6. Data Types

(a) The set of Booleans.
(b) The finite sets.
(c) Atomic formulae and the traffic light example. (The example will be omitted 2008).
(d) The disjoint union of sets and disjunction.
(e) The $\Sigma$-set. (Will be omitted 2008.)
(f) Natural Deduction and Dependent Type Theory. (Will be largely omitted 2008).
(g) The set of natural numbers.
(h) Lists. (Will probably be omitted 2008.)
(i) Universes. (Will probably be omitted 2008.)
(j) Algebraic types. (Will be omitted 2008.)

(a) The Set of Booleans

Formation Rule

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

Introduction Rules

$\text{tt} : \text{Bool} \quad (\text{Bool-I}_{\text{tt}})$  
$\text{ff} : \text{Bool} \quad (\text{Bool-I}_{\text{ff}})$

Elimination Rule

$C : \text{Bool} \rightarrow \text{Set}$  
$\text{case}_{\text{tt}} : C \text{tt} \quad \text{case}_{\text{ff}} : C \text{ff}$

$\frac{\text{Case}_{\text{Bool}} C \text{case}_{\text{tt}} \text{case}_{\text{ff}} \text{tt} = \text{case}_{\text{tt}} : C \text{tt}}{\text{Case}_{\text{Bool}} C \text{case}_{\text{tt}} \text{case}_{\text{ff}} \text{ff} = \text{case}_{\text{ff}} : C \text{ff}}$  
$(\text{Bool-E}_{\text{tt}})$  
$(\text{Bool-E}_{\text{ff}})$

Equality Rules

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

Remarks

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

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

where the additional argument $C$ is required in order to determine the type of $\text{case}_{\text{tt}}$, of $\text{case}_{\text{ff}}$, and of the result of this construct.
Remarks (Cont.)

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

Remarks (Cont.)

- The argument $C$ above has no computational content.
  - It is not needed in order to compute $\text{Case}_{\text{Bool}} C \text{ case}_{\text{tt}} \text{ case}_{\text{ff}} \text{tt}$ and $\text{Case}_{\text{Bool}} C \text{ case}_{\text{tt}} \text{ case}_{\text{ff}} \text{ff}$.
  - $C$ 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.)

- If we define
  
  $C := \lambda b. \text{Bool}. D$
  
  $f := \lambda b. \text{Case}_{\text{Bool}} C \text{ case}_{\text{tt}} \text{ case}_{\text{ff}} b$
  
  $: (b : \text{Bool}) \rightarrow C \ b$

  where
  
  $(b : \text{Bool}) \rightarrow C \ b = (b : \text{Bool}) \rightarrow D$

  we have:

  - $f \text{tt} : C \text{tt}$.
  - $f \text{ff} : C \text{ff}$.
  - $f : (b : \text{Bool}) \rightarrow C \ b$.

Remarks (Cont.)

- We can write the elimination rule in a more compact but less readable way:
  
  $\text{Case}_{\text{Bool}} : (C : \text{Bool} \rightarrow \text{Set}) \rightarrow (\text{case}_{\text{tt}} : C \text{tt}) \rightarrow (\text{case}_{\text{ff}} : C \text{ff}) \rightarrow (b : \text{Bool}) \rightarrow C \ b$

- $\text{tt}$, $\text{ff}$ are the constructors of $\text{Bool}$.
Notice that we then get for $C : \text{Bool} \to \text{Set}$,

\[
    \begin{align*}
        \text{case}_{\text{tt}} & : C \text{tt}, \\
        \text{case}_{\text{ff}} & : C \text{ff}
    \end{align*}
\]

\[
    f := \text{Case}_{\text{Bool}} C \text{case}_{\text{tt}} \text{case}_{\text{ff}} ,
\]

\[
    : (b : \text{Bool}) \to C b
\]

\[
    f \text{tt} = \text{Case}_{\text{Bool}} C \text{case}_{\text{tt}} \text{case}_{\text{ff}} \text{tt} = \text{case}_{\text{tt}} : C \text{tt},
\]

\[
    f \text{ff} = \text{Case}_{\text{Bool}} C \text{case}_{\text{tt}} \text{case}_{\text{ff}} \text{ff} = \text{case}_{\text{ff}} : C \text{ff}.
\]

So we obtain functions from $\text{Bool}$ into other sets without having to write $\lambda b : \text{Bool} . \cdots$.

That’s why we choose the argument to eliminate from as the last one.

This is similar to the definition of for instance $(\cdot \cdot \cdot)$ in Haskell

\[
    (\cdot \cdot \cdot) : \text{int} \to \text{int} \to \text{int}.
\]

$(\cdot \cdot \cdot) 3$ is the function which takes an integer and adds it $3$.

Shorter than writing $\lambda x : \text{int} . 3 + x$.

Note that we have the following order of the arguments of $\text{Case}_{\text{Bool}}$:

- 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 $\text{Case}_{\text{Bool}}$ is a “then _else _if ” – the condition (if . . .) is the last one.

Assume we have introduced in type theory

\[
    \begin{align*}
        \text{Name} & : \text{Bool} \to \text{Set} , \\
        \text{Name tt} & = \text{FemaleName} , \\
        \text{Name ff} & = \text{MaleName} .
    \end{align*}
\]
Select Example

Then we can define the function

\begin{align*}
\text{SelectBool} & : (b : \text{Bool}) \rightarrow \text{Name} \\
\text{SelectBool } \ tt & = \text{sara} \\
\text{SelectBool } \ ff & = \text{tom}
\end{align*}

as follows:

\[ \text{SelectBool} = \text{Case}_{\text{Bool}} \text{Name } \text{sara} \text{ tom} \]

Note that by using twice the \( \eta \)-rule we get that

\[ \text{SelectBool} = \lambda b : \text{Bool}. \text{Case}_{\text{Bool}} \ (\lambda d : \text{Bool}. \text{Name} \ d) \text{ sara} \text{ tom } b \]

Select Example

We verify the correctness of \( \text{SelectBool} \):

\begin{align*}
\text{SelectBool } \ tt & = \text{Case}_{\text{Bool}} \text{Name } \text{sara} \text{ tom } tt = \text{sara} \\
\text{SelectBool } \ ff & = \text{Case}_{\text{Bool}} \text{Name } \text{sara} \text{ tom } ff = \text{tom}.
\end{align*}

Jump over \( \land_{\text{Bool}} \)

Example: \( \land_{\text{Bool}} \)

We want to introduce conjunction

\[ \land_{\text{Bool}} : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \]

This will be of the form

\[ \land_{\text{Bool}} = \lambda (b, c : \text{Bool}). t \]

for some term \( t \).

\( t \) will be defined by case distinction on \( b \), so we get

\[ \land_{\text{Bool}} = \lambda (b, c : \text{Bool}). \text{Case}_{\text{Bool}} \ C \ e \ f \ b \]

for some \( e, f \).

Example: \( \land_{\text{Bool}} \)

\[ \land_{\text{Bool}} = \lambda (b, c : \text{Bool}). \text{Case}_{\text{Bool}} \ C \ e \ f \ b \]

\( C \) will be the set into which we are eliminating, depending on a Boolean value.

- It need to be an element of \( \text{Bool} \rightarrow \text{Set} \).
- Therefore we have \( C = \lambda d : \text{Bool}. D \) for some \( D \) which might depend on \( d \).
- The set, into which we are eliminating, is always the same, namely \( \text{Bool} \).
- So \( D = \text{Bool} \) and therefore we have

\[ C = \lambda d : \text{Bool}. \text{Bool} \]
**Example:** \( \land_{\text{Bool}} \)

- Note that in \( \lambda d_{\text{Bool}}.\text{Bool} \)
  
  \text{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 \( \text{Set} \).
  - So here \( \text{Bool} \) is a term.

---

**Two Meanings of Elements of Set**

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

---

**Example:** \( \land_{\text{Bool}} \)

- So
  
  \[ \land_{\text{Bool}} = \lambda(b, c : \text{Bool}).\text{Case}_{\text{Bool}}(\lambda d_{\text{Bool}}.\text{Bool}) e f b \]
  
  for some \( e, f \).
  
  For conjunction we have:
  - If \( b \) is true then
    
    \[ b \land c = \text{tt} \land c = c \]
    
    So the if-case \( e \) above is \( c \).
  - If \( c \) is false then
    
    \[ b \land c = \text{ff} \land c = \text{ff} \]
    
    So the else-case \( f \) above is \( \text{ff} \).

