z} & < \; \mathrm{S}\ m & = & \top \\
\mathrm{S}\ n & < \; \mathrm{S}\ m & = & n < m
\end{array}
$$

lessnat2.agda

Example: Tuples of Length n

- We define tuples (or vectors) of length n in Agda:

$$\begin{array}{l} \mathrm{data\ Nil} : \mathrm{Set\ where} \\ \quad [\,] : \mathrm{Nil} \\ \mathrm{data\ Cons}\ (A\ B : \mathrm{Set}) : \mathrm{Set\ where} \\ \quad \_{::}\_ : A \to B \to \mathrm{Cons}\ A\ B \end{array}$$

($\mathrm{Cons}\ A\ B$ is just $A \times B$ with a convenient name for the constructor).

- Now we can define

$$\begin{array}{llll} \multicolumn{4}{l}{\mathrm{Tuple} : \mathrm{Set} \to \mathbb{N} \to \mathrm{Set}} \\ \mathrm{Tuple} & A & \mathrm{Z} & = \ \mathrm{Nil} \\ \mathrm{Tuple} & A & (\mathrm{S}\ n) & = \ \mathrm{Cons}\ A\ (\mathrm{Tuple}\ A\ n) \end{array}$$

Tuples of Length n

- Therefore,

$$\mathrm{Tuple}\ A\ n = \underbrace{\mathrm{Cons}\ A\ (\mathrm{Cons}\ A\ \cdots (\mathrm{Cons}\ A}_{n \text{ times}}\ \mathrm{Nil}) \cdots)) \ .$$

- The elements of $\mathrm{Tuple}\ A\ n$ are

$$a_1\ :: (a_2\ \cdots (a_n :: [\,]) \cdots)$$

  for elements $a_1, \ldots, a_n$ of A.
  If we add $\mathrm{infixr}\ ::\ n$ for some n we can write as well the following

$$a_1\ :: a_2\ \cdots a_n :: [\,]$$

- In ordinary mathematical notation, we would write $\langle a_1, \ldots, a_n \rangle$ for such an element.
- Jump over next slides.

Remarks on Tuples of Length n

- In **ordinary mathematics**, we would define

$$\begin{array}{rcl} \mathrm{Tuple}(A,0) & := & \{\langle\rangle\} \ , \\ \mathrm{Tuple}(A,n+1) & := & \{\langle a_1,\ldots,a_{n+1}\rangle \quad \mid a_1,\ldots,a_{n+1} \in A\} \ . \end{array}$$

- If we define

$$\begin{array}{rcl} [\,] & := & \langle\rangle \ , \\ a_1 :: \langle a_2,\ldots,a_{n+1}\rangle & := & \langle a_1,\ldots,a_{n+1}\rangle \ , \end{array}$$

  then this reads:

$$\begin{array}{rcl} \mathrm{Tuple}(A,0) & := & \{[\,]\} \ , \\ \mathrm{Tuple}(A,n+1) & := & \{a :: b \quad \mid a \in A \wedge b \in \mathrm{Tuple}(A,n)\} \ . \end{array}$$

Remarks on Tuples of Length n

- In the type theoretic definition we have **constructors**

$$
\begin{array}{lcl}
[\,] & : & \mathrm{Tuple}\ A\ \mathrm{Z} \\
\_::\_ & : & A \rightarrow \mathrm{Tuple}\ A\ n \rightarrow \mathrm{Tuple}\ A\ (\mathrm{S}\ n)
\end{array}
$$

- This is the **type theoretic analogue** of the previous definitions.

