4. Data Types

(a) The set of Booleans.
(b) The finite sets.
(c) Atomic formulae and the traffic light example. (The example will probably be omitted 2005).
(d) The disjoint union of sets and disjunction.
(e) The $\Sigma$-set. (Will be omitted 2005.)
(f) Predicate Logic in Dependent Type Theory.
(g) Natural Deduction and Dependent Type Theory. (Will be largely omitted 2005).
(h) The set of natural numbers.
(i) Lists. (Will probably be omitted 2005.)
(j) Universes. (Will probably be omitted 2005.)
(k) Algebraic types. (Will be omitted 2005.)
(a) The Set of Booleans

**Formation Rule**

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

**Introduction Rules**

\[ \text{tt} : \text{Bool} \quad (\text{Bool-I}_0) \quad \text{ff} : \text{Bool} \quad (\text{Bool-I}_1) \]

**Elimination Rule**

\[
\frac{C : \text{Bool} \rightarrow \text{Set} \quad \text{ic} : C \quad \text{tt} \quad \text{ec} : C \quad \text{ff} \quad \text{cond} : \text{Bool}}{	ext{Case}_{\text{Bool}} C \quad \text{ic ec cond} : C \quad \text{cond} \quad (\text{Bool-El})}
\]
Equality Rules

\[
\begin{align*}
C : \text{Bool} \to \text{Set} & \quad ic : C \ \text{tt} \quad ec : C \ \text{ff} & \quad (\text{Bool-Eq}_0) \\
\text{Case}_{\text{Bool}} C \ ic \ ec \ \text{tt} = ic : C \ \text{tt} & \\
C : \text{Bool} \to \text{Set} & \quad ic : C \ \text{tt} \quad ec : C \ \text{ff} & \quad (\text{Bool-Eq}_1) \\
\text{Case}_{\text{Bool}} C \ ic \ ec \ \text{ff} = ec : C \ \text{ff} 
\end{align*}
\]

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

In the above

\( \texttt{tt} \) stands for true, \( \texttt{ff} \) stands for false.

\( ic \) stands for “if-case”, \( ec \) for “else-case”.

\( \texttt{cond} \) for “condition”.

Therefore \( \texttt{Case}_{\texttt{Bool}} \ C \ ic \ ec \ b \) can be read as

\[
\text{if } b \text{ then } ic \text{ else } ec
\]

where the additional argument \( C \) is required in order to determine the type of \( ic \), of \( ec \), 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 have $C : \text{Bool} \rightarrow \text{Set}$, since the set into which we are eliminating might depend on the Boolean value.

That is necessary in order to define functions $f : (b : \text{Bool}) \rightarrow D$ where $D$ depends on $b$.

If we define

$$f := \lambda b^{\text{Bool}}. \text{Case}_{\text{Bool}} C \ ic \ ec \ b$$

we have:

- $f \ tt : C \ tt$.
- $f \ ff : C \ ff$.
- $f : (b : \text{Bool}) \rightarrow C \ b$. 
The argument $C$ above has no computational content. It is not needed in order to compute $\text{Case}_{\text{Bool}} C \ ic \ ec \ tt$ and $\text{Case}_{\text{Bool}} C \ ic \ ec \ ff$.

$C$ 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.
We can write the elimination rule in a more compact but less readable way:

\[ \text{Case}_{\text{Bool}} : (C' : \text{Bool} \rightarrow \text{Set}) \rightarrow (ic : C' \text{ tt}) \rightarrow (ec : C' \text{ ff}) \rightarrow (\text{cond} : \text{Bool}) \rightarrow C' \text{ cond} \]

\text{tt, ff are the constructors of \text{Bool}.}
Remarks (Cont.)

Notice that we then get for \( C : \text{Bool} \rightarrow \text{Set}, \)
\( ic : C \text{ tt}, ec : C \text{ ff} \)

\[ f := \text{Case}_{\text{Bool}} C \text{ ic ec}, \]
\[ : (\text{cond} : \text{Bool}) \rightarrow C \text{ cond} \]

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

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.
Remarks (Cont.)

This is similar to the definition of for instance (+) in **curried form** in Haskell

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

\[(+) \text{ 3 is the function which takes an integer and adds to it 3.}
\]

**Shorter** than writing \(\lambda x^{\text{int}}.3 + x\).
Remarks (Cont.)

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

\[
\text{Names } : \text{Bool} \rightarrow \text{Set}, \\
\text{Names } tt = \text{MaleNames}, \\
\text{Names } ff = \text{FemaleNames}.
\]
Select Example

Then we can define the function

\[
\text{SelectBool} : (b : \text{Bool}) \rightarrow \text{Names } b
\]

\[
\text{SelectBool } \text{tt} = \text{Tim} \\
\text{SelectBool } \text{ff} = \text{Sara}
\]

as follows:

\[
\text{SelectBool} = \text{Case}_{\text{Bool}} \text{Names } \text{Tim } \text{Sara}
\]

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

\[
\text{SelectBool} = \lambda b^{\text{Bool}}. \text{Case}_{\text{Bool}} (\lambda d^{\text{Bool}}. \text{Names } d) \text{ Tim } \text{Sara } b
\]
Select Example

We verify the correctness of \texttt{SelectBool}:

\[
\texttt{SelectBool tt} = \texttt{CaseBool Names Tim Sara tt = Tim}, \\
\texttt{SelectBool ff} = \texttt{CaseBool Names Tim Sara ff = Sara}.
\]

Jump over AND
Example: AND

- We want to introduce conjunction

  \[ \text{AND} : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \ . \]

- This will be of the form

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

  for some term \( t \).

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

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

  for some \( e, f \).
Example: AND

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

- \(C\) 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: AND

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.
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: AND

So

\[ \text{AND} = \lambda(b, c : \text{Bool}). \text{Case}_{\text{Bool}} (\lambda d^{\text{Bool}}. \text{Bool}) \; e \; f \; b \]

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: AND

In total we define therefore

\[ \text{AND} = \lambda(b, c : \text{Bool}).\text{Case}_{\text{Bool}} (\lambda d^{\text{Bool}}.\text{Bool}) \ c \ ff \ b \]
\[ : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \]

We verify the correctness of this definition:

\[ \text{AND} \ \text{tt} \ c = \text{Case}_{\text{Bool}} (\lambda d^{\text{Bool}}.\text{Bool}) \ c \ ff \ \text{tt} = c. \] as desired.

\[ \text{AND} \ \text{ff} \ c = \text{Case}_{\text{Bool}} (\lambda d^{\text{Bool}}.\text{Bool}) \ c \ ff \ \text{ff} = \text{ff}. \] Correct as desired.

Jump over derivation of AND
Derivation of AND

We derive in the following \( \text{AND} : \text{Bool} \to \text{Bool} \to \text{Bool} \).

We write \( \text{Bool} \), if it is a type in **boldface red**, and if it is a term, in *italic blue*.
Derivation of AND

First we derive

\[ b : \text{Bool}, c : \text{Bool} \implies \lambda d^{\text{Bool}}. \text{Bool} : \text{Bool} \rightarrow \text{Set} : \]

\[
\frac{\text{Bool} : \text{Set} \quad (\text{Context}_1)}{b : \text{Bool} \implies \text{Context} \quad (\text{Bool-F})}
\]

\[
\frac{b : \text{Bool} \implies \text{Bool} : \text{Set} \quad (\text{Context}_1)}{b : \text{Bool}, c : \text{Bool} \implies \text{Context} \quad (\text{Bool-F})}
\]

\[
\frac{b : \text{Bool}, c : \text{Bool} \implies \text{Context} \quad (\text{Context}_1)}{b : \text{Bool}, c : \text{Bool} \implies \text{Bool} : \text{Set} \quad (\text{Bool-F})}
\]

\[
\frac{b : \text{Bool}, c : \text{Bool}, d : \text{Bool} \implies \text{Context} \quad (\text{Bool-F}')}{b : \text{Bool}, c : \text{Bool}, d : \text{Bool} \implies \text{Bool} : \text{Set} \quad (\rightarrow -\text{I})}
\]

\[ b : \text{Bool}, c : \text{Bool} \implies \lambda d^{\text{Bool}}. \text{Bool} : \text{Bool} \rightarrow \text{Set} \]
Derivation of AND

We derive

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

(using part of the derivation above):

\[ \cdots \]

\[ b:\text{Bool}, c: \text{Bool}, d: \text{Bool} \Rightarrow \text{Context} \quad (\text{Bool}^\text{F}) \]

\[ b: \text{Bool}, c: \text{Bool}, d: \text{Bool} \Rightarrow \text{Bool : Set} \quad (\text{Bool}^\text{I}_0) \]

\[ b: \text{Bool}, c: \text{Bool} \Rightarrow (\lambda d^\text{Bool}. \text{Bool}) \text{ tt = Bool : Set} \quad (\text{Sym}^\text{Elem}) \]

\[ b: \text{Bool}, c: \text{Bool} \Rightarrow \text{Bool} = (\lambda d^\text{Bool}. \text{Bool}) \text{ tt : Set} \]
Derivation of AND

Similarly follows

\[ b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Bool} = (\lambda d^\text{Bool}.\text{Bool}) \text{ ff : Set} \]
Derivation of AND

Using part of the proof above, we derive

\[ b : \text{Bool}, \ c : \text{Bool} \implies c : (\lambda d. \text{Bool}) \ tt \]

... \[ b : \text{Bool}, c : \text{Bool} \vdash \text{Context} \] (Ass) \[ b : \text{Bool}, c : \text{Bool} \vdash c : \text{Bool} \]

\[ b : \text{Bool}, c : \text{Bool} \vdash (\lambda d. \text{Bool}) \ tt : \text{Set} \] (Transfer0)

We derive using \[(\text{Transfer}_0)\]

\[ b : \text{Bool}, c : \text{Bool} \implies ff : (\lambda d. \text{Bool}) \ ff \]

... \[ b : \text{Bool}, c : \text{Bool} \vdash \text{Context} \] (Bool^\text{I1}) \[ b : \text{Bool}, c : \text{Bool} \vdash \text{ff : Bool} \]

\[ b : \text{Bool}, c : \text{Bool} \vdash \text{ff : Set} \] (Transfer0)

\[ b : \text{Bool}, c : \text{Bool} \vdash \text{ff : (\lambda d. \text{Bool}) ff} \]
Derivation of AND

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

\[
\begin{align*}
\ldots \\
\frac{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context}}{b : \text{Bool}, c : \text{Bool} \Rightarrow b : \text{Bool}} \quad (\text{Ass})
\end{align*}
\]
Derivation of AND

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

\[
\begin{align*}
\text{b:} Bool, c: Bool \Rightarrow \lambda d^{bool}. Bool: Bool \rightarrow Set \\
b: Bool, c: Bool \Rightarrow c: (\lambda d^{bool}. Bool) \text{ tt} \\
b: Bool, c: Bool \Rightarrow ff: (\lambda d^{bool}. Bool) \text{ ff} \\
\hline
b: Bool, c: Bool \Rightarrow b: Bool \quad \text{(Bool-El)} \\
b: Bool \Rightarrow \lambda c^{bool}. \text{Case}_Bool (\lambda d^{bool}. Bool) \text{ c ff b: Bool} \quad \text{(→-I)} \\
\hline
\lambda (b, c: Bool). \text{Case}_Bool (\lambda d^{bool}. Bool) \text{ c ff b: Bool} \rightarrow Bool \rightarrow Bool \quad \text{(→-I)}
\end{align*}
\]
Elimination into Type

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

Elimination Rule into Type

\[
\frac{C : \text{Bool} \rightarrow \text{Type} \quad \text{ic} : C \ \text{tt} \quad \text{ec} : C \ \text{ff} \quad \text{cond} : \text{Bool}}{\text{Case}^\text{Type}_{\text{Bool}} C \ \text{ic} \ \text{ec} \ \text{cond} : C \ \text{cond}} \quad (\text{Bool-El}^\text{Type})
\]

Equality Rules into Type

\[
\begin{align*}
\frac{C : \text{Bool} \rightarrow \text{Type} \quad \text{ic} : C \ \text{tt} \quad \text{ec} : C \ \text{ff}}{\text{Case}^\text{Type}_{\text{Bool}} C \ \text{ic} \ \text{ec} \ \text{tt} = \text{ic} : C \ \text{tt}} \quad (\text{Bool-Eq}_0^\text{Type}) \\
\frac{C : \text{Bool} \rightarrow \text{Type} \quad \text{ic} : C \ \text{tt} \quad \text{ec} : C \ \text{ff}}{\text{Case}^\text{Type}_{\text{Bool}} C \ \text{ic} \ \text{ec} \ \text{ff} = \text{ec} : C \ \text{ff}} \quad (\text{Bool-Eq}_1^\text{Type})
\end{align*}
\]
Elimination into Type (Cont.)