---

**Example:** \( \land_{\text{Bool}} \)

- In total we define therefore
  
  \[ \land_{\text{Bool}} = \lambda(b, c : \text{Bool}).\text{Case}_{\text{Bool}}(\lambda d_{\text{Bool}}.\text{Bool}) c \text{ ff } b \]
  
  : \( \text{Bool} \to \text{Bool} \to \text{Bool} \)

- We verify the correctness of this definition:
  - \( \land_{\text{Bool}} \text{ tt } c = \text{Case}_{\text{Bool}}(\lambda d_{\text{Bool}}.\text{Bool}) \text{ c ff } \text{tt} = c. \) as desired.
  - \( \land_{\text{Bool}} \text{ ff } c = \text{Case}_{\text{Bool}}(\lambda d_{\text{Bool}}.\text{Bool}) \text{ c ff } \text{ff} = \text{ff}. \) Correct as desired.

Jump over derivation of \( \land_{\text{Bool}} \)
Derivation of $\land_{\text{Bool}}$

We derive in the following $\land_{\text{Bool}} : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$.

We write $\text{Bool}$, if it is a type in boldface red, and if it is a term, in italic blue.

Derivation of $\land_{\text{Bool}}$

First we derive

$$b : \text{Bool}, c : \text{Bool} \Rightarrow (\lambda d^{\text{Bool}}. \text{Bool}) \text{ tt} : \text{Set}$$

(derivations omitted)

Similarly follows

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

(derivations omitted)
Derivation of $\land_{\text{Bool}}$

- Using part of the proof above, we derive
  \[
  b : \text{Bool}, c : \text{Bool} \Rightarrow c : (\lambda d. \text{Bool}) \text{ tt}
  \]

- We derive using (Transfer0)
  \[
  b : \text{Bool}, c : \text{Bool} \Rightarrow \text{ff} : (\lambda d. \text{Bool}) \text{ ff}
  \]

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

We can extend add elimination and equality rules, having as result $\text{Type Bool}$:

**Elimination Rule into Type**

\[
\begin{align*}
\text{Case}_\text{Boolean} \ C \ \text{case}_\text{tt} \ \text{case}_\text{ff} \ b : C \ b
\end{align*}
\]

**Equality Rules into Type**

\[
\begin{align*}
\text{Case}_\text{Boolean} \ C \ \text{case}_\text{tt} \ \text{case}_\text{tt} \ \text{tt} = \text{case}_\text{tt} : C \ \text{tt}
\end{align*}
\]

\[
\begin{align*}
\text{Case}_\text{Boolean} \ C \ \text{case}_\text{tt} \ \text{case}_\text{ff} \ \text{ff} = \text{case}_\text{ff} : C \ \text{ff}
\end{align*}
\]
Example Select

Assume we have introduced

FemaleName : Set
= \{jill, sara\}

MaleName : Set
= \{tom, jim\}

Then we can define

Name : Bool → Set
:= \lambda x^{Bool}. \text{Case}_{Bool}^{Type} (\lambda y^{Set}. \text{FemaleName MaleName} x)

: Bool → Set

(b) The Finite Sets

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

**Formation Rule**

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

**Introduction Rules**

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

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

Rules for Fin\(_n\)

**Elimination Rule**

\[
C : \text{Fin}_n → \text{Set} \\
\begin{align*}
  s_0 &: C A^n_0 \\
  s_1 &: C A^n_1 \\
  \vdots & \\
  s_{n-1} &: C A^n_{n-1} \\
  a &: \text{Fin}_n
\end{align*}
\]

\[ \text{Case}_n C s_0 \ldots s_{n-1} a : C\ a \quad (\text{Fin}_n\text{-El}) \]
The Finite Sets (Cont)

Equality Rules

\[ C : \text{Fin}_n \rightarrow \text{Set} \]
\[ s_0 : C \text{ A}_0^n \]
\[ s_1 : C \text{ A}_1^n \]
\[ \vdots \]
\[ s_{n-1} : C \text{ A}_{n-1}^n \]

\( \text{Case}_n C \ s_0 \ldots s_{n-1} \text{ A}_k^n = s_k : C \text{ A}_k^n \) \hspace{1cm} (\text{Fin}_n\text{-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:

\[ \text{Case}_n C \ s_0 \ldots s_{n-1} \text{ A}_k^n = s_k : C \text{ A}_k^n \]

- We sometimes even omit the type:

\[ \text{Case}_n C \ s_0 \ldots s_{n-1} \text{ A}_k^n = s_k \]

More Compact Elimination Rules

\[ \text{Case}_n : (C : \text{Fin}_n \rightarrow \text{Set}) \rightarrow (s_0 : C \text{ A}_0^n) \rightarrow \ldots \rightarrow (s_{n-1} : C \text{ A}_{n-1}^n) \rightarrow (a : \text{Fin}_n) \rightarrow C \ a \]

Elimination into Type

- Similarly as for \( \text{Bool} \) we can write down elimination rules, where \( C : \text{Fin}_n \rightarrow \text{Type} \) (instead of \( C : \text{Fin}_n \rightarrow \text{Set} \)).

- This can be done for all sets defined later as well.
Rules for $\top$

$\top$ is the special case $\text{Fin}_n$ for $n = 1$ (we write $\text{true}$ for $\text{A}_1$):

**Formation Rule**

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

**Introduction Rules**

\[
\text{true} : \top \quad (\top-\text{I})
\]

**Elimination Rule**

\[
\begin{array}{c}
C : \top \rightarrow \text{Set} \\
c : C \text{ true} \\
t : \top
\end{array}
\quad
\frac{\text{Case } \top \ c \ t}{C \ t : C \ t} \quad (\top-\text{El})
\]

### Equality Rule

\[
\text{Case } \top \ c \text{ true} = c
\]

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

Jump over next slide (advanced material)

---

Rules for $\bot$

$\bot$ is the special case $\text{Fin}_n$ for $n = 0$:

**Formation Rule**

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

There is no **Introduction Rule**

**Elimination Rule**

\[
\begin{array}{c}
C : \bot \rightarrow \text{Set} \\
f : \bot
\end{array}
\quad
\frac{\text{Case } \bot \ f}{C \ f : C \ f} \quad (\bot-\text{El})
\]

There is no **Equality Rule**

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

---

Rules for $\top$ (Cont.)

- Case $\top$ is **computationally not very interesting**.
- Case $\top$ $c$ is the constant function $\lambda x.\top. c$.
- However, in Agda we might not be able to derive $\lambda t.\top. c : (t : \top) \rightarrow C \ t$

From a **logic point of view**, it expresses:

From an element of $C \text{ true}$ we obtain an element of $C \ t$ for every $t : \top$.

So there is no $C : \bot \rightarrow \text{Set}$ s.t. $C \text{ true}$ 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 $\text{true}$.

This equality is called **Leibnitz equality**.
(c) Atomic Formulae

Full title of this section: Atomic formulae and the Traffic Light Example.

Atom can be defined as follows:

\[
\text{Atom} : \text{Bool} \rightarrow \text{Set} \\
\text{Atom} = \text{Case}_{\text{Bool}} (\lambda b. \text{Set}) \top \bot
\]

So we have

\[
\text{Atom \text{tt}} = \top \\
\text{Atom \text{ff}} = \bot
\]

Jump over Traffic Light Example.

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:

\[
\text{data Colour : Set where} \\
\text{red : Colour} \\
\text{green : Colour}
\]

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')

\[
\text{record PhysState : Set where} \\
\text{field} \\
\text{sigA : Colour} \\
\text{sigB : Colour}
\]
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.
  - onlyAGreen – signal A (and A') is green, signal B is red.
  - onlyBGreen – signal B is green, signal A is red.

We therefore define

```haskell
data ControlState : Set where
  allRed : ControlState
  onlyAGreen : ControlState
  onlyBGreen : ControlState
```

Control States to Physical States

- We define the state of signals A, B depending on a control state:
  - toSigA : ControlState → Colour
  - toSigA allRed = red
  - toSigA onlyAGreen = green
  - toSigA onlyBGreen = red
  - toSigB : ControlState → Colour
  - toSigB allRed = red
  - toSigB onlyAGreen = red
  - toSigB onlyBGreen = green

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

```haskell
toPhysState : ControlState → PhysState
toPhysState c = record{sigA = toSigA c ;
  sigB = toSigB c }
```
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:

  \[
  \text{CorAux} : \text{Colour} \to \text{Colour} \to \text{Set} \\
  \text{CorAux} \text{ red } \_ = \top \\
  \text{CorAux} \text{ green red } = \top \\
  \text{CorAux} \text{ green green } = \bot
  \]

Safety of the System

- Now we show that all control states are safe:

  \[
  \text{corProof} : (s : \text{ControlState}) \to \text{Cor} (\text{toPhysState} s) \\
  \text{corProof allRed} = \text{true} \\
  \text{corProof onlyAGreen} = \text{true} \\
  \text{corProof onlyBGreen} = \text{true}
  \]

See `exampleTrafficLight1.agda`

Safety Predicate (Cont.)