Componentwise Sum of n-Tuples

- We define **component-wise sum of tuples of length n**.
  - Using mathematical notation, this sum for instance as follows:

$$\langle 2,3,4\rangle + \langle 5,6,7\rangle = \langle 7,9,11\rangle \quad .$$

Componentwise Sum of n-Tuples

$$
\begin{array}{lllll}
\mathrm{sum}\mathbb{N}\mathrm{Tuple} : (n : \mathbb{N}) \to \mathrm{Tuple}\ \mathbb{N}\ n \to \mathrm{Tuple}\ \mathbb{N}\ n \to \mathrm{Tuple}\ \mathbb{N}\ n \\
\mathrm{sum}\mathbb{N}\mathrm{Tuple}\quad \mathrm{Z} & [\,] & [\,] & = & [\,] \\
\mathrm{sum}\mathbb{N}\mathrm{Tuple}\quad (\mathrm{S}\ n) & (m :: l) & (m' :: l') & = & \\
\qquad (m + m') :: (\mathrm{sum}\mathbb{N}\mathrm{Tuple}\ n\ l\ l')
\end{array}
$$

tuple.agda

6 (h) Lists

- We define the set of lists of elements of type `A` in Agda.
- We have two constructors:
  - [], generating the empty list.
  - _::_, adding an element of `A` in front of a list
- So we define lists as follows:

$$\text{infixr } 20\ \_{::}\_$$

$$\begin{array}{lll} \multicolumn{3}{l}{\text{data List } (A : \text{Set}) : \text{Set where}} \\ \quad [\,] & : & \text{List } A \\ \quad \_{::}\_ & : & A \to \text{List } A \to \text{List } A \end{array}$$

Elimination Principle for Lists

- The elimination principle is structural recursion on lists:
  Assume
  - $A : \mathrm{Set}$
  - $C : \mathrm{Set}$, depending on $l : \mathrm{List}\ A$.

  Then we can define

$$
\begin{array}{llcl}
f : (l : \mathrm{List}\ A) \to C & & & \\
f & [\,] & = & \{!\ \ !\} \\
f & (a :: l) & = & \{!\ \ !\}
\end{array}
$$

  and in the second goal we can make use of $f\ l$.

Example: Length of a List

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\mathrm{length} : \mathrm{List}\ \mathbb{N} \to \mathbb{N}} \\
\mathrm{length} & [\,] & = & \mathrm{Z} \\
\mathrm{length} & (\_ :: \mathit{l}) & = & \mathrm{S}\ (\mathrm{length}\ \mathit{l})
\end{array}
$$

Example: sumlist

- $\mathrm{sumlist}\ l$ will compute the sum of the elements of list l.

$$\begin{array}{llll} \multicolumn{4}{l}{\mathrm{sumlist} : \mathrm{List}\ \mathbb{N} \to \mathbb{N}} \\ \mathrm{sumlist} & [\,] & = & \mathrm{Z} \\ \mathrm{sumlist} & (n :: l) & = & n + \mathrm{sumlist}\ l \end{array}$$

Interesting Exercise

- Define

$$\_{+\!+}\_ : \{A : \mathrm{Set}\} \to \mathrm{List}\ A \to \mathrm{List}\ A \to \mathrm{List}\ A \ ,$$

  s.t. $l +\!+ l'$ is the result of appending the list l' at the end of list l.
- E.g., if $\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}$ are elements of A, then

$$\begin{array}{l} \mathrm{a} :: \mathrm{b} :: [\,] \quad +\!+ \quad \mathrm{c} :: \mathrm{d} :: [\,] \\ = \mathrm{a} :: \mathrm{b} :: \mathrm{c} :: \mathrm{d} :: [\,] \end{array}$$

list.agda

6 (i) Universes

- A universe U is a set, the elements of which are codes for sets.
- So we have
  - $\mathrm{U} : \mathrm{Set}$,
  - $\mathrm{T} : \mathrm{U} \to \mathrm{Set}$ (the decoding function).
- We consider in the following a universe closed under
  - $\bot$, $\top$, Bool, $\mathbb{N}$,
  - $+$,
  - Σ,
  - the dependent function type.