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

Unfortunately, Bool, True, False are reserved keywords in Agda, so we use Bool’, True’, False’ instead.

We introduce Bool’ by simply listing its constructors (similarly to Haskell syntax):

\[
\text{data Bool’ = tt | ff}
\]

This introduces as well constants

- \( \text{tt :: Bool’} \)
- \( \text{ff :: Bool’} \)
Bool in Agda

With this syntax, each constructor can occur at most once in a data type,
i.e. we cannot define a second type having constructor \( \texttt{tt} \),
  e.g. for defining \( \text{True'} \) (which is used later):

\[
\text{data True'} = \text{tt}
\]
The definition of Bool above is treated in Agda as an abbreviation for the following three more fundamental Agda definitions:

\[
\begin{align*}
\text{Bool}' & : \text{Set} \\
& = \text{data tt | ff} \\
\text{tt} & : \text{Bool}' \\
& = \text{tt@Bool}' \\
\text{ff} & : \text{Bool}' \\
& = \text{ff@Bool}'
\end{align*}
\]
Notation for Constructors

The official notation for a constructor of a set $A$ is

$$C@A$$

The notation $C@A$ is, what is displayed, when evaluating expressions in Agda.

This notation is necessary, since a constructor might belong to different sets.

For instance we can introduce both

$$\text{Bool'} :: \text{Set} = \text{data } \text{tt } | \text{ff}$$
$$\text{True'} :: \text{Set} = \text{data } \text{tt}$$

In this situation we need to be able to distinguish between $\text{tt}@\text{Bool'}$ and $\text{tt}@\text{True'}$ in order to get decidable type checking.
Notation for Constructors

It was not possible in Agda to avoid the use of @ by forcing the user to use different names for constructors; we will later introduce as well sets of the form

\[ D \ (a :: A) :: \text{Set} = \text{data } C \ldots \]

Then \( C@D \ldots \) can be a constructor of \( D a \) for any \( a : A \).

Using \( C \) alone would cause problems with decidable type checking.

If Agda can resolve the type itself, one can write however \( C@_ \) instead of \( C@A \).
Notation for Constructors

- It is recommended to avoid the use of the symbol \(@\), and for instance define in the above situation by hand

\[
C (a :: A) :: D = C @ _ \ldots
\]

- The abbreviation

\[
data \text{Bool'} = \text{tt} | \text{ff}
\]
does this automatically by

- defining the set \(\text{Bool}'\),
- defining \(\text{tt}\) as \(\text{tt}@_\) and
- defining \(\text{ff}\) as \(\text{ff}@_\).
Notation for Constructors

However the abbreviation

\[
\text{data } A = C \mid D \mid \cdots
\]

can be used only if
- one is defining a set \( A \) (and not a type \( A \)),
- and if the set one is defining has no parameter.

So it cannot be used in order to define

\[
D \ (a :: A) \\
:: \text{Set} \\
= \text{data } C \ \cdots
\]
Evaluation of terms in Agda

Agda has several methods for evaluating expressions:

- `agda-compute-whnf`, “Compute weak head normal form”,
- `agda-compute-whnfs`, “Compute weak head normal form strict”,
- `agda-nfC`, “Compute to a depth”,
- `agda-nfC100`, “Compute to depth 100”.
Evaluation of terms in Agda

The above mentioned methods can be executed (directly or by using the goal-menu), while in a goal.

An expression typed into the goal will be taken as default input to that function.

But that can be modified.

The methods follow different evaluating strategies.

- **Compute weak head normal form** reduces a term until it starts with a constructor or the outer most function doesn’t reduce any further, even if its arguments are evaluated.

- **Compute to depth 100** seems to work best in most cases.
Elimination in Agda is **based on case distinction.**

Assume we want to define

\[ f : \text{Bool}' \to \text{Bool}', \text{ s.t.} \]

\[ f \; \text{tt} = \text{ff}, \]

\[ f \; \text{ff} = \text{tt}. \]

So we have the goal:

\[
\begin{align*}
f \quad (x :: \text{Bool}') \\
:: & \quad \text{Bool}' \\
= & \quad \{! \!} \\
\end{align*}
\]
We can then type into the goal \( x \) and choose the menu item “\texttt{agda-case}”. This introduces a case distinction by the constructor used for introducing \( x \): \( x \) could have been introduced as \texttt{tt} or \texttt{ff}.

The goal expands to:

\[
f \ (x :: \texttt{Bool}^{\prime})
\]
\[
:: \texttt{Bool}^{\prime}
\]
\[
= \text{case } x \text{ of}
\]
\[
\quad (\texttt{tt}) \rightarrow \{! \!\}
\]
\[
\quad (\texttt{ff}) \rightarrow \{! \!\}
\]
Case Distinction (Cont.)

The *value of x* in the first goal can be tested as follows:

Position the cursor in the first goal and choose one of the methods for evaluating expressions, e.g. *compute weak head normal form strict*.

Then type into the mini-buffer $x$.

One gets the answer

$t\text{t@}. $
Alternatively, check, the cursor being in that goal, the context
(use goal-menu “agda-context”):
It contains

\[ x :: \text{Bool} = \text{tt@}. \]

Similarly one finds that in the second goal \( x \) is \( \text{ff@}. \)
Now we can solve the new goals by inserting
- \( ff \) into the first one,
- \( tt \) into the second one.

We obtain a function:

\[
\begin{align*}
f(x :: \text{Bool'}) &:: \text{Bool'} \\
&= \text{case } x \text{ of} \\
&\quad (tt) \rightarrow ff \\
&\quad (ff) \rightarrow tt
\end{align*}
\]

\(f x\) is the **negation of** \(x\).
Testing the Defined Function

We can test our function by using one of the evaluation methods of Agda, e.g. **compute weak head normal form strict**.

We have to create a goal for this.

- The reduction machinery is **context dependent**.
- The context depends on where in the buffer we are.
- See the above example where \( x \) was depending on the goal \( \texttt{tt} \) or \( \texttt{ff} \).

Not every place in the buffer is a good place.

**Good places for context are goals**, and that’s **the only place** where Agda allows us to **compute the weak head normal form of expressions**.
Testing the Defined Function

So we type in a dummy goal:

\[
\begin{align*}
\text{test} &:: \text{Set} \\
&= \{! !\}
\end{align*}
\]

move to the new goal

choose \textbf{compute weak head normal form strict} or another evaluation method of Agda,

and type into the mini-buffer \( f \ tt \).

The result shown is \texttt{ff@\_}.
(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**

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

(for \( k = 0, \ldots, n - 1 \))
Rules for $\text{Fin}_n$

**Elimination Rule**

\[
\begin{align*}
C &: \text{Fin}_n \rightarrow \text{Set} \\
  s_0 &: C \ A_0^n \\
  s_1 &: C \ A_1^n \\
  \ldots \\
  s_{n-1} &: C \ A_{n-1}^n \\
  a &: \text{Fin}_n \\
\end{align*}
\]

\[\begin{array}{c}
\text{Case}_n \ C \ s_0 \ \ldots \ s_{n-1} \ a : C \ a
\end{array}\]

($\text{Fin}_n\text{-El}$)
The Finite Sets (Cont)

Equality Rules

\[ C : \text{Fin}_n \rightarrow \text{Set} \]

\[ s_0 : C \ A_0^n \]

\[ s_1 : C \ A_1^n \]

\[ \ldots \]

\[ s_{n-1} : C \ A_{n-1}^n \]

\[ \text{Case}_n C \ s_0 \ldots s_{n-1} A_k^n = s_k : C \ A_k^n \] (Fin\_n-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:

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

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

  \[
  \text{Case}_n C \ s_0 \ldots s_{n-1} \ A_k^n = s_k
  \]
More Compact Elimination Rules

\[ \text{Case}_n : (C : \text{Fin}_n \rightarrow \text{Set}) \rightarrow (s_0 : C \ A^n_0) \rightarrow \cdots \rightarrow (s_{n-1} : C \ A^n_{n-1}) \rightarrow (a : \text{Fin}_n) \rightarrow C \ a \]
Elimination into Type

Similarly as for \( \text{Bool} \)' we can write down **elimination rules**, where
\[
C : \text{Fin}_n \to \text{Type} \quad \text{(instead of } C : \text{Fin}_n \to \text{Set}).
\]

This can be done for all sets defined later as well.
Rules for True

True is the special case $\text{Fin}_n$ for $n = 1$ (we write $\text{true}$ for $A_0^1$):

Formation Rule

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

Introduction Rules

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

Elimination Rule

$$C : \text{True} \rightarrow \text{Set} \quad c : C \text{ true} \quad t : \text{True} \quad (\text{True-El})$$

\[ \text{Case}_\text{True} \quad c \ t : C \ t \]
Rules for True

Equality Rule

\[
\frac{C : \text{True} \rightarrow \text{Set} \quad c : C \text{ true}}{\text{Case}_{\text{True}} \quad c \text{ true} = c : C \text{ true}} \quad \text{(True-Eq)}
\]

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

Jump over next slide (advanced material)
Case\textsubscript{True} is **computationally not very interesting**.

- Case\textsubscript{True} \(c\) is the constant function \(\lambda x^{\text{True}}.c\).
- However, in Agda we might not be able to derive

\[
\lambda t^{\text{True}'} . c : (t : \text{True}') \rightarrow C \ t
\]

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

- From an element of \(C \ \text{true}\) we obtain an element of \(C \ t\) **for every** \(t : \text{True}\).
- So there is no \(C : \text{True} \rightarrow \text{Set}\) s.t. \(C \ \text{true}\) is inhabited, but \(C \ x\) is not inhabited for some other \(x : \text{True}\).
- This means that all elements of \(x\) of type \(\text{True}\) are **indistinguishable from true**, i.e. they are **identical to true**.
- This equality is called **Leibnitz equality**.
Formulae as Types

In type theory, formulas are certain types.

A formula expressed as a type, is **type-theoretically true**, if it has an element.

The elements of such a type are **proofs of this formula**.

Therefore

**Truth in type theory means provability.**

True has exactly one proof, and corresponds therefore to the always **always true formula**.
Rules for False

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

**Formation Rule**

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

**There is no Introduction Rule**

**Elimination Rule**

$$ \frac{C : \text{False} \rightarrow \text{Set} \quad f : \text{False}}{\text{Case}_{\text{False}} f : C f} \quad (\text{False-El}) $$

**There is no Equality Rule**

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

- False has no elements.
- It is the formula, which is always false, since it has no proofs.
- Often called falsum or absurdity.
Case $\text{False}$ expresses: from an element $f$ of $\text{False}$ we obtain an element of any set (which might depend on $f$).