- Now we define

  \[
  \text{Cor} : \text{PhysState} \to \text{Set} \\
  \text{Cor} s = \text{CorAux} (\text{PhysState}.\text{sigA} s) (\text{PhysState}.\text{sigB} s)
  \]

- **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 (Cont.)

- The first element true was an element of \text{Cor} (\text{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 toSigB onlyAGreen = green, then in the last step we obtain one goal of type \( \bot \).
  - 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.)

**Visualisation \((A + B)\)**

- Informally, if
  - \( A = \{1, 2\} \) and
  - \( B = \{1, 2, 3\} \), then
  - \( A + B = \{\text{inl}(1), \text{inl}(2), \text{inr}(1), \text{inr}(2), \text{inr}(3)\} \)
  - Each element of \( A + B \) is
    - either of the form \( \text{inl}(a) \) for some \( a : A \)
    - or of the form \( \text{inr}(b) \) for \( b : B \).

**Disjoint Union**

- 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
    - \( \text{inl} : A \rightarrow A + B \) and \( \text{inr} \).
    - Elements from \( a : A \) are inserted into \( A + B \) as \( \text{inl} a : A + B \).
    - Elements from \( b : B \) are inserted into \( A + B \) as \( \text{inr} b : A + B \).
    - \( \text{inl} \) stands for “in-left”, \( \text{inr} \) for “in-right”.
    - If we have \( a : A \) and \( a : B \), then \( a \) is represented both as \( \text{inl} a \) and \( \text{inr} a \) in \( A + B \).
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

\[ A \times B = \{p(1, 1), p(1, 2), p(1, 3), p(2, 1), p(2, 2), p(2, 3)\} \]

Each element of \( A \times B \) is of the form \( p(a, b) \) 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

\[ A + B = \{\text{inr}(b) \mid b : B\} \]

which has a copy of each element of \( B \).

\( A \times B \) is empty, since we cannot form a pair \( p(a, b) \) where \( a : A \), \( b : B \), since there is no element \( a : A \).

Rules for \( A + B \)

Formation Rule

\[
\begin{align*}
A : \text{Set} & \\
B : \text{Set} & \\
A + B : \text{Set} & \\
\hline
\text{(-F)}
\end{align*}
\]

Introduction Rules

\[
\begin{align*}
A : \text{Set} & \\
B : \text{Set} & \\
an : A & \\
\text{inl} \ A B a : A + B & \\
A : \text{Set} & \\
B : \text{Set} & \\
bn : B & \\
\text{inr} \ A B b : A + B & \\
\hline
\text{(+I_{\text{inl}})} & \\
\text{(+I_{\text{inr}})}
\end{align*}
\]

Elimination Rules

\[
\begin{align*}
A : \text{Set} & \\
B : \text{Set} & \\
C : (A + B) \rightarrow \text{Set} & \\
\text{case}_{\text{inl}} : (a : A) \rightarrow C \ (\text{inl} \ A B a) & \\
\text{case}_{\text{inr}} : (b : B) \rightarrow C \ (\text{inr} \ A B b) & \\
d : A + B & \\
\text{Case}_+ \ A B C \text{ case}_{\text{inl}} \text{ case}_{\text{inr}} d : C & \\
\hline
\text{(+El)}
\end{align*}
\]

\((\text{case}_{\text{inl}}, \text{case}_{\text{inr}} \ \text{stand for} \ \text{"case left", "case right"}).\)
Rules for $A + B$

**Equality Rules**

\[
\begin{align*}
\text{Case}_+ \ A \ B \ C \ \text{case}_{\text{inl}} \ \text{case}_{\text{inr}} \ (\text{inl} \ A \ B \ a) \\
= \text{case}_{\text{inl}} \ a : C \ (\text{inl} \ A \ B \ a) 
\end{align*}
\]

\[
\begin{align*}
\text{Case}_+ \ A \ B \ C \ \text{case}_{\text{inl}} \ \text{case}_{\text{inr}} \ (\text{inr} \ A \ B \ b) \\
= \text{case}_{\text{inr}} \ b : C \ (\text{inr} \ A \ B \ b)
\end{align*}
\]

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:
  - $\_ + \_ : \text{Set} \to \text{Set} \to \text{Set}$, written infix.
  - $\text{inl} : (A, B : \text{Set}) \to A \to (A + B)$.
  - $\text{inr} : (A, B : \text{Set}) \to B \to (A + B)$.
  - $\text{Case}_+ : (A, B : \text{Set})$
    \[\to (C : (A + B) \to \text{Set})\]
    \[\to ((a : A) \to C \ (\text{inl} \ A \ B \ a))\]
    \[\to ((b : B) \to C \ (\text{inr} \ A \ B \ b))\]
    \[\to (d : A + B)\]
    \[\to C \ d .\]

- Equality rule as before.

### Disjoint Union in Agda

- The disjoint union can be defined as a “data”-set having **two constructors**
  - $\text{inl}$ (in-left for left injection) and
  - $\text{inr}$ (in-right for right injection):

\[
\begin{align*}
\text{data} \ _+\_ \ (A \ B : \text{Set}) : \text{Set} \text{ where} \\
\text{inl} : A \to A + B \\
\text{inr} : B \to A + B
\end{align*}
\]

### Disjoint Union in Agda (Cont.)

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

\[
\begin{align*}
f : A + B \to \text{Bool} \\
f x = \{! !\}
\end{align*}
\]

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

\[
\begin{align*}
f : A + B \to \text{Bool} \\
f \ (\text{inl} \ a) = \text{tt} \\
f \ (\text{inr} \ b) = \text{ff}
\end{align*}
\]
Use of Concrete Disjoint Sets

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

```agda
data Plant : Set where
tree : Tree → Plant
flower : Flower → Plant
```

Now one can define for instance

```agda
isFlower : Plant → Bool
isFlower (tree t) = ff
isFlower (flower f) = tt
```

Disjunction

\( A \lor B \) is true iff \( A \) is true or \( B \) is true.

Therefore a proof of \( A \lor B \) consists of a proof of \( A \) or a proof of \( B \), plus the information which one.

It is therefore an element \( \text{inl} \ p \) for a proof \( p : A \) or an element \( \text{inr} \ q \) for a proof \( q : B \).

Therefore the set of proofs of \( A \lor B \) is the disjoint union of \( A \) and \( B \), i.e. \( A + B \).

We can identify \( A \lor B \) with \( A + B \).

Disjunction in Agda

Or is represented as disjoint union in type theory.

In Agda we can type the symbol for \( \lor \) using Leim as \texttt{\backslash vee}.

```agda
data _\lor_ (A B : Set) : Set where
or1 : A → A \lor B
or2 : B → A \lor B
```

See \texttt{exampleproofpropllogic7.agda}.

On the blackboard \( A \rightarrow A \lor B \) and \( A \lor A \rightarrow A \) will now be shown in Agda.

Example (Disjunction)

The following derives \( (A \lor B) \rightarrow (B \lor A) \):

```agda
lemma3 : A \lor B → B \lor A
lemma3 (or1 a) = or2 a
lemma3 (or2 b) = or1 b
```

See \texttt{exampleproofpropllogic9.agda}.
Disjunction with more Args.

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

\[
data \text{OR3 } (A \ B \ C: \text{Set}) : \text{Set} \text{ where} \\
\text{or1} : A \rightarrow \text{OR3 } A \ B \ C \\
\text{or2} : B \rightarrow \text{OR3 } A \ B \ C \\
\text{or2} : C \rightarrow \text{OR3 } A \ B \ C
\]

See exampleproofprologic8.agda.

Jump over $\Sigma$-Type.

Rules for $\Sigma$

Formation Rule

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

Introduction Rule

\[
\frac{A : \text{Set} \quad B : A \rightarrow \text{Set} \quad a : A \quad b : B \ a}{p \ A \ B \ a \ b : \Sigma \ A \ B} \quad (\Sigma\text{-I})
\]

(e) The $\Sigma$-Set

The $\Sigma$-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 $B : A \rightarrow \text{Set}$.

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 $\Sigma$-set is introduced by the user using a constructor, similar to the previous sets.
- The $\Sigma$-set behaves sometimes better than the standard product.