Rules for the Universe

Formation Rule

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

$$\frac{a : \mathrm{U}}{\mathrm{T}\ a : \mathrm{Set}}\ (\text{T-F})$$

Rules for the Universe

Introduction and Equality Rules

$$\widehat{\perp} : \mathrm{U} \qquad (\text{U-I}_{\widehat{\perp}}) \qquad \mathrm{T}\ (\widehat{\perp}) = \perp : \mathrm{Set} \qquad (\text{T-Eq}_{\widehat{\perp}})$$

$$\widehat{\top} : \mathrm{U} \qquad (\text{U-I}_{\widehat{\top}}) \qquad \mathrm{T}\ (\widehat{\top}) = \top : \mathrm{Set} \qquad (\text{T-Eq}_{\widehat{\top}})$$

$$\widehat{\mathrm{Bool}} : \mathrm{U} \qquad (\text{U-I}_{\widehat{\mathrm{Bool}}}) \qquad \mathrm{T}\ (\widehat{\mathrm{Bool}}) = \mathrm{Bool} : \mathrm{Set} \qquad (\text{T-Eq}_{\widehat{\mathrm{Bool}}})$$

$$\widehat{\mathbb{N}} : \mathrm{U} \qquad (\text{U-I}_{\widehat{\mathbb{N}}}) \qquad \mathrm{T}\ (\widehat{\mathbb{N}}) = \mathbb{N} : \mathrm{Set} \qquad (\text{T-Eq}_{\widehat{\mathbb{N}}})$$

Rules for the Universe

Introduction and Equality Rules (Cont.)

$$\frac{a : \mathrm{U} \qquad b : \mathrm{U}}{a \mathbin{\widehat{+}} b : \mathrm{U}} \ (\mathrm{U\text{-}I}_{\widehat{+}})$$

$$\mathrm{T}\ (a \mathbin{\widehat{+}} b) = \mathrm{T}\ a + \mathrm{T}\ b : \mathrm{Set} \quad (\mathrm{T\text{-}Eq}_{\widehat{+}})$$

$$\frac{a : \mathrm{U} \qquad b : \mathrm{T}\ a \to \mathrm{U}}{\widehat{\Sigma}\ a\ b : \mathrm{U}} \ (\mathrm{U\text{-}I}_{\widehat{\Sigma}})$$

$$\mathrm{T}\ (\widehat{\Sigma}\ a\ b) = \Sigma\ (\mathrm{T}\ a)\ (\lambda x^{\mathrm{T}\ a}.\mathrm{T}\ (b\ x)) : \mathrm{Set} \quad (\mathrm{T\text{-}Eq}_{\widehat{\Sigma}})$$

Rules for the Universe

Introduction and Equality Rules (Cont.)

$$\frac{a : \mathrm{U} \qquad b : \mathrm{T}\ a \to \mathrm{U}}{\widehat{\Pi}\ a\ b : \mathrm{U}}\ (\mathrm{U\text{-}I}_{\widehat{\Pi}})$$

$$\mathrm{T}\ (\widehat{\Pi}\ a\ b) = (x : \mathrm{T}\ a) \to \mathrm{T}\ (b\ x) : \mathrm{Set} \quad (\mathrm{T\text{-}Eq}_{\widehat{\Pi}})$$

Elimination and Equality Rules

- There exist as well elimination rules and corresponding equality rules for the universe.
- They are very long (one step for each of constructor of U) and are not very much used.
- They follow the principles present in previous rules.
- We have of course as well the equality versions of the formation-, introduction- and equality rules.

Applications of the Universe

- Ordinary elimination rules don't allow to eliminate into Set.
- However often, one can verify, that all sets needed are "elements of a universe",
  - i.e. there are codes in the universe representing them.
- Then one can eliminate into the universe instead of Set and use T to obtain the required function.