Considered as a formula, this means: from a proof of $\text{False}$ we obtain a proof of every other formula.

I.e. $\text{False}$ implies everything.

In logic this principle is called “Ex falsum quodlibet” (from the absurdity follows anything).

E.g. A false formula like “$0 = 1$” or “Swansea lies in Germany” implies everything.
Case\textsubscript{False} has no computational meaning, since there is no element it can be applied to.

Applies of course only if we are working in a terminating type theory.

If we had full recursion, we could define $f : \text{False}$ by $f = f$. However that $f$ doesn’t reduce to canonical form.

That’s why it’s important to carry out the termination check in Agda, otherwise one obtains for instance elements of False.
Finite Sets in Agda

- **Finite sets** can be introduced by giving one constructor for each element. E.g.

\[
\text{data Colour} = \text{blue} \mid \text{red} \mid \text{green}
\]

- With this we obtain \text{red :: Colour}
Finite Sets in Agda

- Elimination is done via case distinction.
- In the “Colour” example above for instance, we can define

\[
is\_red\ (c :: \text{Colour}) :: \text{Bool}'
\]

\[
= \text{case } c \text{ of}
\]

\[
\begin{align*}
\text{(red)} & \rightarrow \text{tt} \\
\text{(green)} & \rightarrow \text{ff} \\
\text{(blue)} & \rightarrow \text{ff}
\end{align*}
\]
In the current version of Agda, True and False are reserved keywords for the built-in type Bool. Therefore we have to rewrite them as True', False'.

The definition of True in Agda is straightforward:

\[
\text{data True'} = \text{true}
\]

Case distinction will require to solve the case true:

\[
g \ (x :: \text{True'}) \\
:: \text{Bool'} \\
= \text{case x of} \\
\quad (\text{true}) \rightarrow \{! !\}
\]
True in Agda

- Alternatively, we can define True in Agda as the empty sig:

  \[ \text{True} = \text{sig}\{\} \]

- Then the element \text{true} of \text{True} is defined as follows

  \[ \text{true} = \text{struct}\{\} :: \text{True} \]

- However, since we have no \( \eta \)-rule, we don’t get that for \( a :: \text{True} \) we have \( a = \text{true} \).
False in Agda

In Agda we can define the empty set as a “data”-set with no constructors:

\[
data \text{False}' =
\]

If we want to solve

\[
g \ (x :: \text{False}') :: \text{Bool}' = \{! !\}
\]

we can insert into the goal \(x\) and choose menu-item “agda-case”.
The result is

\[ g \ (x :: \text{False}') \]
\[ :: \text{Bool}' \]
\[ = \text{case } x \text{ of } \{ \} \]

If we make case distinction on \( x \) there is no case to choose from, so we don’t have to define anything.
Example for the Use of False

Assume the type of trees:

\[
\text{data Tree} = \text{oak} \mid \text{pine} \mid \text{spruce}
\]

Below we will show, how to introduce a function

\[
\text{IsConifer :: Tree} \to \text{Set}
\]

s.t.

\[
\text{IsConifer oak} = \text{False'} \\
\text{IsConifer pine} = \text{True'} \\
\text{IsConifer spruce} = \text{True'}
\]
Example for the Use of False

If we want to define a function from trees, which are conifers, into another set, we can do so by requiring an additional argument “IsConifer”:

\[
f \ (t :: \text{Tree})
\]

\[
(p :: \text{IsConifer} \ t)
\]

\[
:: \ A
\]

\[
= \ \text{case} \ t \ \text{of}
\]

\[
oak \ \rightarrow \ \text{case} \ p \ \text{of} \ \{ \}
\]

\[
pine \ \rightarrow \ \cdots
\]

\[
spruce \ \rightarrow \ \cdots
\]
Example for the Use of False

- In order to use $f$ we have to **know** that $t$ is a conifer,
  i.e. we have to provide an argument $p$ which expresses the fact that we know this.

- Note that we **don’t have to invent a result** of $f$ in case $t$ is an oak tree.
Similarly we can introduce a stack of elements of type \( A \), together with a predicate

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

s.t.

\[
\text{NonEmpty} \; \text{nil} = \text{False}'
\]

where \( \text{nil} \) is the empty stack:

\[
\text{NonEmpty} \; (s :: \text{Stack}) :: \text{Set} \equal{} \begin{cases} \text{False}' & \text{if } (\text{nil}) \\ \text{True}' & \text{if } (\text{cons} \; a \; l) \end{cases}
\]
Example 2 for the Use of False

Now we can define

\[
\text{top} \ (s :: \text{Stack})
\]
\[
(p :: \text{NonEmpty} \ s)
\]
\[
:: \ A
\]
\[
= \ \text{case} \ s \ \text{of}
\]
\[
\text{(nil)} \quad \rightarrow \quad \text{case} \ p \ \text{of} \ \{ \ \}
\]
\[
\text{(cons} \ a \ s') \quad \rightarrow \quad a
\]

(See exampleStack.agda).

Again we don’t have to provide a result, in case s is empty.
(c) Atomic Formulae

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

- We have introduced two formulae:
  - **True**, the always true formula.
    - Corresponds to truth value `tt : Bool`.
  - **False**, the always false formula.
    - Corresponds to truth value `ff : Bool`.
Atomic Formulae

A formula expressing equality between two elements of $\text{Fin}_n$ (for fixed $n$) can now be introduced as follows:

Define a function

$$\text{Eq}_n : \text{Fin}_n \to \text{Fin}_n \to \text{Bool}$$

s.t.

$$\text{Eq}_n A_n^i A_n^i = \text{true}$$
$$\text{Eq}_n A_n^i A_n^j = \text{false} \text{ for } i \neq j$$

$\text{Eq}_n$ can be defined easily (for fixed $n$) by case distinction on its two arguments.
Now apply an operation

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

which maps the truth value to the corresponding formula, i.e. define now

\[
\text{Eq}_n : \text{Fin}_n \rightarrow \text{Fin}_n \rightarrow \text{Set} \\
\text{Eq}_n \ x \ y \ = \ \text{atom} (\text{Eq}_n, \text{Bool} \ x \ y)
\]
**Atomic Formulae**

ATOM is defined s.t.

\[
\text{atom } tt = \text{ True} \\
\text{atom } ff = \text{ False}
\]

So we get for Eqn above

\[
\text{Eqn } A_n^i A_n^i = \text{ True} \\
\text{Eqn } A_n^i A_n^j = \text{ False for } i \neq j
\]

So

\[
\text{Eqn } A_n^i A_n^i \text{ is inhabited, has a proof, is true;}
\]

\[
\text{for } i \neq j, \text{ Eqn } A_n^i A_n^j \text{ is not inhabited, has not a proof, is false.}
\]
Atomic Formulae

atom can be defined as follows:

\[
\text{atom} = \text{Case}_{\text{Bool}}^{\text{Type}} (\lambda b : \text{Bool}. \text{Set}) \ True \ False
\]

: \text{Bool} \to \text{Set}
atom in Agda

atom \( b :: \text{Bool}' \)
\[
\begin{array}{ll}
\text{::} & \text{Set} \\
= & \text{case } b \text{ of} \\
(tt) & \rightarrow \text{True}' \\
(ff) & \rightarrow \text{False}'
\end{array}
\]
Decidable Predicates

In general, atom allows us to define **decidable predicates** on sets.

A predicate is **decidable** if it can be decided by a Boolean valued function.

E.g. **equality on natural numbers** is decidable, since we can define a function $\text{Eq}_{\mathbb{N}, \text{Bool}} : \mathbb{N} \to \mathbb{N} \to \text{Bool}$ which decides it.

Equality on **functions** $\mathbb{N} \to \mathbb{N}$ is **undecidable**, since we cannot define such a function – in order to check equality between $f$ and $g$ we need to check equality between $f \, n$ and $g \, n$ for all $n : \mathbb{N}$. 
Decidable Predicates (Cont.)

- Assume we have a **set of states** of a system $A$.
  - E.g. the set of states a railway controller can choose.
- Assume we have a function $f : A \rightarrow \text{Bool}$.
  - E.g. $f a$ means: **state $a$ is safe**.
- Let now $g : A \rightarrow \text{Set}, g a = \text{atom}(f a)$.
  - If $f a$ is **true** (e.g. $a$ is safe), $g a$ is **inhabited**.
  - If $f a$ is **false** (e.g. $a$ is unsafe), $g a$ is **not inhabited**.
- Now, the existence of a $h : (a : A) \rightarrow g a$ means:
  - For all $a : A$ we have $g a$ is **inhabited**,
  - i.e. for all $a : A$, $f a$ is **true**,
  - e.g. for all $a : A$, $a$ is **safe**.

Jump over Traffic Light Example.
The Traffic Light Example

Assume a road crossing, controlled by traffic lights:

- A
- A'
- B
- B'
- A
- B
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:

\[
data \text{ Colour} = \text{red} | \text{green}\]

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 \( B' \))

\[
\text{Phys\_State} :: \text{Set} \\
= \text{sig} \\
sigA :: \text{Colour} \\
sigB :: \text{Colour}
\]
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.
- Agreen – signal A (and A’) is green, signal B is red.
- Bgreen – signal B is green, signal A is red.
We therefore define

\[
data \text{ Control}_\text{State} = \text{Allred} \mid \text{Agreen} \mid \text{Bgreen}
\]
Control States to Physical States

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

\[
\text{toSigA } (s :: \text{Control\_State}) \\
:: \text{Colour} \\
= \text{case } s \text{ of} \\
\quad (\text{Allred}) \rightarrow \text{red} \\
\quad (\text{Agreen}) \rightarrow \text{green} \\
\quad (\text{Bgreen}) \rightarrow \text{red}
\]
Control States to Physical States

toSigB \ (s :: \text{Control\_State})
\quad :: \text{Colour}
\quad = \text{case } s \text{ of}
\quad \quad (\text{Allred}) \rightarrow \text{red}
\quad \quad (\text{Agreen}) \rightarrow \text{red}
\quad \quad (\text{Bgreen}) \rightarrow \text{green}
Control States to Physical States

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

\[
\text{phys\_state} \ (s :: \text{Control\_State}) \\
:: \text{Phys\_State} \\
= \text{struct} \\
\quad \text{sigA} = \text{toSigA} \ s \\
\quad \text{sigB} = \text{toSigB} \ s
\]
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:

\[
\text{CorAux} \quad (a, b :: \text{Colour})
\]
\[
:: \text{Set}
\]
\[
= \text{case } a \text{ of}
\]
\[
\text{(red)} \rightarrow \text{True}'
\]
\[
\text{(green)} \rightarrow \text{case } b \text{ of}
\]
\[
\text{(red)} \rightarrow \text{True}'
\]
\[
\text{(green)} \rightarrow \text{False}'
\]
Now we define

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

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

In the current example this wouldn’t have caused problems, but in more complex examples it does (due to the lack of the $\eta$-rule).
Safety of the System

Now we show that all control states are safe:

\[
\text{cor\_proof} \ (s :: \text{Control\_State}) \\
:: \ Cor(\text{phys\_state} \ s) \\
= \ \text{case } s \ \text{of} \\
\quad (\text{Allred}) \to \ true \\
\quad (\text{Agreen}) \to \ true \\
\quad (\text{Bgreen}) \to \ true
\]
Safety of the System (Cont.)

The first element \texttt{true} was an element of \texttt{Cor(phys\_state Allred)}, which reduces to \texttt{True}.

Similarly for the other two elements.

This works only because \texttt{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 \texttt{False}, which we can’t solve.
Safety of the System (Cont.)

If one makes a **mistake** which results in an unsafe situation

- e.g. sets $\text{toSigB \ Agreen} = \text{green}$,

then in the last step we obtain one goal of type $\text{False}'$.