Equality Rule

\[
\text{Case}_\Sigma \ A \ B \ C \ c \ d : C \ d \quad (\Sigma\text{-Eq})
\]

Additionally we have the Equality versions of the formation-, introduction- and elimination-rules.
The $\Sigma$-Set using the Log. Framew.

- The more compact notation is:
  - $\Sigma : (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{Set}$.
  - $p : (A : \text{Set}) \rightarrow (B : A \rightarrow \text{Set}) \rightarrow (a : A) \rightarrow (B a) \rightarrow \Sigma A B$.

Equality rule as before.

The $\Sigma$-Set and the Dep. Prod.

- Both the $\Sigma$-set and the dep. product have similar introduction rules.
  - For the $\Sigma$-set, the constructors have additional arguments $A, B$ necessary for bureaucratic reasons only.
  - One can define the projections $\pi_0, \pi_1$ using $\text{Case}_\Sigma$:
    \[
    \pi_0 = \text{Case}_\Sigma A B (\lambda x (\Sigma A B). A) (\lambda x^A . \lambda y^B (B x). x) \\
    \pi_1 = \text{Case}_\Sigma A B (\lambda x (\Sigma A B). B \pi_0(x)) (\lambda x^A . \lambda y^B (B x). y)
    \]
  - On the other hand, from $\pi_0, \pi_1$ we can define $\text{Case}_\Sigma$ as follows:
    \[
    \lambda A^{\text{Set}} . \lambda B^{A \rightarrow \text{Set}} . \lambda C^{(\Sigma A B) \rightarrow \text{Set}} . \\
    \lambda s^{(a : A) \rightarrow (b : B a) \rightarrow C (p a b)} . \lambda d^{(\Sigma A B) . s} . \pi_0(d) . \pi_1(d).
    \]

However the dependent product has the $\eta$-rule (which is however not implemented in Agda).

Because of the lack of $\eta$-rule, $\Sigma$ works usually better than the dependent product in Agda.

I personally don’t use the dependent product of Agda much.
The \( \Sigma \)-Set in Agda

- \( \Sigma \) can be defined as a “data”-set with a constructor, e.g. \( p \):

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

- Elimination uses case-distinction:

\[
f : \Sigma \ A \ B \rightarrow D
f \ (p \ a \ b) = \{! !\}
\]

\[\text{sigmaset.agda}\]

The \( \Sigma \)-Set in Agda (Cont.)

- Again one usually defines concrete \( \Sigma \)-sets more directly.

  **Example:** Assume we have defined
  
  - a set PlantGroup for groups of plants (e.g. “tree”, “flower”),
  
  - depending on \( g : \text{PlantGroup} \), sets (PlantsInGroup \( g \))
    for plants in that group.

- The set of plants can then be defined as

\[
\text{data } \text{Plant} : \text{Set} \text{ where }
\]
\[
\text{plant} : (g : \text{PlantGroup}) \rightarrow \text{PlantsInGroup} \ g \rightarrow \text{Plant}
\]
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 \( A \land B \) from a proof of \( A \) and a proof of \( B \):

\[
\frac{p : A \quad q : B}{(p, q) : A \land B} \quad (\times - I)
\]

This means that we can **derive** \( A \land B \) from \( A \) and \( B \).

---

Example 1

For instance, assume we want to prove that a function \( \text{sort} \) from lists to lists is a sorting algorithm.

Then we have to show that for every list \( l \) the application of \( \text{sort} \) 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 \( \text{sort} l \) is sorted,
- then, that \( \text{sort} l \) 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 \( \land \).

---

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 \( A \land B \) is

\[
\frac{A \quad B}{A \land B} \quad (\land - I)
\]

The type theoretic introduction rule corresponds exactly to this rule.

---

Conjunction (Cont.)

The **elimination rule** for \( \land \) allows to project a proof of \( A \land B \) to a proof of \( A \) and a proof of \( B \):

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

This means that we can **derive** from \( A \land B \) both \( A \) and \( B \).

This corresponds to the **natural deduction elimination rule** for \( \land \):

\[
\frac{A \land B}{A} \quad (\land - \text{El}_0) \quad \frac{A \land B}{B} \quad (\land - \text{El}_1)
\]

---

Omit Example 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 \( \texttt{tt} \) or \( \texttt{ff} \) indicating whether \( n \) is in this list or not.

Assume now a sorting function \( \text{sort} \) from lists of natural numbers to natural numbers, plus a proof that it is a sorting function, i.e. that \( \text{sort} \ l \) is sorted and has the same elements as \( l \) for every list \( l \).

We want to apply \( f \) to \( \text{sort} \ l \) and need therefore a proof that \( \text{sort} \ l \) is sorted.

We have that the conjunction of “\( \text{sort} \ l \) is sorted” and “\( \text{sort} \ l \) has the same elements as \( l \)” holds.

Using the elimination rule for \( \land \) one can conclude the desired property, that \( \text{sort} \ l \) is sorted.

Example 3

Assume a proof of \( A \land B \).

We want to show \( B \land A \).

By \( \land \)-elimination we obtain from \( A \land B \) that \( B \) holds.

Similarly we conclude that \( A \) holds.

Using \( \land \)-introduction we conclude \( B \land A \).

In natural deduction, this proof is as follows:

\[
\begin{align*}
\frac{A \land B}{\neg \neg B} & (\land \lnot \lnot) \\
\frac{A \land B}{\land \neg} & (\land \neg) \\
\frac{A \land B}{B \land A} & (\land \lnot) \\
\end{align*}
\]

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 \( \lor \) allows to form a proof of \( A \lor B \) from a proof of \( A \) or from a proof of \( B \).

\[
\begin{align*}
\frac{A : \text{Set}}{\text{inl} \ A \ B \ p : A + B} & (+\text{-inl}) \\
\frac{B : \text{Set}}{\text{inr} \ A \ B \ p : A + B} & (+\text{-inr}) \\
\end{align*}
\]

Disjunction (Cont.)

Omitting the premises \( A, B : \text{Set} \) and omitting them as arguments of \( \text{inl} \) and \( \text{inr} \) (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:

\[
\begin{align*}
\frac{A : \text{Set}}{\text{inl} \ p : A + B} & (+\text{-inl}) \\
\frac{B : \text{Set}}{\text{inr} \ p : A + B} & (+\text{-inr}) \\
\end{align*}
\]
Disjunction (Cont.)

This means that we can derive \( A \lor B \) from \( A \) and from \( B \).

This is what is expressed by the natural deduction introduction rules for \( \lor \):

\[
\frac{A}{A \lor B} \quad (\lor\text{-}\text{I}_{\text{inl}}) \\
\frac{B}{A \lor B} \quad (\lor\text{-}\text{I}_{\text{inr}})
\]

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 \( \lor \) 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 \( \lor \) 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 \( B \):

\[
\begin{align*}
A &: \text{Set} \\
B &: \text{Set} \\
C &: (A \lor B) \to \text{Set} \\
sl &: (a : A) \to C \text{ (inl } A B a) \\
sr &: (b : B) \to C \text{ (inr } A B b) \\
d &: A \lor B \\
\text{Case}_+ A B C sl sr d &: C \quad (+\text{-El})
\end{align*}
\]

Omitting the dependency of \( C \) on \( A \lor B \), the premises \( A, B \) and \( C \), and the arguments \( A, B \) and \( C \), we get:

\[
\begin{align*}
d &: A \lor B \\
sl &: A \to C \\
sr &: B \to C \\
\text{Case}_+ sl sr d &: C \quad (+\text{-El})
\end{align*}
\]

This means that we can derive from \( A \lor B \) 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 $\lor$:

$$
\frac{A \lor B}{A \vdash C} \quad \frac{A \vdash C}{B \vdash C} \quad (\lor\text{-El})
$$

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

$$
\frac{}{A} \quad \frac{}{\vdash} \quad \frac{}{C}
$$

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 $\geq 5$.

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 $\geq 5$.
- 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 $\geq 5$.
- Therefore it is equal to 2, equal to 3, or $\geq 5$.

Now from the elimination rule for $\lor$ we conclude that the prime number chosen is equal to 2, equal to 3, or $\geq 5$.

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 $A_1, \ldots, A_n$):

\[
\frac{A \lor B}{A \lor B} \quad \frac{A \vdash B}{A \lor B} \quad \frac{B \vdash B}{A \lor B} \quad \frac{(\lor-I_{\text{inr}})}{B \lor A}
\]

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

**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 $A$:

\[
p : A \Rightarrow q : B \quad (\rightarrow I)
\]

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**

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 $\rightarrow$ for logical implication.
Example

We extend the proof that, if we have \( A \lor B \), then we have \( B \lor A \), to a proof of

\[(A \lor B) \supset (B \lor A)\]

The previous proof can be easily transformed into a proof of \( A \lor B \vdash B \lor A \).