Applications of the Universe

- Example: Define

$$\begin{array}{rcl} \widehat{\mathrm{Atom}} & : & \mathrm{Bool} \to \mathrm{U} \ , \\ \widehat{\mathrm{Atom}} & := & \mathrm{Case}_{\mathrm{Bool}}\ (\lambda x^{\mathrm{Bool}}.\mathrm{U})\ \widehat{\top}\ \widehat{\bot} \ , \\ & & \\ \mathrm{Atom} & : & \mathrm{Bool} \to \mathrm{Set} \ , \\ \mathrm{Atom} & : & \lambda x^{\mathrm{Bool}}.\mathrm{T}\ (\widehat{\mathrm{Atom}}\ x) \ , \end{array}$$

  Then

  - $\mathrm{Atom}\ \mathrm{tt} = \top$,
  - $\mathrm{Atom}\ \mathrm{ff} = \bot$.

Universes in Agda

- U and T need to be defined simultaneously.
  - Usually Agda type checks definitions in sequence, so no reference to later definitions possible.
  - Special construct **mutual**.
    - Everything in the scope of it is type checked simultaneously.
    - Scope determined by indentation.
  - It is necessary, since the definition of U refers to that of T, and the definition of T refers to that of U.
  - In general mutual allows simultaneous inductive and/or recursive definitions.
  - The termination checker can handle certain terminating simultaneous inductive and/or recursive definitions like the universe.

Universes in Agda (Cont.)

$$
\begin{array}{l}
\mathrm{mutual} \\
\quad \mathrm{data}\ \mathrm{U} : \mathrm{Set}\ \mathrm{where} \\
\qquad \begin{array}{lcl}
\bot\mathrm{hat} & : & \mathrm{U} \\
\mathrm{tophat} & : & \mathrm{U} \\
\mathrm{Boolhat} & : & \mathrm{U} \\
\mathbb{N}\mathrm{hat} & : & \mathrm{U} \\
\_{+}\mathrm{hat}\_ & : & \mathrm{U} \to \mathrm{U} \to \mathrm{U} \\
\Sigma\mathrm{hat} & : & (a : \mathrm{U}) \to (\mathrm{T}\ a \to \mathrm{U}) \to \mathrm{U} \\
\Pi\mathrm{hat} & : & (a : \mathrm{U}) \to (\mathrm{T}\ a \to \mathrm{U}) \to \mathrm{U}
\end{array}
\end{array}
$$

Universes in Agda (Cont.)

T in the following is to be intended the same as U:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{T} : \mathrm{U} \to \mathrm{Set}} \\
\mathrm{T} \quad \bot\mathrm{hat} & = & \bot \\
\mathrm{T} \quad \mathrm{tophat} & = & \top \\
\mathrm{T} \quad \mathrm{Boolhat} & = & \mathrm{Bool} \\
\mathrm{T} \quad \mathbb{N}\mathrm{hat} & = & \mathbb{N} \\
\mathrm{T} \quad (a + \mathrm{hat}\ b) & = & \mathrm{T}\ a + \mathrm{T}\ b \\
\mathrm{T} \quad (\Sigma\mathrm{hat}\ a\ b) & = & \Sigma\ (\mathrm{T}\ a)\ (\lambda x \to \mathrm{T}\ (b\ x)) \\
\mathrm{T} \quad (\Pi\mathrm{hat}\ a\ b) & = & \Pi\ (\mathrm{T}\ a)\ (\lambda x \to \mathrm{T}\ (b\ x))
\end{array}
$$

6 (j) Algebraic Types

- The construct **data** in Agda is much more powerful than what is covered by type theoretic rules.
- In general we can define now sets having arbitrarily many constructors with arbitrarily many arguments of arbitrary types.

$$
\begin{array}{l}
\mathrm{data\ A : Set\ where} \\
\quad \mathrm{C}_1 \ : (a_1 : \mathrm{A}^1_1) \to (a_2 : \mathrm{A}^1_2) \to \cdots \ (a_{n_1} : \mathrm{A}^1_{\mathrm{n}_1}) \to A \\
\quad \mathrm{C}_2 \ : (a_1 : \mathrm{A}^2_1) \to (a_2 : \mathrm{A}^2_2) \to \cdots (a_{n_2} : \mathrm{A}^2_{\mathrm{n}_2}) \ \to A \\
\quad \ldots \\
\quad \mathrm{C}_m : (a_1 : \mathrm{A}^{\mathrm{m}}_1) \to (a_2 : \mathrm{A}^{\mathrm{m}}_2) \to \cdots (a_{n_m} : \mathrm{A}^{\mathrm{m}}_{\mathrm{n}_{\mathrm{m}}}) \to A
\end{array}
$$