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 $\text{cor\_proof \ Agreen}$, but this would be rejected by the termination check.)
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

\[ \text{inl} : A \to A + B \text{ and } \text{inr}. \]

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 $A + B$. 
Disjoint Union

Informally, if

\[ A = \{1, 2\} \]

and

\[ B = \{3, 4, 5\} \]

then

\[ A + B = \{\text{inl}(1), \text{inl}(2), \text{inr}(3), \text{inr}(4), \text{inr}(5)\} \]

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 \( b : B \).

Jump over Comparision with Product
Comparison with the Product

Note that if we have again

\[ A = \{1, 2\} \]

and

\[ B = \{3, 4, 5\}, \]
	hen then for the product we have informally

\[ A \times B = \{p(1, 3), p(1, 4), p(1, 5), p(2, 3), p(2, 4), p(2, 5)\} \]

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

Formation Rule

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

Introduction Rules

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

$$\frac{A : \text{Set} \quad B : \text{Set} \quad b : B}{\text{inr} \ A \ B \ b : A + B} \quad (+-\text{I}_{\text{inr}})$$
Rules for $A + B$

Elimination Rules

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

\[ (+-\text{El}) \]

\((cl, cr \text{ stand for “case left”, “case right”})\).
Rules for $A + B$

Equality Rules

Case$_+\ A\ B\ C\ cl\ cr\ (\text{inl } A\ B\ a)\ =\ cl\ a : C\ (\text{inl } A\ B\ a)\ (\text{+-Eq}_{\text{inl}})$

Case$_+\ A\ B\ C\ cl\ cr\ (\text{inr } A\ B\ b)\ =\ cr\ b : C\ (\text{inr } A\ B\ b)\ (\text{+-Eq}_{\text{inr}})$

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 equality rules is:

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

CS_336/CS_M36 (part 2) Interactive Theorem Proving; Lentterm 2005, Sec. 4(d)
Disjoint Union in Agda

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

\[
(+)
\begin{align*}
(A :: \text{Set}) \\
(B :: \text{Set}) \\
:: \text{Set} \\
= \text{data inl}(a :: A) \mid \text{inr}(b :: B)
\end{align*}
\]
The notation \((+\rangle\) means, that \(+\) can be used **infix**.

Now we have, if \(A, B :: \text{Set}\):

- \(\text{inl}@(A + B) :: A \rightarrow (A + B)\)
- \(\text{inr}@(A + B) :: B \rightarrow (A + B)\)

This can be checked using the menu “**infer type**” in a dummy goal.

Note that we cannot assign a type to \(\text{inl}@_\) or \(\text{inr}@_\).

\(\langle+\rangle\) **cannot** be defined using the abbreviated **data** notation
(which would be of the form
**data** \((+\rangle = \cdots\)).
Elimination is again represented by case distinction. So if want to define for $A, B :: \text{Set}$ for instance

$$f \ (c :: A + B) :: \text{Bool'}$$

$$= \ \{! \ !\}$$

we can type into the goal $c$ and choose menu “agda-case”.
We obtain

\[ f \ (c :: A + B) :: \text{Bool}' \]

\[ = \ \text{case } c \ \text{of} \]

\[ (\text{inl } a) \to \{! \! \! \} \]

\[ (\text{inr } b) \to \{! \! \! \} \]

and insert into the first goal e.g. true and the second one false
Use of Concrete Disjoint Sets

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

  \[
  \text{data Plant} = \text{tree}(t :: \text{Tree}) \mid \text{flower}(f :: \text{Flower})
  \]

- Now one can define for instance

  \[
  \text{isFlower} \quad (p :: \text{Plant}) \\
  :: \quad \text{Bool}' \\
  = \quad \text{case } p \text{ of} \\
  \quad (\text{tree } t) \quad \rightarrow \quad \text{ff} \\
  \quad (\text{flower } f) \quad \rightarrow \quad \text{tt}
  \]
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 write $\lor$ for it (on slides we write $\lor$) and define it as follows:

\[(\lor) \quad (A, B :: \text{Set}) \quad :: \quad \text{Set} \quad = \quad \text{data or1}(a :: A) \mid \text{or2}(b :: B)\]

- See `exampleproofproplogic7.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 \((A \lor B) \rightarrow (B \lor A)\):

```
lemma3  (ab :: A \lor B)
  ::  B \lor A
  =  case ab of
    (or1 a)  \rightarrow  or2@_ a
    (or2 b)  \rightarrow  or1@_ b
```

See exampleproofproplgic9.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):

\[
\text{OR3} \ (A, B, C :: \text{Set}) \\
:: \ \text{Set} \\
= \ \text{data} \ or1 \ (a :: A) \mid or2 \ (b :: B) \mid or3 \ (c :: C)
\]

See exampleproofpropllogic8.agda.

Jump over \(\Sigma\)-Type.
(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 $B : A \to \text{Set}$.

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.
**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**

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

**Elimination Rule**

\[
\begin{align*}
A & : \text{Set} \\
B & : A \rightarrow \text{Set} \\
C & : (\Sigma A B) \rightarrow \text{Set} \\
c & : (a : A) \rightarrow (b : B a) \rightarrow C (p A B a b)
\end{align*}
\]

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

**Equality Rule**

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

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:

- $\Sigma : (A : \text{Set})$
  - $\rightarrow (A \rightarrow \text{Set})$
  - $\rightarrow \text{Set}$.

- $p : (A : \text{Set})$
  - $\rightarrow (B : A \rightarrow \text{Set})$
  - $\rightarrow (a : A)$
    - $\rightarrow (B\ a)$
    - $\rightarrow \Sigma A B$.
The $\Sigma$-Set using the Log. Framew.

Case$_\Sigma$:

$(A : \text{Set})$
$
\rightarrow (B : A \rightarrow \text{Set})
$
$
\rightarrow (C : (\Sigma A B) \rightarrow \text{Set})
$
$
\rightarrow ((a : A, b : B a) \rightarrow C (\mathit{p} A B a b))
$
$
\rightarrow (d : \Sigma A B)
$
$
\rightarrow C \ d.$
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 $\text{Case}_\Sigma$:

\[
\begin{align*}
\pi_0 &= \text{Case}_\Sigma A B \left( \lambda x^{\Sigma A B}.A \right) \left( \lambda x^A.\lambda y^{B x}.x \right) \\
\pi_1 &= \text{Case}_\Sigma A B \left( \lambda x^{\Sigma A B}.B \pi_0(x) \right) \left( \lambda x^A.\lambda y^{B x}.y \right)
\end{align*}
\]

On the other hand, from $\pi_0, \pi_1$ we can define $\text{Case}_\Sigma$ as follows:

\[
\begin{align*}
\lambda A^{\text{Set}}. \lambda B^{A \rightarrow \text{Set}}. \lambda C^{(\Sigma A B) \rightarrow \text{Set}}. \\
\lambda s^{(a:A) \rightarrow (b:B a) \rightarrow C (p a b)}. \lambda d^{(\Sigma A B)} . s \pi_0(d) \pi_1(d)
\end{align*}
\]
The $\Sigma$-Set and the Dep. Prod.

- 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.

\[
\text{p} : \Sigma (A :: \text{Set}) \quad (B :: A \rightarrow \text{Set}) \quad :: \text{Set} \\
= \text{data} \ p \ (a :: A) \ (b :: Ba)
\]

Elimination uses \text{case-distinction}:

\[
f \ (c :: \Sigma A B) \quad :: \ D \\
= \text{case } c \text{ of} \\
\ (p \ a \ b) \quad \rightarrow \quad \cdots
\]
Again one usually defines concrete $\Sigma$-sets more directly.

**Example:** Assume we have defined

- a set $\text{Plant\_Group}$ for **groups of plants** (e.g. “tree”, “flower”),
- depending on $g :: \text{Plant\_Group}$, sets $\text{Plants\_in\_group \ g}$ for **plants in that group**.

The **set of plants** can then be defined as

```agda
data Plant = plant (g :: Plant\_Group)(pg :: Plants\_in\_group g)
```
Not surprisingly, for **elimination** we use **case distinction**, e.g.:

\[
f \ (p :: \text{Plant}) \\
\quad :: \quad \text{Plant\_group} \\
\quad = \quad \text{case } p \ \text{of} \\
\quad \quad (\text{plant } g \ pg) \ \rightarrow \ g
\]
We have already seen how to represent the propositional connectives and decidable atomic formulae in Agda and therefore as well in dependent type theory:

- Implication

\[ A \rightarrow B \]

is represented as the nondependent function type

\[ A \rightarrow B \]

- Conjunction

\[ A \land B \]

is represented as one of the two versions of the product of \( A \) and \( B \).
Disjunction

\[ A \lor B \]

is represented as the disjoint union of \( A \) and \( B \).

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

\[ \text{atom} (f \ a_1 \ \cdots \ a_n) \]
Negation

- We haven’t introduced $\neg A$ in dependent type theory.
- $\neg A$ is true iff $A$ is false iff there is no proof of $A$.
- Now we can show that there is no proof of $A$ iff $A \rightarrow \text{False}$ is true:
  - If there is no proof of $A$, then from every proof of $A$ we can obtain a proof of False (since there is no proof of $A$); therefore $A \rightarrow \text{False}$ is true.
  - On the other hand, if we $A \rightarrow \text{False}$ is true, i.e. has a proof, then there cannot be any proof of $A$, because from it we could get a proof of False, which is the empty set.
- Therefore $\neg A$ is true iff $A \rightarrow \text{False}$ is true.
- Therefore we can identify $\neg A$ with $A \rightarrow \text{False}$.
In this subsection we will investigate, how to represent universal and existential quantification in dependent type theory.

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

We write therefore

\[ \forall x : A . B \text{ for} \]

“for all \( x \) of type \( A \), \( B \) holds”

(where \( B \) usually depends on \( x \));

\[ \exists x : A . B \text{ for} \]

“there exists an \( x \) of type \( A \), s.t. \( B \) holds”

(again \( B \) usually depends on \( x \)).
Universal Quantification

\[ \forall x : A. B \text{ is true iff, for all } x : A \text{ there exists a proof of } B \text{ (with that } x) . \]

Therefore a proof of \( \forall x : A. B \) is a function, which takes an \( x : A \) and computes an element of \( B \).

Therefore the set of proofs of \( \forall x : A. B \) is the set of functions, mapping an element \( x : A \) to an element of \( B \).

This set is just the dependent function type \( (x : A) \rightarrow B \).

Therefore we can identify \( \forall x : A. B \) with \( (x : A) \rightarrow B \).
\textbf{∀ in Agda}

- \( \forall x : A. B \) is represented by \((x : A) \rightarrow B\) in Agda.
- As an example,
  - we define a `<`-operation on Bool using \( \mathsf{ff} < \mathsf{tt} \) is true and \( b < b' \) is false, otherwise.
  - Then we show \( \forall x : \text{Bool}' \cdot \neg(x < x) \).
- See \texttt{exampleproofproplogic10b.agda}.
Example (\(\forall, \text{Cont.}\))

First we define a Boolean valued less-than relation on \(\text{Bool}'\) as follows:

\[
\text{LessBool} \quad (a, b :: \text{Bool}')
\]
\[
:: \quad \text{Set}
\]
\[
= \quad \text{case } a \text{ of}
\]
\[
\begin{align*}
(tt) & \rightarrow \text{ff} \\
(ff) & \rightarrow b
\end{align*}
\]

Explanation of this definition:
- if \(a\) is true, then \(a\) is never less than \(b\).
- if \(a\) is false, then \(a\) is less than \(b\) iff \(b\) is true, so the truth value of \(\text{LessBool} a b\) is the same as \(b\).
Example ($\forall$, Cont.)