By \( \supset \)-introduction, it follows \((A \lor B) \supset (B \lor A)\).

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 \( B \):

\[
\frac{p : A \supset B \quad q : A}{p \triangleright q : B} \quad (\supset \text{-El})
\]

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 \( \supset \):

\[
\frac{A \supset B \quad A}{B} \quad (\supset \text{-El})
\]

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:

\[
\frac{A, A \supset B \vdash A \supset B}{A, A \supset B \vdash (\supset \text{-El})} \\
\frac{A \vdash (A \supset B) \supset B}{\vdash (\supset \text{-I})}
\]

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 \(x : A\):

\[
\frac{x : A \Rightarrow p : B}{\lambda x^A.p : \forall x^A.B} (\rightarrow \text{-I})
\]

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 \( B[a := a] \):

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

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

\[
\frac{\forall x : A. B \quad a : A}{B[a := a]} (\forall\text{-El})
\]
**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 $\forall n, m : \mathbb{N}.0 + (n + m) == (n + m)$.

**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 $p : B[x := a]$:

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

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

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

**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 $S n : \mathbb{N}$ and $S n > n$.
  - 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 is written as follows:

$$\frac{\forall n^\mathbb{N}.0 + n == n \quad N \in \mathbb{N} \quad n : \mathbb{N} \quad m : \mathbb{N} \quad n + m : \mathbb{N}}{\forall n, m : \mathbb{N}.0 + (n + m) == (n + m)} (N=\text{-El})$$

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

$$\frac{n : \mathbb{N} \quad m : \mathbb{N} \quad 0 + (n + m) == (n + m)}{\forall n, m : \mathbb{N}.0 + (n + m) == (n + m)} (\forall=\text{-I})$$
Example

In natural deduction, this proof reads as follows:

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

\[
\frac{n : \mathbb{N} \vdash \exists m^\mathbb{N} \cdot m > n \quad \text{(}\exists\text{-I})}{\forall n^\mathbb{N} . \exists m^\mathbb{N} \cdot m > n}
\]

Existential Quantification (Cont.)

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

\[
\exists x^A . B \quad x^A , B \vdash C \quad \text{(}\exists\text{-El})
\]

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 \( \forall n^\mathbb{N} . \exists m^\mathbb{N} \cdot m > n \land \text{Prime}(m) \).

We want to show that for all \( n \) there exist two primes above it, i.e.

\[
\forall n^\mathbb{N} . \exists m, k : \mathbb{N} . m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) .
\]

We can derive this as follows:

- Assume \( n : \mathbb{N} \).
- We have \( \exists m^\mathbb{N} . m > n \land \text{Prime}(m) \).
- So assume \( m : \mathbb{N} \) and \( m > n \land \text{Prime}(m) \).
- We have as well \( \exists k^\mathbb{N} . k > m \land \text{Prime}(k) \).
- So assume \( k : \mathbb{N} \) and \( k > m \land \text{Prime}(k) \).
Then we can conclude
\[ m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \]
and therefore as well
\[ \exists m, k : \mathbb{N}. m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \]
Now by \( \exists \)-elimination twice follows
\[ n : \mathbb{N} \vdash \exists m, k : \mathbb{N}. m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \]
without assuming \( m, k \) as above.
By \( \forall \)-introduction follows
\[ \forall n : \mathbb{N}. \exists m, k : \mathbb{N}. m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \]

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

First step: Under the global assumption
\[ n : \mathbb{N}, m : \mathbb{N}, m > n \land \text{Prime}(m), k : \mathbb{N}, k > m \land \text{Prime}(k) \]
we prove the following
\[
\frac{
  k : \mathbb{N} \\
  m : \mathbb{N} \\
  \exists k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)
}{
  \exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)
}
\]
So we have shown
\[ n : \mathbb{N}, m : \mathbb{N}, m > n \land \text{Prime}(m), k : \mathbb{N}, k > m \land \text{Prime}(k) \vdash \exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) \]

Second step: Under the assumption
\[ n : \mathbb{N}, m : \mathbb{N}, m > n \land \text{Prime}(m) \]
we can conclude
\[ \exists k : \mathbb{N}. k > m \land \text{Prime}(k) \]
and then conclude by \( \exists \)-elimination and Step 1
\[
\frac{
  \exists k : \mathbb{N}. k > m \land \text{Prime}(k) \\
  k : \mathbb{N}, k > m \land \text{Prime}(k) \vdash \exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)
}{
  \exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)
}
\]
Example

Third step: Again we can conclude

\[ n : \mathbb{N} \vdash \exists m \in \mathbb{N}. m > n \land \text{Prime}(m) \]

and then conclude by \(\exists\)-elimination and Step 2

\[ n : \mathbb{N} \vdash \exists m \in \mathbb{N}. m > n \land \text{Prime}(m) \]

Constructive Logic (Cont.)

- We can derive as well a function which depending on \( p : A + B \) decides whether \( p = \text{inl}(a) \) or \( p = \text{inr}(b) \).
- 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.

Constructive (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 \( A \rightarrow B \), and we have
    \[ \lambda x : A. \pi_0(p x) : \forall x : A. C[x, f x] \]
    i.e. we have a proof that \( \forall x : A. C[x, f x] \) 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 \( x \).

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 $\text{halts } x$ 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
  $$\text{halts } x \lor \neg \text{halts } x.$$  

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:
    $$\forall x^\text{TM}. \text{halts } x \lor \neg \text{halts } x.$$  
  - Here $\text{TM}$ 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 $x : \text{TM}$ whether $\text{halts } x$ 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 \( A \lor \neg A \) (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 not provable for all formulae \( A \).

- Jump over remaining slides

Weak vs. strong Disj./Quant.

- 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

\[
A \lor B \quad \text{and} \quad \exists x^A.B
\]

whereas weak disjunction and weak existential quantification are the formulae

\[
\neg(\neg A \land \neg B) \quad \text{and} \quad \neg\forall x^A.\neg B
\]
Weak vs. strong Disj./Quant.

- 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 \( p : A \lor B \) 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.)

- Proof (using classical logic) of
  \[ \exists x^A. B \leftrightarrow (\neg \forall x^A. \neg B) : \]
- We have classically:
  \[ \neg A \supset A : \]
  - If \( A \) is true, then \( \neg A \supset A \) holds.
  - If \( A \) is false, then \( \neg A \) is false, therefore \( \neg A \supset A \) holds.

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 \( p : A \lor \neg A \)
- Jump over the following proofs.

Constructive Logic (Cont.)

- We show intuitionistically \( \neg \exists x^A. B \leftrightarrow \forall x^A. \neg B : \)
  - 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):
  \[ (\exists x^A. B) \leftrightarrow (\neg \exists x^A. B) \leftrightarrow (\forall x^A. \neg B) \]
Constructive Logic (Cont.)

Proof of \( A \lor B \leftrightarrow \neg(\neg A \land \neg B) \):

We show intuitionistically \( \neg(\neg A \lor B) \leftrightarrow (\neg A \land \neg B) \):

Assume \( \neg(\neg A \lor B) \). If \( A \), then \( \neg A \), a contradiction, therefore \( \neg A \).

Similarly we get \( \neg B \), therefore \( \neg A \land \neg B \).

Assume \( \neg A \land \neg B \), show \( \neg(\neg A \lor B) \).

Assume \( A \lor B \). If \( A \), then a contradiction with \( \neg A \), similarly with \( B \).

Now it follows (classically):

\[
(A \lor B) \leftrightarrow \neg(\neg A \lor B) \leftrightarrow \neg(\neg A \land \neg B)
\]

Class. Logic for \( \exists, \lor \)-free Formulae

We show that for formulas \( A \) built from \( \neg, \lor, \land, \forall \) 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.

Class. Logic for \( \exists, \lor \)-free Formulae

Case \( A \equiv \text{Atom } c \).

We make case distinction on \( c \).

If \( c = \text{tt} \), then we have \( A \equiv \top \), which is provable, therefore as well \( \neg \neg A \supset A \).

If \( c = \text{ff} \), 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

\[
\neg A \vdash A
\]

and obtain

\[
\neg \neg A \supset A
\]

Class. Logic for \( \exists, \lor \)-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 \( \neg C \), 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 \( \neg A \), which is \( A \supset \bot \), we get therefore \( \bot \), which completes the proof for this case.
Class. Logic for $\exists$, $\lor$-free Formulae

Case $A \equiv B \land C$, 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 $B \land C$. 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 $B \land C$, i.e. $A$.