Meaning of "data"

- The idea is that A as before is the least set A s.t. we have constructors:

  $$\begin{aligned} \mathrm{C_i} : (a_{i1} : \mathrm{A_{i1}}) \\ &\to \cdots \\ &\to (a_{in_i} : \mathrm{A_{in_i}}) \\ &\to \mathrm{A} \end{aligned}$$

  where a constructor always constructs new elements.
- In other words the elements of A are exactly those constructed by those constructors.

Strictly Positive Algebraic Types

- In the types A_{ij} we can make use of A.
  - However, it is difficult to understand A, if we have **negative** occurrences of A.
  - Example:

$$\begin{array}{l} \mathrm{data}\ \mathrm{A} : \mathrm{Set}\ \mathrm{where} \\ \qquad \mathrm{C} : (\mathrm{A} \to \mathrm{A}) \to \mathrm{A} \end{array}$$

  - What is the least set A having a constructor

$$\mathrm{C} : (\mathrm{A} \to \mathrm{A}) \to \mathrm{A} \qquad ?$$

Strictly Positive Algebraic Types

- If we
  - have constructed some elements of A already,
  - find a function $f : \mathrm{A} \to \mathrm{A}$, and
  - add $\mathrm{C\ f}$ to A,

  then f might no longer be a function $\mathrm{A} \to \mathrm{A}$.
  (f applied to the new element $\mathrm{C}\ f$ might not be defined).
- In fact, the termination checker issues a warning, if we define A as above.
- We shouldn't make use of such definitions.

Strictly Positive Algebraic Types

- A "good" definition is the set of lists of natural numbers, defined as follows:

$$\begin{array}{l} \mathrm{data}\ \mathbb{N}\mathrm{List} : \mathrm{Set}\ \ \mathrm{where} \\ \qquad \begin{array}{lcl} [\,] & : & \mathbb{N}\mathrm{List} \\ \_{::}\_ & : & \mathbb{N} \to \mathbb{N}\mathrm{List} \to \mathbb{N}\mathrm{List} \end{array} \end{array}$$

- The constructor $\_{::}\_$ of $\mathbb{N}\mathrm{List}$ refers to $\mathbb{N}\mathrm{List}$, but in a positive way: We have: if $a : \mathbb{N}$ and $l : \mathbb{N}\mathrm{List}$, then

$$(a :: l) : \mathbb{N}\mathrm{List} \enspace .$$

Strictly Positive Algebraic Types

- If we add $a :: l$ to $\mathbb{N}\mathrm{List}$, the reason for adding it (namely $l : \mathbb{N}\mathrm{List}$) is not destroyed by this addition.
- So we can "construct" the set $\mathbb{N}\mathrm{List}$ by
  - starting with the empty set,
  - adding [] and
  - closing it under $\_ :: \_$ whenever possible.

- Because we can "construct" $\mathbb{N}\mathrm{List}$, the above is an acceptable definition.

Strictly Positive Algebraic Types

- In general:

$$\begin{array}{l} \mathrm{data}\ \mathrm{A} : \mathrm{Set}\ \mathrm{where} \\ \quad \mathrm{C}_1 \ : (a_1 : \mathrm{A}^1_1) \to (a_2 : \mathrm{A}^1_2) \to \cdots (a_{n_1} : \mathrm{A}^1_{\mathrm{n}_1}) \to A \\ \quad \mathrm{C}_2 \ : (a_1 : \mathrm{A}^2_1) \to (a_2 : \mathrm{A}^2_2) \to \cdots (a_{n_2} : \mathrm{A}^2_{\mathrm{n}_2}) \to A \\ \quad \cdots \\ \quad \mathrm{C}_m : (a_1 : \mathrm{A}^{\mathrm{m}}_1) \to (a_2 : \mathrm{A}^{\mathrm{m}}_2) \to \cdots (a_{n_m} : \mathrm{A}^{\mathrm{m}}_{\mathrm{n}_\mathrm{m}}) \to A \end{array}$$

  is a strictly positive algebraic type, if all A_{ij} are
  - either types which don't make use of A
  - or are A itself.
- And if A is a strictly positive algebraic type, then A is acceptable.