Then we define ($<$) as follows

$$(<) \ (a, b :: \text{Bool'})$$

:: Set

= atom (LessBool $a \ b$)
Example $(\forall, \text{Cont.})$

- $(<)$ can be used infix, i.e. we can write $a < b$ for $(<) a b$.

- We introduce $\text{Not } A$:

\[
\text{Not } (A :: \text{Set}) :: \text{Set} = A \rightarrow \text{False}
\]

- The statement that $(<)$ is antireflexive is

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

which is represented in Agda as follows:

\[
\text{Lemma4} :: \text{Set} = (a :: \text{Bool'}) \rightarrow \text{Not } (a < a)
\]
Example ($\forall$, Cont.)

Lemma4 :: Set

\[ = (a :: \text{Bool}') \rightarrow \text{Not} (a < a) \]

Mark: Since $\text{Not} (a < a) = (a < a) \rightarrow \text{False}'$, we have

Lemma4 = (a :: \text{Bool}') \rightarrow \text{Not} (a < a)

\[ = (a :: \text{Bool}') \rightarrow (a < a) \rightarrow \text{False}' \]
Lemma4 = $(a :: \text{Bool'}) \rightarrow (a < a) \rightarrow \text{False'}$

- We want to prove Lemma4.
  - A proof of Lemma4 will be an element $\text{lemma4} :: \text{Lemma4}$.

- So we have to solve the following goal:

  \[
  \text{lemma4} :: \text{Lemma4} = \{! !\}
  \]

- The type of the goal is

  \[
  \text{Lemma4} = (a :: \text{Bool'}) \rightarrow (a < a) \rightarrow \text{False'}
  \]
Example ($\forall$, Cont.)

\[
\text{lemma4 :: Lemma4} = \{! !\}
\]

Type of goal is Lemma4 = \((a :: \text{Bool'}) \rightarrow (a < a) \rightarrow \text{False'}\).

- An element of \((a :: \text{Bool'}) \rightarrow (a < a) \rightarrow \text{False'}\) can be introduced by \(\lambda\)-abstracting \(\lambda(a :: \text{Bool'})\) and \(\lambda(aa :: (a < a)):\)

\[
\text{lemma4 :: Lemma4} = \lambda(a :: \text{Bool'}) \rightarrow \\
\lambda(aa :: a < a) \rightarrow \\
\{! !\}
\]

- The type of goal is now the conclusion of \((a :: \text{Bool'}) \rightarrow (a < a) \rightarrow \text{False'}, namely \text{False'}\).
Example (∀, Cont.)

\[
\text{lemma4} :: \text{Lemma4} \\
= \lambda(a :: \text{Bool'}) \rightarrow \\
\lambda(aa :: a < a) \rightarrow \\
\{! !\}
\]

Type of goal is \text{False}'.

- We need to make use of our assumptions, namely \\
  \(a :: \text{Bool'}\) and \(aa :: a < a\).

- \(a < b\) is defined by case disjunction on \(a\) and \(b\).
  - Unless we know that \(a = \text{tt}\) or \(a = \text{ff}\), we don’t know much about \(a < a\).
  - So it seems to be a good step to make case distinction on \(a\).
Example (\(\forall\), Cont.)

\[
\text{lemma4} :: \text{Lemma4} \\
= \lambda(a :: \text{Bool'}) \rightarrow \\
\lambda(aa :: a < a) \rightarrow \\
\text{case } a \text{ of} \\
(tt) \rightarrow \{! !\} \\
(ff) \rightarrow \{! !\}
\]

The type of both goals is the same as before, namely \(\text{False'}\), since it didn’t depend on \(a\).
Example (∀, Cont.)

\[
\text{lemma4} :: \text{Lemma4} \\
= \lambda(a :: \text{Bool}') \to \\
\lambda(aa :: a < a) \to \\
\text{case } a \text{ of} \\
(tt) \to \{! !\} \\
(ff) \to \{! !\}
\]

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

- In case of \(a = tt\), we have \(a < a = (tt < tt) = \text{False}'\)
- In case of \(a = ff\), we have \(a < a = (ff < ff) = \text{False}'\)
Example ($\forall$, Cont.)

\[
\text{lemma4} :: \text{Lemma4} \\
= \lambda(a :: \text{Bool'}) \to \\
\lambda(aa :: a < a) \to \\
\text{case } a \text{ of} \\
\quad (\text{tt}) \to \{! !\} \\
\quad (\text{ff}) \to \{! !\}
\]

Since in both goals we have $aa :: (a < a) = \text{False'}$, we can make case distinction on $aa$, which is the empty case distinction.
We finish our proof as follows:

\[
\text{lemma4} :: \text{Lemma4} \\
= \lambda(a :: \text{Bool}^l) \rightarrow \\
\lambda(aa :: a < a) \rightarrow \\
\text{case } a \text{ of} \\
\quad (\texttt{tt}) \rightarrow \text{case } aa \text{ of } \{ \} \\
\quad (\texttt{ff}) \rightarrow \text{case } aa \text{ of } \{ \}
\]
Example ($\forall$, Cont.)

In the previous example,

- the type of goal was $\text{False}'$,
- and $aa : \text{False}'$.

So, instead of using the empty case distinction directly, we could have as well inserted $aa$ in those goals:

\[
\begin{align*}
\text{lemma4a} & \quad : \quad \text{Lemma4} \\
& = \lambda(a :: \text{Bool}') \rightarrow \\
& \quad \lambda(aa :: a < a) \rightarrow \\
& \quad \text{case } a \text{ of} \\
& \quad \quad (\text{tt}) \rightarrow aa \\
& \quad \quad (\text{ff}) \rightarrow aa
\end{align*}
\]
Existential Quantification

- $\exists x : A. B$ is true iff there exists an $a : A$ such that $B[x := a]$ is true.
- Therefore a proof of $\exists x : A. B$ is a pair $\langle a, p \rangle$ consisting of an element $a : A$ and a proof $p$ of $B[x := a]$.
- Therefore the set of proofs of $\exists x : A. B$ is the dependent product $(x : A) \times B$.
- We can identify $\exists x : A. B$ with $(x : A) \times B$. 
\( \exists x : A . B \) is represented therefore in Agda by one of the two dependent products in Agda.

Using meaningful names, we can define \( \exists x : A . B \) as follows:

Version 1 :: Set
= sig
    a :: A
    b :: B[x := a]

Version 2 :: Set
= data exists (a :: A)(b :: B[x := a])
\exists \text{ in Agda}

Above $B[x := a]$ is the result of substituting in $B$ for $x$ the variable $a$. 
Example (∃)

As an example,
- we define negation \( \neg \) on \( \text{Bool} \),
- define an equality \( (==) \) on \( \text{Bool} \),
- and show \( \forall a : \text{Bool}' . \exists b : \text{Bool}' . a == \neg b \).

See exampleproofpropllogic11.agda.
Example ( */, Cont.*)

\( \neg \) is defined as follows:

\[
\text{neg } (a :: \text{Bool'}) \\
:: \text{Bool'} \\
= \text{case } a \text{ of} \\
(\text{tt}) \rightarrow \text{ff} \\
(\text{ff'}) \rightarrow \text{tt}
\]
Example (∃)

A Boolean valued equality on $\text{Bool}'$ is defined as follows:

$$\text{EqBool} \ (a, b :: \text{Bool}') :: \text{Bool}' = \text{case } a \text{ of}$$

$$\begin{align*}
(tt) & \rightarrow b \\
(ff) & \rightarrow \neg b
\end{align*}$$

Then we define

$$\text{==(a, b :: \text{Bool}') :: \text{Bool}' = \text{atom (EqBool a b)}}$$
Example (лежащее, Cont.)

(==) can be written infix, i.e. we can write $a == b$ for $a b$. 
Example (∃, Cont.)

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

\[
\text{Lemma5aux } (a :: \text{Bool}') \\
:: \text{Set} \\
= \text{sig} \\
\quad b :: \text{Bool}' \\
\quad ab :: a == \neg b
\]

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

\[
\text{Lemma5 } :: \text{Set} \\
= (a :: \text{Bool}') \rightarrow \text{Lemma5aux } a
\]
Example ($\exists$, Cont.)

- A proof of Lemma5 is an element
  
  \[
  \text{lemma5 :: Lemma5}
  \]

  and we get the goal

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

- The type of goal is

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

- Any goal of function type is usually best solved by using $\lambda$-abstracting.
Example (∃, Cont.)

Lemma5 :: Set
= \(a :: \text{Bool}'\) \rightarrow \text{Lemma5aux } a

- We get

\[
\text{lemma5} :: \text{Lemma5} \\
= \lambda (a :: \text{Bool}') \\
\rightarrow \{! !\}
\]

- The type of the goal is

\[
\text{Lemma5aux } a = \text{sig} \\
b :: \text{Bool}' \\
ab :: a == \text{neg } b
\]
Example (∃, Cont.)

\[
\text{lemma5} :: \text{Lemma5} = \lambda (a :: \text{Bool'}) \\
\to \{! !\}
\]

Type of goal is

\[
sig \\
\quad b :: \text{Bool'} \\
\quad ab :: a == \text{neg } b
\]

- We cannot show this goal universally for all \( a \) directly.
- We have to provide a different \( b \) depending on whether \( a = tt \) or \( a = ff \).
- So we need to make case distinction on \( a \).
Example (∃, Cont.)

We get

\[ \text{lemma5} :: \quad \text{Lemma5} \]
\[ = \quad \lambda (a :: \text{Bool}') \]
\[ \rightarrow \text{case } a \text{ of} \]
\[ (\text{tt}) \rightarrow \{! \!\} \]
\[ (\text{ff}) \rightarrow \{! \!\} \]
Example (∃, Cont.)

\[
\text{lemma5 :: Lemma5} = \lambda(a :: \text{Bool'}) \rightarrow \text{case } a \text{ of } \{(\text{tt}) \rightarrow \{! !\}, (\text{ff}) \rightarrow \{! !\}\}
\]

- In case of \(a = \text{tt}\), the type of goal is

\[
\text{Lemma5aux tt} = \text{sig} \\
\quad b :: \text{Bool'} \\
\quad ab :: \text{tt} == \text{neg } b
\]

- So we can use goal menu \text{intro} and obtain:
Example (∃, Cont.)

\[
\text{lemma5} :: \text{Lemma5} = \lambda(a :: \text{Bool}') \to \text{case } a \text{ of}
\]

\[
\begin{align*}
\text{(tt)} & \to \text{struct} \\
& \quad b = \{! !\} \\
& \quad ab = \{! !\}
\end{align*}
\]

The first goal can be solved by setting \( b := \text{ff} \).

Then the type of the second goal is

\[
\begin{align*}
\text{(tt == neg } b) &= \text{(tt == neg } \text{ff)} \\
&= \text{(tt == tt)} \\
&= \text{True}'
\end{align*}
\]

which can be solved by setting \( ab := \text{true} \).
So we get:

\[
\text{lemma5 :: Lemma5} = \lambda(a :: \text{Bool'}) \rightarrow \text{case } a \text{ of}
\]

\[
\begin{align*}
\text{(tt)} & \rightarrow \text{struct} \\
& \quad b = \text{ff} \\
& \quad ab = \text{true} \\
\text{(ff)} & \rightarrow \{! !\}
\end{align*}
\]

In case \(a = \text{ff}\), we use again intro and obtain:
Example (♯, Cont.)

\[
\text{lemma5 :: Lemma5} = \lambda (a :: \text{Bool}') \rightarrow \text{case } a \text{ of}
\]

\[
\begin{align*}
\text{tt} & \rightarrow \text{struct} \\
\quad b &= \text{ff} \\
\quad ab &= \text{true}
\end{align*}
\]

\[
\begin{align*}
\text{ff} & \rightarrow \text{struct} \\
\quad b &= \{! \! \} \\
\quad ab &= \{! \! \}
\end{align*}
\]

The case \( a = \text{ff} \) can be solved in a similar way by setting \( b = \text{tt}, \ ab = \text{true} \).
Example (‡, Cont.)