Case $A \equiv B$, and we have stability for $B$.

$\neg B \equiv B \supset \bot$.

$\bot \equiv \bot = $ 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 $B \supset \bot \equiv \neg B$.

Class. Logic for $\exists$, $\lor$-free Formulae

Case $A \equiv \forall x B \land C$, 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 $\neg C$, which by the stability of $C$ implies $C$.

So assume $\neg C$ and show $\bot$.

We show $\neg A$.

· Assume $A$, which is $\forall x B \land C$.
· Then we obtain $C$, and by $\neg C$ therefore $\bot$.
· By $\neg\neg A$ we therefore get $\bot$, and are done.

(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:

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

\[ 0 : N \quad (\text{N-I}_0) \]

\[ n : N \quad (\text{N-I}_S) \]

\[ S n : N \]

Primitive Recursion

**Primitive Recursion expresses:**
Assume we have

- $a : N$.
- and, if $n : N$, $x : N$ then $g n x : N$.

Then we can define $f : N \to N$, s.t.

- $f 0 = a$,
- $f (S n) = g n (f n)$.

Example

The function $f : N \to N$ with $f n = 2 \cdot n$ can be defined **primitive recursively** by:

- $f 0 = 0$.
- $f (S n) = S (S (f n))$.

Therefore take in the definition above:

- $a = 0$,
- $g n x = S (S x)$.
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 \( C \)
  \[
  f : \mathbb{N} \rightarrow C
  \]
- Further, \( C \) can now **depend on \( \mathbb{N} \).**
- We obtain the following set of rules:

Rules for the Natural Numbers

**Formation Rule**

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

**Introduction Rules**

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

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

Rules for the Natural Numbers

**Elimination Rule**

\[
C : \mathbb{N} \rightarrow \text{Set} \\
\begin{align*}
a : & C\ 0 \\
g : & (x : \mathbb{N}) \rightarrow C\ x \rightarrow C\ (S\ x) \\
n : & \mathbb{N} \\
P\ C\ a\ g\ n : & C\ n
\end{align*}
\]

\[ (\text{N-EI}) \]

**Equality Rules**

\[
P\ C\ a\ g\ 0 = a \quad (\text{N-Eq}_0)
\]

\[
P\ C\ a\ g\ (S\ n) = g\ n\ (P\ C\ a\ g\ n) \quad (\text{N-Eqs})
\]

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

Elimination into Type

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

**Strong elimination Rule**

\[
n : \mathbb{N} \Rightarrow C[n] : \text{Type} \\
a : C[0] \\
g : (x : \mathbb{N}) \rightarrow C[x] \rightarrow C[S\ x] \\
n : \mathbb{N} \\
\frac{P^\text{Type}_C a\ g\ n : C[n]}{} \quad (\text{N-EI}^\text{Type})
\]

**Strong Equality Rules**

\[
P^\text{Type}_C a\ g\ 0 = a \quad (\text{N-Eq}_0^\text{Type})
\]

\[
P^\text{Type}_C a\ g\ (S\ n) = g\ n\ (P^\text{Type}_C a\ g\ n) \quad (\text{N-Eqs}^\text{Type})
\]
Rules for the Natural Numbers

Note that if we define in the elimination rule \( f := P \ C \ g \) (which is \( \eta \)-equal to \( \lambda n^\mathbb{N}. P \ C \ g \ n \)) 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:

\[
\begin{align*}
f 0 & = a \\
f (S \ n) & = g \ n \ (f \ n)
\end{align*}
\]

Logical Framework Rules for \( \mathbb{N} \)

The more compact notation is:

\[ \begin{align*}
\mathbb{N} & : \text{Set}, \\
0 & : \mathbb{N}, \\
S & : \mathbb{N} \to \mathbb{N}, \\
P & : (C : \mathbb{N} \to \text{Set}) \to C \ 0 \\
& \to (\ (x : \mathbb{N}) \to C \ x \to C \ (S \ x) ) \\
& \to (n : \mathbb{N}) \\
& \to C \ n .
\end{align*}\]

The same equality rules as before.

Natural Numbers in Agda

\( \mathbb{N} \) is defined using \textbf{data}:

\[
\begin{align*}
data \ \mathbb{N} : \text{Set} & \text{ where} \\
Z & : \mathbb{N} \\
S & : \mathbb{N} \to \mathbb{N}
\end{align*}
\]

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

Therefore we have

\[
\begin{align*}
Z & : \mathbb{N} \\
S & : \mathbb{N} \to \mathbb{N}
\end{align*}
\]

Elimination Rules for \( \mathbb{N} \) in Agda

Elimination is represented in Agda as before via case distinction.

Assume we want to define

\[
\begin{align*}
f & : (n : \mathbb{N}) \to A \\
f \ n & = \{! \ !\}
\end{align*}
\]

\( A \) possibly depending on \( n \),

Then we can distinguish the cases \( n = Z \) and \( n = S \ m \) and obtain:

\[
\begin{align*}
f \ Z & = \{! \ !\} \\
f \ (S \ n) & = \{! \ !\}
\end{align*}
\]
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

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

$$f \ n = f \ n$$

exampleNat1.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:

$$g : \mathbb{N} \to \mathbb{N}$$

$$g \ Z = Z$$

$$g \ (S \ n) = g \ {! \ !}$$

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

$$g : \mathbb{N} \to \mathbb{N}$$

$$g \ Z = Z$$

$$g \ (S \ n) = g \ n$$

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:

\[
\begin{align*}
\text{one} &:= S \, Z \\
\text{fib} : \mathbb{N} \to \mathbb{N} \\
\text{fib} \, Z &= \text{one} \\
\text{fib} \, (S \, Z) &= \text{one} \\
\text{fib} \, (S \, (S \, n)) &= \text{fib} \, n + \text{fib} \, (S \, n)
\end{align*}
\]

\text{fib1.agda}

Limitations of Termination Checker

Assume we define the predecessor function

\[
\begin{align*}
\text{pred} : \mathbb{N} \to \mathbb{N} \\
\text{pred} \, Z &= Z \\
\text{pred} \, (S \, n) &= n
\end{align*}
\]

i.e.

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

Limitations of Termination Checker

Then the function

\[
\begin{align*}
f : \mathbb{N} \to \mathbb{N} \\
f \, Z &= Z \\
f \, (S \, n) &= f \, (\text{pred} \, n)
\end{align*}
\]

terminates always

(it returns for all \( n : \mathbb{N} \) the value \( Z \)).

However, check-termination fails.

\text{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:

```agda
infixr 10 _ + _
_ + _ : N → N → N
n + Z = n
n + S m = S (n + m)
```

The definition is correct, since when defining \( n + S m \), \( n + m \) is defined before \( n + S m \).

Because of the line

```agda
infixr 10 _ + _ ,
```

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

Example: Multiplication

Definition

```agda
infixr 20 _ * _
_* _ : N → N → N
n * Z = Z
n * S m = n * m + n
```

Because of the line

```agda
infixr 20 _ * _ ,
```

_ * _ binds more than _ + _.

Remember we had infixr 10 _ + _.

We can use in the definition of _ * _, and can refer in case of \( n * S m \) to \( n * m \), which is defined before \( n * S m \).

Equality on \( \mathbb{N} \)

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

```agda
_==Bool_ : N → N → Bool
Z ==Bool Z = tt
S n ==Bool S m = n ==Bool m
_ ==Bool _ = ff
```

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

Equality on \( \mathbb{N} \)

Then we can define equality _==_ on \( \mathbb{N} \) as follows

```agda
_==_ : N → N → Set
n == m = Atom (n ==Bool m)
```
Equality on $\mathbb{N}$ (Cont.)

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

\[
\begin{align*}
==_N & : \mathbb{N} \to \mathbb{N} \to \text{Set} \\
Z & == Z = \top \\
S \ n & == S \ m = n == m \\
_ & == _ = \bot
\end{align*}
\]

\textit{nat1.agda}

Reflexivity of $==$ (Cont.)

Reflexivity of $==$ is the formula:

\[
\forall n : \mathbb{N}. n == n
\]

Type theoretically this means that we have to prove

\[
\text{refl : Refl}
\]

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

This can now be shown using \textit{pattern matching}:

\[
\begin{align*}
\text{refl : Refl} \\
\text{refl Z} & = \{! !\} \\
\text{refl (S n)} & = \{! !\}
\end{align*}
\]

Reflexivity of $==$

Ref: $\text{Set}$

\[
\text{Refl} = (n : \mathbb{N}) \to n == n
\]

\[
\text{refl : Refl}
\]

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

where the type of the goal is $n == n$.
Reflexivity of == (Cont.)