Strictly Positive Algebraic Types

- The definitions of finite sets, $\Sigma\ A\ B$, $A + B$ and $\mathbb{N}$ were strictly positive algebraic types.

One further Example

- The set of binary trees can be defined as follows:

$$
\begin{array}{l}
\mathrm{data}\ \mathrm{BinTree} : \mathrm{Set}\ \ \mathrm{where} \\
\quad \begin{array}{lcl}
\mathrm{leaf} & : & \mathrm{BinTree} \\
\mathrm{branch} & : & \mathrm{Bintree} \to \mathrm{Bintree} \to \mathrm{Bintree}
\end{array}
\end{array}
$$

- This is a strictly positive algebraic type.
  bintree.agda

Extensions of Strict. Pos. Alg. Types

- An often used extension is to define several sets simultaneously inductively.
- Example: the even and odd numbers:

$$
\begin{array}{l}
\text{mutual} \\
\qquad \text{data Even} : \text{Set where} \\
\qquad\quad \begin{array}{lcl} \mathrm{Z} & : & \text{Even} \\ \mathrm{S} & : & \text{Odd} \to \text{Even} \end{array} \\
\qquad \text{data Odd} : \text{Set where} \\
\qquad\quad \begin{array}{lcl} \mathrm{S}' & : & \text{Even} \to \text{Odd} \end{array}
\end{array}
$$

- In such examples the constructors refer strictly positive to all sets which are to be defined simultaneously.
  evenodd.agda

Extensions of Strict. Pos. Alg. Types

- We can even allow $\mathrm{A}_{ij} = B_1 \to A$ or even $\mathrm{A}_{ij} = B_1 \to \cdots \to B_l \to A$, where A is one of the types introduced simultaneously.
- Example (called "Kleene's O"):

$$
\begin{array}{l}
\mathrm{data}\ \mathrm{O} : \mathrm{Set}\ \ \mathrm{where} \\
\qquad \begin{array}{lcl}
\mathrm{leaf} & : & \mathrm{O} \\
\mathrm{succ} & : & \mathrm{O} \to \mathrm{O} \\
\mathrm{lim} & : & (\mathbb{N} \to \mathrm{O}) \to \mathrm{O}
\end{array}
\end{array}
$$

- The last definition is unproblematic, since, if we have $f : \mathbb{N} \to \mathrm{O}$ and construct $\lim f$ out of it, adding this new element to O doesn't destroy the reason for adding it to O.
- So again O can be "constructed".

Elimination Rules for data

- Functions f from strictly positive algebraic types can now be defined by case distinction as before.
- For termination we need only that in the definition of f, when have to define $f\ (\mathrm{C}\ a_1\ \cdots\ a_n)$, we can refer only to f applied to elements used in $\mathrm{C}\ a_1\ \cdots\ a_n$.

Examples

- For instance
  - in the Bintree example, when defining

$$\mathrm{f} : \mathrm{Bintree} \to A$$

by case-distinction, then the definition of

$$\mathrm{f}\ (\mathrm{branch}\ l\ r)$$

can make use of $\mathrm{f}\ l$ and $\mathrm{f}\ r$.

Examples

- In the example of O, when defining

$$\mathrm{g} : \mathrm{O} \to A$$

by case-distinction, then the definition of

$$\mathrm{g}\ (\mathrm{lim}\ f)$$

can make use of $\mathrm{g}\ (f\ \ n)$ for all $n : \mathbb{N}$.