The resulting proof is as follows:

\[
\text{lemma5} :: \text{Lemma5} = \lambda(a :: \text{Bool}') \rightarrow \text{case } a \text{ of}
\]

\[
\begin{align*}
\text{(tt) } & \rightarrow \text{ struct} \\
& \quad b = \text{ff} \\
& \quad ab = \text{true}
\end{align*}
\]

\[
\begin{align*}
\text{(ff) } & \rightarrow \text{ struct} \\
& \quad b = \text{tt} \\
& \quad ab = \text{true}
\end{align*}
\]
Complex Example

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

  \begin{align*}
  & \text{postulate } A :: \text{Set} \\
  & \text{postulate } \text{EqA} :: A \rightarrow A \rightarrow \text{Set} \\
  & \text{postulate } B :: \text{Set} \\
  & \text{postulate } \text{EqB} :: B \rightarrow B \rightarrow \text{Set}
  \end{align*}

- We will introduce
  - the disjoint union $AB$ of $A$ and $B$
  - an equality $\text{EqAB}$ on $AB$
  - and show that if $\text{EqA}$ and $\text{EqB}$ are symmetric, so is $\text{EqAB}$.

- See exampleDisjointUnionEqual.agda.
Equality Sets

EqA (and EqB) could be decidable equalities,
i.e. $\text{EqA} = \lambda(a, b :: A) \to \text{atom} \ (\text{eqboolA} \ a \ b)$,
where $\text{eqboolA} :: A \to A \to \text{Bool}'$,

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

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

which reads in Agda as follows:

$$\text{EqN\_N} \ (f, g :: \mathbb{N} \to \mathbb{N})$$
$$:: \ \text{Set}$$
$$= \ (n :: \mathbb{N}) \to f \ n == g \ n$$

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

The last equality is undecidable, since in order to check whether \( \text{Eq} \mathbb{N} \_ \mathbb{N} \ f \ g \) holds we have to check for all \( n : \mathbb{N} \) whether \( f \ n = g \ n \) holds.
The formation of $A + B$ is straightforward:

\[
(+) \ (A, B :: Set) :: Set
\]

\[
= \text{data inl}(a :: A) \mid \text{inr}(b :: B)
\]
We define the equality $\text{Eq}_{AB}$ on $A + B$ as follows:

- Assume $ab, ab' : A + B$.

- If one is of the form $\text{inl} \ a$ and the other of the form $\text{inr} \ b$, then $\text{Eq}_{AB} \ ab \ ab'$ should be false, so we define

  $$\text{Eq}_{AB} (\text{inl} \ a) \ (\text{inr} \ b) = \text{Eq}_{AB} (\text{inr} \ a) \ (\text{inl} \ a) = \text{False}' .$$

- If they are of the form $\text{inl} \ a$ and $\text{inl} \ a'$, respectively, then $\text{Eq}_{AB} \ ab \ ab'$ should be true if $\text{Eq}_A \ a \ a'$ holds.

  This can be achieved by defining

  $$\text{Eq}_{AB} (\text{inl} \ a) \ (\text{inl} \ a') = \text{Eq}_A \ a \ a' .$$
Complex Example (Cont.)

- If they are of the form \( \text{inr } b \) and \( \text{inr } b' \), respectively, then \( \text{EqAB } ab \ ab' \) should be true if \( \text{EqB } b \ b' \) holds.
- This can be achieved by defining in this case

\[
\text{EqAB } \text{(inr } b \text{) (inr } b' \text{)} = \text{EqB } b \ b'.
\]

- The above equations will be definitional equalities, i.e. for instance \( \text{EqAB } \text{(inl } a \text{) (inl } a' \text{)} \) will rewrite to \( \text{EqA } a \ a' \).
Complex Example (Cont.)

The definition of $\text{EqAB}$ is as follows:

\[
\text{EqAB} :: (A + B) \to (A + B) \to \text{Set} = \lambda(ab, ab' :: A + B) \to \\
\text{case } ab \text{ of} \\
\text{(inl } a\text{)} \to \text{ case } ab' \text{ of} \\
\quad \text{(inl } a'\text{)} \to \text{ EqA } a \ a' \\
\quad \text{(inr } b'\text{)} \to \text{ False'} \\
\text{(inr } b\text{)} \to \text{ case } ab' \text{ of} \\
\quad \text{(inl } a'\text{)} \to \text{ False'} \\
\quad \text{(inr } b'\text{)} \to \text{ EqB } b \ b'
\]
The following code is equivalent to the previous code:

\[
\text{EqAB } (ab, ab' :: A + B)
\]

\[
:: \text{ Set}
\]

\[
= \text{ case } ab \text{ of}
\]

\[
\quad (\text{inl } a) \rightarrow \text{ case } ab' \text{ of}
\]

\[
\quad \quad (\text{inl } a') \rightarrow \text{ EqA } a \ a'
\]

\[
\quad (\text{inr } b') \rightarrow \text{ False}'
\]

\[
\quad (\text{inr } b) \rightarrow \text{ case } ab' \text{ of}
\]

\[
\quad \quad (\text{inl } a') \rightarrow \text{ False}'
\]

\[
\quad (\text{inr } b') \rightarrow \text{ EqB } b \ b'
\]
We introduce the formulae expressing that $\text{EqA}$, $\text{EqB}$, $\text{EqAB}$ are symmetric:

Symmetry of $\text{EqA}$ is the formula

$$\forall a, a' : A. \text{EqA} a a' \rightarrow \text{EqA} a' a$$

which translates as follows:

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

$$= (a, a' :: A) \rightarrow \text{EqA} a a' \rightarrow \text{EqA} a' a$$
The others are similar:

\[
\text{SymB} :: \text{Set} \\
= (b, b' :: B) \rightarrow \text{EqB} \ b \ b' \rightarrow \text{EqB} \ b' \ b
\]

\[
\text{SymAB} :: \text{Set} \\
= (ab, ab' :: A + B) \rightarrow \text{EqAB} \ ab \ ab' \rightarrow \text{EqAB} \ ab' \ ab
\]
Note that \( \text{SymA} \) is the **statement** expressing that \( \text{EqA} \) is symmetric.

- It is not a proof that \( \text{EqA} \) is symmetric.
- We can define \( \text{SymA} \) independently of whether \( \text{EqA} \) is symmetric or not.
- A proof that \( \text{EqA} \) is symmetric is **an element of** \( \text{SymA} \), i.e a term \( \text{symA} \) s.t.

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

- Note that we don’t have to show that \( \text{SymA} \) holds.
- We have to show that if \( \text{SymA} \) and \( \text{SymB} \) hold, then \( \text{SymAB} \) holds.
Complex Example

What we want to show is that $\text{SymA}$ and $\text{SymB}$ implies $\text{SymAB}$.

So we need to solve

\[
\text{symAB} :: \text{SymA} \to \text{SymB} \to \text{SymAB} = \{! !\}
\]

As pointed out before, it is equivalent and more convenient to define $\text{symAB}$ as follows:

\[
\text{symAB} \quad (\text{symA} :: \text{SymA})
\quad (\text{symB} :: \text{SymB})
:: \text{SymAB} = \{! !\}
\]
The type of the goal is `SymAB` which is
\((ab, ab' :: A + B) \rightarrow \text{EqAB } ab \ ab' \rightarrow \text{EqAB } ab' \ ab.\)

An element of this type can be introduced by a \(\lambda\)-term, and using \texttt{agda-goal-menu} “intro” results in the code on the next slide.
Complex Example

\[ \text{symAB} \quad (\text{symA} :: \text{SymA}) \]
\[ (\text{symB} :: \text{SymB}) \]
\[ :: \text{SymAB} \]
\[ = \lambda(ab, ab' :: A + B) \rightarrow \]
\[ \lambda(abab' :: \text{EqAB} \ ab \ ab') \rightarrow \]
\[ \{! !\} \]

- The type of the goal is now \( \text{EqAB} \ ab' \ ab \).
- We make case distinction on \( ab \) and \( ab' \) and obtain the following:
Complex Example

\[ \text{symAB} \quad (\text{symA :: SymA}) \]
\[ \text{(symB :: SymB)} \]
\[ \quad :: \quad \text{SymAB} \]
\[ = \quad \lambda (ab, ab' :: A + B) \rightarrow \]
\[ \quad \lambda (abab' :: \text{EqAB} \ ab \ ab') \rightarrow \]
\[ \text{case } ab \ of \]
\[ \quad (\text{inl} \ a) \quad \rightarrow \quad \text{case } ab' \ of \]
\[ \quad \quad \quad (\text{inl} \ a') \quad \rightarrow \quad \{! !\} \]
\[ \quad \quad \quad (\text{inr} \ b') \quad \rightarrow \quad \{! !\} \]
\[ \quad (\text{inr} \ b) \quad \rightarrow \quad \text{case } ab' \ of \]
\[ \quad \quad \quad (\text{inl} \ a') \quad \rightarrow \quad \{! !\} \]
\[ \quad \quad \quad (\text{inr} \ b') \quad \rightarrow \quad \{! !\} \]
In case \( ab = \text{inl} \ a \) and \( ab' = \text{inl} \ a' \) we
have to show

\[
\text{Eq}_{AB} \ ab' \ ab \ , \quad \text{which is equal to} \quad \text{Eq}_{A} \ a' \ a
\]

and have as assumption

\[
\text{abab}' :: \text{Eq}_{AB} \ ab \ ab' \ , \quad \text{which is equal to} \quad \text{Eq}_{A} \ a \ a'.
\]

So we have to derive from \( abab' : \text{Eq}_{A} \ a \ a' \) an element of \( \text{Eq}_{A} \ a' \ a \).

We have

\[
symA : (a, a' :: A) \rightarrow \text{Eq}_{A} \ a \ a' \rightarrow \text{Eq}_{A} \ a' \ a .
\]

Therefore we can apply \( symA \) to \( a, a' \) and \( abab' \).
Complex Example (Cont.)

(Case $ab = \text{inl} \, a$, $ab' = \text{inl} \, a'$)

We obtain

$$\text{symA} \ a \ a' \ abab' : \text{EqA} \ a' \ a$$

Since $\text{EqA} \ a' \ a = \text{EqAB} \ ab' \ ab$ we get as well

$$\text{symA} \ a \ a' \ abab' : \text{EqAB} \ ab' \ ab$$

and can use this to solve our first goal.
Complex Example

symAB  \((symA :: SymA)\)

\((symB :: SymB)\)

::  SymAB

\[= \lambda(ab, ab' :: A + B) \rightarrow \lambda(abab' :: EqAB ab ab') \rightarrow\]

case \(ab\) of

\((\text{inl } a) \quad \rightarrow \quad \text{case } ab' \text{ of}\)

\((\text{inl } a') \quad \rightarrow \quad \text{symA } a a' \ abab'\)

\((\text{inr } b') \quad \rightarrow \quad \{! \} \)

\((\text{inr } b) \quad \rightarrow \quad \text{case } ab' \text{ of}\)

\((\text{inl } a') \quad \rightarrow \quad \{! \}\)

\((\text{inr } b') \quad \rightarrow \quad \{! \}\)
In case \( ab = \text{inr} \ b \) and \( ab' = \text{inr} \ b' \) we can similarly use

\[
symB \ b \ b' \ abab'
\]

in order to solve our goal.