In order to prove \( \text{refl } Z \), we observe

\[
(Z == Z) = \text{Atom } (Z == \text{Bool } Z) \\
= \text{Atom } \text{tt} \\
= \top
\]

Therefore the goal can be solved by taking \( \text{true} : \top \).

---

Reflexivity of == (Cont.)

The complete proof is as follows:

\[
\text{refl} : \text{Refl} \\
\text{refl } Z = \text{true} \\
\text{refl } (S \, n) = \text{refl } n
\]

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

reflnat.agda

---

Symmetry of ==

Symmetry of == is the formula:

\[
\forall n, m : \text{N}.n == m \rightarrow m == n
\]

Type theoretically this means that we have to prove

\[
\text{Sym} : \text{Set} \\
\text{Sym} = (n \, m : \text{N}) \rightarrow n == m \rightarrow m == n
\]

In Agda this is shown by defining

\[
\text{sym} : \text{Sym} \\
\text{sym } n \, m \, \text{nm} = \{! !\}
\]
Symmetry of == (Cont.)

\[ \text{Sym} : \mathbb{N} \rightarrow \mathbb{N} \]

This can now be shown using case distinction on both \( n \) and \( m \):

\[
\begin{align*}
\text{sym} : \text{Sym} \\
\text{sym} Z \quad Z \quad nm &= \{! !\} \\
\text{sym} Z \quad (S \; m) \quad nm &= \{! !\} \\
\text{sym} (S \; n) \quad Z \quad nm &= \{! !\} \\
\text{sym} (S \; n) \quad (S \; m) \quad nm &= \{! !\}
\end{align*}
\]

For convenience we spell out the type of \( \text{sym} \) in the following.

Symmetry of == (Cont.)

\[ \text{sym} : \mathbb{N} \rightarrow \mathbb{N} \]

\[
\begin{align*}
\text{sym} Z \quad Z \quad nm &= \{! !\} \\
\text{sym} Z \quad (S \; m) \quad nm &= \{! !\} \\
\text{sym} (S \; n) \quad Z \quad nm &= \{! !\} \\
\text{sym} (S \; n) \quad (S \; m) \quad nm &= \{! !\}
\end{align*}
\]

In case \( \text{sym} Z \; (S \; m) \; nm \), we have

\[
\begin{align*}
\text{sym} Z \; (S \; m) \; nm &= \{! !\} \\
\text{nm} : \mathbb{N} \Rightarrow S \; m = \bot
\end{align*}
\]

so there is no element in \( \text{nm} \), we can solve it as

\[
\text{sym} Z \; (S \; m) \; ()
\]

Symmetry of == (Cont.)

\[ \text{sym} : \mathbb{N} \rightarrow \mathbb{N} \]

\[
\begin{align*}
\text{sym} Z \quad Z \quad _ &= \text{true} \\
\text{sym} Z \quad (S \; m) \quad nm &= \{! !\} \\
\text{sym} (S \; n) \quad Z \quad nm &= \{! !\} \\
\text{sym} (S \; n) \quad (S \; m) \quad nm &= \{! !\}
\end{align*}
\]

In case \( \text{sym} Z \; (S \; m) \; nm \), the goal is

\[
(Z == Z) = \top
\]

which can be solved by using \( \text{true} \).

The argument \( nm \) is irrelevant and can be replaced by \( _ \).
Symmetry of == (Cont.)

In case \( \text{sym} (S\ n) (S\ m)\ nm\), we have that the type of the goal is
\[
(S\ m == S\ n) = (m == n)
\]
This goal can be solved by
\[
\text{sym}\ n\ m\ nm : m == n
\]
which is type correct since
\[
\text{nm} : (S\ n == S\ m) = (n == m)
\]

Symmetry of == (Cont.)

The complete proof is as follows:
\[
\text{sym} : (n\ m : \mathbb{N}) \rightarrow n == m \rightarrow m == n
\]
\[
\text{sym}\ Z\ Z\ _ = \text{true}
\]
\[
\text{sym}\ Z\ (S\ m)\ ()
\]
\[
\text{sym} (S\ n)\ Z\ ()
\]
\[
\text{sym} (S\ n)\ (S\ m)\ nm\ =\ \{!\ !\}
\]

Note that this code termination checks, since in the last equation \( \text{sym} n m nm\) is defined before
\[
\text{sym} (S\ n)\ (S\ m)\ nm.
\]

symnat.agda

Symmetry of == (Cont.)

In the cases
\[
\text{sym} Z (S\ m)\ nm\text{ and}
\]
\[
\text{sym} (S\ n) Z\ nm
\]
we have that \( nm\) is an element of \( \bot\), and the goal is \( \bot\).
So we can, instead of using empty case distinction on \( nm\), return the proof \( nm\) and obtain the following:

Symmetry of == (Cont.)

\[
\text{sym} : (n\ m : \mathbb{N}) \rightarrow n == m \rightarrow m == n
\]
\[
\text{sym} Z\ Z\ _ = \text{true}
\]
\[
\text{sym} Z (S\ m)\ nm\ =\ nm
\]
\[
\text{sym} (S\ n) Z\ nm\ =\ nm
\]
\[
\text{sym} (S\ n) (S\ m)\ nm\ =\ \text{sym} n m nm
\]
symnat2.agda
Example: < on $\mathbb{N}$

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

\[
\begin{align*}
_<\text{Bool}_\_ : & \mathbb{N} \to \mathbb{N} \to \text{Bool} \\
_ & <\text{Bool} \ Z = \text{ff} \\
 Z <\text{Bool} \ S \ m = \text{tt} \\
S \ n <\text{Bool} \ S \ m = n <\text{Bool} \ m
\end{align*}
\]

\[< \_ : \mathbb{N} \to \mathbb{N} \to \text{Set} \]

\[n < m = \text{Atom} \ (n <\text{Bool} \ m) \]

lessnat1.agda

---

Tuples of Length n

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

\[
\begin{align*}
data \ \text{Nil} : \text{Set} & \text{ where} \\
[ ] & : \text{Nil} \\
data \ \text{Cons} \ (A \ B : \text{Set}) & : \text{Set} \text{ where} \\
\_::\_ : & A \to B \to \text{Cons} \ A \ B
\end{align*}
\]

Now we can define

\[
\begin{align*}
\text{Tuple} : \text{Set} & \to \mathbb{N} \to \text{Set} \\
\text{Tuple} \ A \ Z & = \text{Nil} \\
\text{Tuple} \ A \ (S \ n) & = \text{Cons} \ A \ (\text{Tuple} \ A \ n)
\end{align*}
\]

Therefore,

\[\text{Tuple} \ A \ n = \text{Cons} \ A \ (\underbrace{\text{Cons} \ A \cdots (\text{Cons} \ A \ \text{Nil}) \cdots}_{n \ \text{times}}) \]

The elements of Tuple $A \ n$ are

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

for elements $a_1, \ldots, a_n$ of $A$.

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

\[
\text{Tuple}(A, 0) := \{\langle \rangle \}, \\
\text{Tuple}(A, n+1) := \{\langle a_1, \ldots, a_{n+1} \rangle \mid a_1, \ldots, a_{n+1} \in A\}.
\]

If we define

\[
[\ ] := \langle \rangle, \\
a_1 :: \langle a_2, \ldots, a_{n+1} \rangle := \langle a_1, \ldots, a_{n+1} \rangle,
\]

then this reads:

\[
\text{Tuple}(A, 0) := \{[\] \}, \\
\text{Tuple}(A, n+1) := \{a :: b \mid a \in A \land b \in \text{Tuple}(A, n)\}.
\]

Componentwise Sum of n-Tuples

We define component-wise sum of tuples of length n.

Using mathematical notation, this sum for instance as follows:

\[
\langle 2, 3, 4 \rangle + \langle 5, 6, 7 \rangle = \langle 7, 9, 11 \rangle.
\]
(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:

\[
\text{infixr 20 } _::_ \\
\]

\[
data \text{List } (A : \text{Set}) : \text{Set} \text{ where} \\
[\ ] : \text{List } A \\
_::_ : A \rightarrow \text{List } A \rightarrow \text{List } A
\]

Elimination Principle for Lists

The elimination principle is structural recursion on lists:

- Assume \( A : \text{Set} \)
- \( C : \text{Set} \), depending on \( l : \text{List } A \).

Then we can define

\[
f : (l : \text{List } A) \rightarrow C \\
f [\ ] = \{! \} \\
f (a :: l) = \{! \}
\]

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

Example: Length of a List

\[
\text{length : List } \mathbb{N} \rightarrow \mathbb{N} \\
\text{length } [\ ] = Z \\
\text{length } (_:: l) = S (\text{length } l)
\]

Example: sumlist

\( \text{sumlist } l \) will compute the sum of the elements of list \( l \).

\[
\text{sumlist : List } \mathbb{N} \rightarrow \mathbb{N} \\
\text{sumlist } [\ ] = Z \\
\text{sumlist } (n :: l) = n + \text{sumlist } l
\]
Interesting Exercise

Define

\[ _{++} : \{ A : \text{Set} \} \rightarrow \text{List} A \rightarrow \text{List} A \rightarrow \text{List} A , \]

s.t. \( l ++ l' \) is the result of appending the list \( l' \) at the end of list \( l \).

E.g., if \( a, b, c, d \) are elements of \( A \), then

\[
\begin{align*}
a :: b :: [ ] & \quad ++ \quad c :: d :: [ ] \\
= a :: b :: c :: d :: [ ]
\end{align*}
\]

(list.agda)

(i) Universes

A universe \( U \) is a set, the elements of which are codes for sets.

So we have

\[ U : \text{Set}, \]
\[ T : U \rightarrow \text{Set} \text{ (the decoding function)} \]

We consider in the following a universe closed under

\[ \bot, \top, \text{Bool}, N, +, \Sigma, \]
the dependent function type.

Rules for the Universe

Formation Rule

\[ U : \text{Set} \quad (U-F) \]
\[ a : U \quad \frac{}{T a : \text{Set}} \quad (T-F) \]

Introduction and Equality Rules

\[ \hat{\bot} : U \quad (U-I_{\bot}) \]
\[ T (\hat{\bot}) = \bot : \text{Set} \quad (T-Eq_{\bot}) \]

\[ \hat{\top} : U \quad (U-I_{\top}) \]
\[ T (\hat{\top}) = \top : \text{Set} \quad (T-Eq_{\top}) \]

\[ \hat{\text{Bool}} : U \quad (U-I_{\text{Bool}}) \]
\[ T (\hat{\text{Bool}}) = \text{Bool} : \text{Set} \quad (T-Eq_{\text{Bool}}) \]

\[ \hat{N} : U \quad (U-I_{\hat{N}}) \]
\[ T (\hat{N}) = N : \text{Set} \quad (T-Eq_{\hat{N}}) \]
**Rules for the Universe**

**Introduction and Equality Rules (Cont.)**

\[
\frac{a : U \quad b : U}{a \uplus b : U} \quad \text{(U-}\uplus\text{)}
\]

\[
T (a \uplus b) = T a + T b : \text{Set} \quad \text{(T-Eq}_\uplus\text{)}
\]

\[
\frac{a : U \quad b : T a \rightarrow U}{\Sigma a \ b : U} \quad \text{(U-}\Sigma\text{)}
\]

\[
T (\Sigma a \ b) = \Sigma (T a) (\lambda x. T (b x)) : \text{Set} \quad \text{(T-Eq}_\Sigma\text{)}
\]

**Applications of the Universe**

- Ordinary elimination rules don’t allow to eliminate into \text{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 \text{Set} and use \text{T} to obtain the required function.

**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 \text{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

Example: Define

\[
\begin{align*}
\text{Atom} : & \quad \text{Bool} \to U, \\
\text{Atom} := & \quad \text{Case Bool} (\lambda x. \text{Bool} \to U) \Rightarrow U,
\end{align*}
\]

\[
\begin{align*}
\text{Atom} : & \quad \text{Bool} \to \text{Set}, \\
\text{Atom} : & \quad \lambda x. \text{Bool} \Rightarrow \text{Set} (\text{Atom } x),
\end{align*}
\]

Then

\[
\begin{align*}
\text{Atom } \text{tt} &= \top, \\
\text{Atom } \text{ff} &= \bot.
\end{align*}
\]

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.)

\[
\text{mutual}
\]

\[
\begin{align*}
\text{data } U : & \quad \text{Set} \\
& \begin{align*}
\bot & : U \\
\top & : U \\
\text{Bool} & : U \\
\text{N} & : U \\
\_+_ & : U \to U \to U \\
\Sigma & : (a : U) \to (\text{T } a \to U) \to U \\
\Pi & : (a : U) \to (\text{T } a \to U) \to U
\end{align*}
\end{align*}
\]
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.

```agda
data A : Set where
  C₁ : (a₁ : A₁) → · · · → (aₙ₁ : Aₙ₁) → A
  C₂ : (a₁ : A₂₁) → · · · → (aₙ₂ : A₂ₙ₂) → A
  · · ·
  Cₘ : (a₁ : Aₘ₁) → · · · → (aₙₘ : Aₘₙₘ) → A
```

Meaning of “data”

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

```agda
C₁ : (a₁₁ : A₁₁) → · · · → (aᵢₙ₁ : Aᵢₙ₁) → A
```

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 `Aᵢⱼ` we can make use of `A`. However, it is difficult to understand `A`, if we have negative occurrences of `A`.

Example:

```agda
data A : Set where
  C : (A → A) → A
```

What is the least set `A` having a constructor

```agda
C : (A → A) → A
```

If we have constructed some elements of `A` already, find a function `f : A → A`, and add `C f` to `A`, then `f` might no longer be a function `A → A`. (`f` applied to the new element `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.
A “good” definition is the set of lists of natural numbers, defined as follows:

\[
\text{data } \text{NList} : \text{Set where}
\]
\[
\begin{array}{ll}
[] & : \text{NList} \\
_::_ & : \text{N} \rightarrow \text{NList} \rightarrow \text{NList}
\end{array}
\]

The constructor \_::\_ of NList refers to NList, but in a positive way: We have: if \( a : \text{N} \) and \( l : \text{NList} \), then

\[(a :: l) : \text{NList}.\]

In general:

\[
\text{data } A : \text{Set where}
\]
\[
\begin{array}{ll}
C_1 & : (a_1 : A_1^1) \rightarrow (a_2 : A_2^1) \rightarrow \cdots (a_{n_1} : A_{n_1}^1) \rightarrow A \\
C_2 & : (a_1 : A_1^2) \rightarrow (a_2 : A_2^2) \rightarrow \cdots (a_{n_2} : A_{n_2}^2) \rightarrow A \\
\vdots \\
C_m & : (a_1 : A_1^m) \rightarrow (a_2 : A_2^m) \rightarrow \cdots (a_{n_m} : A_{n_m}^m) \rightarrow A
\end{array}
\]

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.

If we add \( a :: l \) to NList, the reason for adding it (namely \( l : \text{NList} \)) is not destroyed by this addition.

So we can “construct” the set NList by

- starting with the empty set,
- adding [] and
- closing it under \_::\_ whenever possible.

Because we can “construct” NList, the above is an acceptable definition.

The definitions of finite sets, \( \Sigma A B \), \( A + B \) and \( \text{N} \) were strictly positive algebraic types.
One further Example

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

```agda
data BinTree : Set where
  leaf : BinTree
  branch : BinTree → BinTree → BinTree
```

- 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:

```agda
mutual
data Even : Set where
  Z : Even
  S : Odd → Even
data Odd : Set where
  S' : Even → Odd
```

- 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 $A_{ij} = B_1 \to A$ or even $A_{ij} = B_1 \to \cdots \to B_l \to A$, where $A$ is one of the types introduced simultaneously.

- Example (called “Kleene’s O”):

```agda
data O : Set where
  leaf : O
  succ : O → O
  lim : (N → O) → O
```

- The last definition is unproblematic, since, if we have $f : N \to O$ and construct $\text{lim } f$ 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 $f(C \ a_1 \ \cdots \ a_n)$, we can refer only to $f$ applied to elements used in $C \ a_1 \ \cdots \ a_n$. 

Examples

- For instance, in the Bintree example, when defining
  \[ f : \text{Bintree} \rightarrow A \]
  by case-distinction, then the definition of
  \[ f \ (\text{branch } l \ r) \]
  can make use of \( f \ l \) and \( f \ r \).

- In the example of \( O \), when defining
  \[ g : O \rightarrow A \]
  by case-distinction, then the definition of
  \[ g \ (\text{lim } f) \]
  can make use of \( g \ (f \ n) \) for all \( n : \mathbb{N} \).