In case \( ab = \text{inl} \ a \), and \( ab' = \text{inr} \ b \)

we have \( \text{EqAB} \ ab \ ab' = \text{False}' \),

therefore

\[
abab' : \text{False}' ,
\]

therefore empty case distinction on \( abab' \) solves the goal.
Similarly, in case $ab = \text{inl } a$, and $ab' = \text{inr } b$ we have that

$$abab' : \text{False}$$

and again empty case distinction on $abab'$ solves the goal.

The complete solution is on the next slide.
Complex Example

\[\text{symAB} \quad (\text{symA} :: \text{SymA})\]

\((\text{symB} :: \text{SymB})\)

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

\[= \lambda(ab, ab' :: A + B) \rightarrow \lambda(abab' :: \text{EqAB} ab \ ab') \rightarrow \]

\[\text{case ab of}\]

\[(\text{inl} \ a) \quad \rightarrow \quad \text{case ab'} of\]

\[(\text{inl} \ a') \quad \rightarrow \quad \text{symA} \ a \ a' \ abab'\]

\[(\text{inr} \ b') \quad \rightarrow \quad \text{case abab'} \ of \ \{ \ \} \]

\[(\text{inr} \ b) \quad \rightarrow \quad \text{case ab'} of\]

\[(\text{inl} \ a') \quad \rightarrow \quad \text{case abab'} \ of \ \{ \ \} \]

\[(\text{inr} \ b') \quad \rightarrow \quad \text{symB} \ b \ b' \ abab'\]
Complex Example

- When we made the empty case distinctions, our goal was of type $\text{False'}$.
- Since in those cases $abab' : \text{False'}$, we could have solved the goal as well by directly inserting $abab'$ in those cases.
- The next slide shows this alternative solution.
Alternative Solution

\[ \text{symAB}' \quad (\text{symA} :: \text{SymA}) \]
\[ (\text{symB} :: \text{SymB}) \]
\[ :: \quad \text{SymAB} \]
\[ = \quad \lambda (ab, ab' :: A + B) \to \]
\[ \lambda (abab' :: \text{EqAB} \ ab \ ab') \to \]
\[ \text{case ab of} \]
\[ \quad \begin{array}{c}
\ (\text{inl} \ a) \ \to \quad \text{case ab' of} \\
\quad \begin{array}{c}
\quad \begin{array}{c}
\ (\text{inl} \ a') \ \to \quad \text{symA} \ a \ a' \ abab'
\quad (\text{inr} \ b') \ \to \quad \text{abab'}
\end{array}
\end{array}
\quad \begin{array}{c}
\ (\text{inl} \ a') \ \to \quad \text{abab'}
\quad (\text{inr} \ b') \ \to \quad \text{symB} \ b \ b' \ abab'
\end{array}
\end{array} \]
Remark on Case Distinction

Case distinction over complex expressions causes problems in Agda.

Example (exampleCaseDistinctionComplexExpression.agda):

Assume we have defined \texttt{ProdBool} as the product of two Boolean values:

\[
\text{ProdBool} :: \text{Set} \\
= \text{sig} \\
\quad \text{first} :: \text{Bool}' \\
\quad \text{snd} :: \text{Bool}'
\]
Remark on Case Distinction

Assume we want to define a function as follows:

\[
\begin{align*}
f \quad (pair :: \text{ProdBool}) \\
(p :: \text{atom (and pair.first pair snd)}) \\
:: \text{atom pair.first} \\
= \text{case pair.first of} \\
(tt) & \rightarrow \{! !\} \\
(ff) & \rightarrow \{! !\}
\end{align*}
\]

Although in the case distinction
- we know that \(pair.first = tt\),
- therefore the type of the goal should be \(atom pair.first = True\'),
- Agda won’t accept to insert true there.
Remark on Case Distinction

The reason is that Agda will use in its reduction mechanism
- only reductions from variables to other expressions,
- but no reductions of complex expressions to other expressions.

It would be very expensive to check reductions for complex expressions:
- This would mean to check whether any subexpression of an expression matches the left side of any of those reductions.
- Checking whether a variable which reduces occurs in a expression is instead a cheap operation.
Workaround

One can work around this problem by defining an auxiliary function, which depends on a variable representing the complex expression.

Then make case distinction on this single variable.

In the example above define:

\[
\begin{align*}
  h &\quad (a, b :: \text{Bool}') \\
  (p :: \text{atom (and } a \ b)) &:: \text{atom } a \\
  &= \text{case } a \text{ of} \\
  (tt) &\rightarrow \text{true} \\
  (ff) &\rightarrow \text{case } p \text{ of } \{ \}
\end{align*}
\]
Workaround

Now one can define the function in question in terms of the auxiliary function:

\[
f (pair :: \text{ProdBool})
\]
\[
(p :: \text{atom (and pair.first pair.snd)})
\]
\[
:: \text{atom pair.first}
\]
\[
= h \text{pair.first pair.second}
\]

In the example h had only as arguments the subexpressions of the complex expression in question. In general it might depend on other variables which form the context of the complex expression in question.
(\(x::A\)) \rightarrow B \textit{ vs. } \lambda(x::A) \rightarrow s

There seems to be a confusion about the two expressions

\[(x :: A) \rightarrow B \textit{ vs. } \lambda(x :: A) \rightarrow s\]

\((x :: A) \rightarrow B\) is the dependent function set.
- It is a set (or a type or a kind or a higher kind).
- Because it is a set, it makes sense to talk about
  \[r :: ((x :: A) \rightarrow B)\].
  - \(r :: C\) makes only sense if \(C\) is a set or a type or a kind or a higher kind.

\[\lambda(x :: A) \rightarrow s\] is a function, which applied to \(x :: A\) returns \(s\).
- \(a :: (\lambda(x :: A) \rightarrow s)\) never makes sense, since \(\lambda(x :: A) \rightarrow s\) is not a set or type or (higher) kind.
(x::A) → B vs. \( \lambda(x::A) \rightarrow s \)

- Especially, \( \lambda(x :: A) \rightarrow \text{Set} \) is a function which returns for \( x :: A \) the type \( \text{Set} \).

- Note that

\[
A \quad (b :: B) \\
:: \quad \text{Set} \\
= \quad d
\]

is an abbreviation for

\[
A \quad (b :: B) \rightarrow \text{Set} \\
= \quad \lambda(b :: B) \rightarrow d
\]

- So \( A \) defined as such is a function, not a set.

- It does not make sense to talk about \( c :: A \).

- Would be the same as \( c :: (\lambda(b :: B) \rightarrow d) \).
(\texttt{x::A}) \rightarrow \texttt{B vs. } \lambda(\texttt{x::A}) \rightarrow \texttt{s}

\begin{align*}
A & \quad (b :: B) \\
:: & \quad \text{Set} \\
= & \quad d
\end{align*}

- It does make sense to talk about \( c :: (b :: B) \rightarrow A b \)

- Since \((b :: B) \rightarrow A b\) is a set.
In this section we study, how derivations in dependent type theory correspond to derivations in natural deduction. (Omitted 2005)

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 $A \land B$ from a proof of $A$ and a proof of $B$:

  $$
  \begin{array}{c}
  p : A \\
  q : B
  \end{array}
  \quad
  \Rightarrow
  \quad
  \left< p, q \right> : A \land B
  \quad
  (\times \text{-I})
  $$

- This means that we can derive $A \land B$ from $A$ and $B$. 
Conjunction and Natural Ded.

In so called natural deduction, one has rules for derivating and eliminating formulas formed using the standard connectives.

There the rule for introducing proofs of $A \land B$ is

$$\frac{A \quad B}{A \land B} (\land\text{-}I)$$

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 `sort l` is sorted,
- then, that `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 `∧`.  

CS_336/CS_M36 (part 2) Interactive Theorem Proving; Lentterm 2005, Sec. 4(g)
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 $B$:

$$
\begin{align*}
\frac{p : A \land B}{\pi_0(p) : A} \quad \text{(} \times\text{-El}_0) \\
\frac{p : A \land B}{\pi_1(p) : B} \quad \text{(} \times\text{-El}_1)
\end{align*}
$$

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

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

$$
\begin{align*}
\frac{A \land B}{A} \quad \text{(} \land\text{-El}_0) \\
\frac{A \land B}{B} \quad \text{(} \land\text{-El}_1)
\end{align*}
$$

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 $\text{tt}$ or $\text{ff}$ 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:

$$
\begin{array}{c}
A \land B \\
B \\
\hline
B \land A
\end{array} \quad (\land\text{-El}_0) \\
\begin{array}{c}
A \land B \\
A \\
\hline
B \land A
\end{array} \quad (\land\text{-El}_1)
$$

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 with the disjoint union.

With this identification, the **introduction rules** for $+$ allows to form a proof of $A \lor B$ from a proof of $A$ or from a proof of $B$.

\[
\frac{A : \text{Set} \quad B : \text{Set} \quad p : A}{\text{inl } A \ B \ p : A + B} \quad (+\text{-}\text{I}_{\text{inl}})
\]

\[
\frac{A : \text{Set} \quad B : \text{Set} \quad p : B}{\text{inr } A \ B \ p : A + B} \quad (+\text{-}\text{I}_{\text{inr}})
\]
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:

\[
\begin{align*}
A : \text{Set} & \quad B : \text{Set} & \quad p : A \\
\hline
\text{inl} \; p : A + B
\end{align*}
\]

\( (+-\text{I}_{\text{inl}}) \)

\[
\begin{align*}
A : \text{Set} & \quad B : \text{Set} & \quad p : B \\
\hline
\text{inr} \; p : A + B
\end{align*}
\]

\( (+-\text{I}_{\text{inr}}) \)
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 $\lor$:

\[
\begin{align*}
\frac{A}{A \lor B} \quad (\lor\text{-}I_{\text{inl}}) & \quad \frac{B}{A \lor B} \quad (\lor\text{-}I_{\text{inr}})
\end{align*}
\]

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 ∨ 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 ∨ 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 \( B \):

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

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

$$
\begin{align*}
\text{Case}_+ \quad & sl \quad sr \quad d : C \\
\hline
& d : A \lor B \\
& sl : A \to C \\
& sr : B \to C
\end{align*}
$$

(\text{+-El})

This means that we can derive from $A \lor B$ a formula $C$, if we can derive $C$ from $A$ and from $B$. 
This is what is expressed by the natural deduction
elimination rules for $\lor$:

\[
\begin{array}{c c c}
A \lor B & A \vdash C & B \vdash C \\
\hline
& C & (\lor\text{-El})
\end{array}
\]

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

\[
A \\
\cdot \\
\cdot \\
C
\]
Note that in natural deduction, from the premise $A \vdash C$, we obtain $A \rightarrow C$, which is the premise used in the corresponding rule in dependent type theory.
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 $A_1, \ldots, A_n$):

\[
\begin{align*}
A \lor B & \quad \frac{A \vdash A}{A \lor B \vdash A} \quad \text{(\lor-I_{\text{inr}})} \\
& \quad \frac{A \vdash B \lor A}{B \lor A} \\
& \quad \frac{B \vdash B}{B \lor A \vdash B} \quad \text{(\lor-I_{\text{inr}})} \\
& \quad \frac{B \vdash B \lor A}{B \lor A} \quad \text{(\lor-El)}
\end{align*}
\]

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 $\rightarrow$ 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 $A$:

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

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 $p:A$. 
This is what is expressed by the **introduction rule for** \( \supset \) **in natural deduction**:

\[
\frac{A \vdash B}{A \supset B} \quad (\supset \text{-I})
\]
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 $(A \lor B) \supset (B \lor A)$. 

Example

The complete proof in natural deduction is as follows:

\[
\begin{align*}
A \lor B & \vdash A \lor B \\
A \vdash A & \quad (\lor\text{-}\text{Inr}) \\
B \vdash B & \quad (\lor\text{-}\text{Inl}) \\
A \lor B & \vdash B \lor A & (\lor\text{-El}) \\
(A \lor B) & \supset (B \lor A) & (\supset\text{-}\text{I})
\end{align*}
\]
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** $B$:

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

- 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** $\supset$:

  $\frac{A \supset B \quad A}{B} (\supset \text{-El})$
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.
A proof in natural deduction is as follows:

\[
\begin{align*}
A, A \supset B & \vdash A \supset B, \\
A, A \supset B & \vdash A \\
A, A \supset B & \vdash B \quad (\supset \text{-El}) \\
A & \vdash (A \supset B) \supset B \quad (\supset \text{-I}) \\
A & \vdash (A \supset B) \supset B \quad (\supset \text{-I}) \\
A \supset (A \supset B) & \supset B
\end{align*}
\]
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)} \text{ (\( \to \)-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 \( x : A \).
This is what is expressed by the natural deduction introduction rule for $\forall$:

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

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 \( A \).
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 $n$, $n + 0 == n$.

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):

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

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

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

$$
\forall x : A. B \quad a : A \\
\quad B[x := a] 
$$

(\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:

\[
\forall n : N. 0 + n \equiv n \quad \frac{n : N, m : N \vdash n \equiv n}{n : N, m : N \vdash n + m \equiv N} \quad \frac{n : N, m : N \vdash n + m \equiv (n + m)}{(\forall - \text{I})}
\]

\[
\forall n, m : N. 0 + (n + m) \equiv (n + m) \quad \frac{n : N \vdash \forall m : N. 0 + (n + m) \equiv (n + m)}{(\forall - \text{I})}
\]
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 $\exists x : A \cdot B$ from an element $a : A$ and a proof $p : B[x := a]$:

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

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

$$\frac{a : A \quad B[x := a]}{\exists x : A \cdot B} \quad (\exists \text{-I})$$
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 reads as follows:

\[
\frac{n : N \vdash n : N}{n : N \vdash S \ n : N} \text{(N-I)} \\
\frac{n : N \vdash \exists m : N. m > n}{\forall n : N. \exists m : N. m > n} \text{ (\forall-I)}
\]
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:
- $C : \text{Set}$, which does not depend on $x : A$,
- $p : (\exists x : A. B)$ and
- $x : A, y : B \Rightarrow c : C$.

Then we have $c[x := \pi_0(p), y := \pi_1(p)] : C$, not depending on $x:A$ or $y:B$. 
Existential Quantification (Cont.)

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

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

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
  \[ \forall n : \mathbb{N}. \exists m : \mathbb{N}. m > n \land \text{Prime}(m). \]
  
- We want to show that for all \( n \) there exist two primes above it, i.e.
  \[ \forall n : \mathbb{N}. \exists m : \mathbb{N}. \exists k : \mathbb{N}. m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k). \]

- 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 \( k > m \land \text{Prime}(k) \).
Example

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

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

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

we prove the following

\[ \begin{array}{l}
    k : \mathbb{N} \quad m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) \\
    m : \mathbb{N} \quad \exists k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) \\
    \exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) \\
\end{array} \]

(∃-I)

(∃-I)

So we have shown

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

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

\[
\begin{align*}
\exists k : \mathbb{N}. k > m & \land \text{Prime}(k) \\
\exists m, k : \mathbb{N}. m > n \land k > m & \land \text{Prime}(m) \land \text{Prime}(k) \quad \text{(\(\exists\)-I)}
\end{align*}
\]

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

Third step: Again we can conclude

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

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

\[
\begin{align*}
  n : \mathbb{N} & \vdash \exists m : \mathbb{N}. m > n \land \text{Prime}(m) \\
  n : \mathbb{N}, m : \mathbb{N}, m > n \land \text{Prime}(m) & \vdash \exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) \\
  \forall n : \mathbb{N}. \exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) & \quad (\exists \text{-I}) \\
  \forall n : \mathbb{N}. \exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) & \quad (\forall \text{-I})
\end{align*}
\]
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** \( A \rightarrow B \), and we have

\[
\lambda x^A. \pi_1(p \, x) : (\forall x : A. C(x, f \, x))
\]
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 \( x \).
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.

Now:

- Any function in type theory is recursive.
- We cannot decide the Turing halting problem, i.e. we cannot decide for a Turing machine whether it halts or not.

Therefore we cannot prove in type theory

$$\forall x : \text{Turing\_Machine}.(x \text{ halts} \lor \neg(x \text{ halts}))$$
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.
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.

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

\[
\text{halts } x \lor \neg \text{halts } x.
\]

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 \( x \).
Unprovability in Type Theory

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.

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 $\neg \neg A \supset A$ don’t hold for all formulae $A$.

- As usual in natural deduction, we write in the following $\bot$ for False.

- Jump over remaining slides
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 replace $\exists x : A . B$ and $A \lor B$ by the above formulae, then all formulae provable in classical logic are derivable.

This requires $\neg \neg A \supset A$, which can be shown for all formulae built from decidable atomic formulae using $\neg$, $\supset$, $\land$, $\forall$.

The formula $A \lor \neg A$ translates into $\neg (\neg A \land \neg \neg A)$, which trivially holds, since $\neg A$ and $\neg \neg A$ implies $\bot$.

In this sense, type theory contains classical logic, but is richer, since it has as well so called strong disjunction and existential quantification.
Constructive Logic (Cont.)

- **Weak disjunction and existential quantification** is expressed by the formulae $\neg(\neg A \land \neg B)$ and $\neg \forall x : A. \neg B$.

- When using only weak disjunction, existential quantification and decidable atomic formulae, we obtain classical logic.

- **Strong disjunction and existential quantification** is expressed by the original type theoretic formulae.

**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, i.e. writing

postulate p :: A ∨ ¬A

- Jump over the following proofs.
Note that weak disjunction and existential quantification don’t have the same constructive content as strong disjunction and existential quantification.

From $p : \neg (\neg A \land \neg B)$ we cannot in general decide whether $A$ or $B$ holds.

From $p : \neg \forall x : A. \neg B$ we cannot in general extract an $a : A$ s.t. $B[x := a]$ holds.
Constructive Logic (Cont.)

Proof (using classical logic) of

\[ \exists x : A \land B \leftrightarrow (\neg \forall x : A \land \neg B) \]

We have classically:

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

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.
We show intuitionistically
\((\neg \exists x : A.B) \iff (\forall x : A.\neg B)\):

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):

\((\exists x : A.B) \iff (\neg \neg \exists x : A.B) \iff (\neg \forall x : A.\neg B)\)
Constructive Logic (Cont.)

Proof of

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

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

- 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):

\[
(A \lor B) \leftrightarrow \neg\neg(A \lor B) \leftrightarrow \neg(\neg A \land \neg B)
\]
We show that for formulas $A$ built from $\neg$, $\supset$, $\land$, $\forall$ and decidable prime formulae we have

$$\neg A \supset A.$$ 

The formula $\neg \neg A \supset A$ is called **stability for $A$**.

This is done by induction over the builtup of these formulae.
Case $A \equiv$ atom $c$.

We make case distinction on $c$.

If $c = \text{tt}$, then we have $A \equiv \text{True}$, $A$ is provable, therefore as well $\neg\neg A \vdash A$.

If $c = \text{ff}$, then we have $A \equiv \text{False} \equiv \bot$.

Assume $\neg\neg A \equiv (\bot \dashv \bot) \dashv \bot$.

$\bot \dashv \bot$ is provable.

Therefore we obtain $\bot$, which is $A$.

So we have

$$\neg\neg A \vdash A$$

and obtain

$$\neg\neg A \vdash A$$.
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.
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 $A \equiv \forall x : B.C$, 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 $\bot$.

We show $\neg A$.

Assume $A$, which is $\forall x : B.C$.

Then we obtain $C$, and by $\neg C$ therefore $\bot$.

By $\neg \neg A$ we therefore get $\bot$, and are done.
Class. Logic for $\exists$, $\lor$-free Formulae

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

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

$\bot \equiv \text{False} = \text{atom false}$.

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 $B \supset \bot \equiv \neg B$. 
(h) 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.
Let $S$ be a type theoretic notation for the operation $x \mapsto x + 1$.

Then the type theoretic rules are

\[\begin{align*}
N : \text{Set} & \quad (\text{N-F}) \\
0 : N & \quad (\text{N-I}_0) \\
\frac{n : N}{S \ n : N} & \quad (\text{N-I}_S)
\end{align*}\]
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 \( f : \mathbb{N} \rightarrow \mathbb{N} \), s.t.
- \( f 0 = a \),
- \( f (S n) = g n (f n) \).
The **computation of** \( f \ n \) proceeds now as follows:

- Compute \( n \).
- If \( n = 0 \), then the result is \( a \).
- Otherwise \( n = S \ n' \).
  - 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 \( f : \mathbb{N} \rightarrow \mathbb{N} \) with \( f \ n = 2 \cdot n \) can be defined \textit{primitive recursively} by:

\begin{align*}
    f \ 0 & = 0. \\
    f \ (S \ n) & = S \ (S \ (f \ n)).
\end{align*}

Therefore take in the definition above:

\begin{align*}
    a & = 0, \\
    g \ n \ x & = S \ (S \ x).
\end{align*}
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

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

Introduction Rules

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

\[ \frac{n : N}{S\ n : N} \quad (\text{N-I}_S) \]
Rules for the Natural Numbers

Elimination Rule

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

Equality Rules

\[
\begin{align*}
P \ C \ a \ f \ 0 & = a \quad (N-\text{Eq}_0) \\
P \ C \ a \ f \ (S \ n) & = f \ n \ (P \ C \ a \ f \ n) \quad (N-\text{Eq}_S)
\end{align*}
\]

Additionally we have the **Equality versions** of the formation-, introduction- and elimination-rules.
Note that if we define in the elimination rule \( g := P \ C \ f \) then

The conclusion of the elimination rule reads:

\[
g n : C \ n
\]

which means that

\[
\lambda n \in \mathbb{N}. g \ n : (n : \mathbb{N}) \rightarrow C \ n.
\]

The equality rules read:

\[
g 0 \quad = \quad a
\]
\[
g (\text{S} \ n) \quad = \quad f \ n \ (g \ n)
\]
The more compact notation is:

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

The same equality rules as before.
Natural Numbers in Agda

- \( N \) is defined using `data`:

\[
data \ N = Z \mid S(n :: N)
\]

(We cannot use 0 for a constructor, since this denotes the builtin native natural number 0 in Agda).

- Therefore we have

\[
\begin{align*}
Z &:: N \\
S &:: N \rightarrow N
\end{align*}
\]
Elimination Rules for N in Agda

- Elimination is represented in Agda as before via case distinction.

- Assume we want to define

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

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

- Then we can type into the goal \( n \) and use the menu `agda-case`. 
We get

\[
\begin{align*}
  f \quad & (n :: N) \\
  :: & A \\
  = & \text{case } n \text{ of} \\
    (Z) & \rightarrow \{! \} \\
    (S \ n') & \rightarrow \{! \}
\end{align*}
\]
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 use menu item “check-termination”, we might obtain an error-message.

If we

- do not make use of \( f \) in the case \( n = Z \) and
- only use of \( f \ n' \) in case \( n = S \ n' \).

then check-termination succeeds.
If \textit{check-termination succeeds}, the definition should be \textit{correct}.

(The lecturer hasn’t checked the algorithm).

However, \textit{if check-termination fails}, the \textit{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:

\[
(\text{one} := S \, Z):
\]

\[
\begin{align*}
\text{fib} \; (n :: N) &:: N \\
&= \text{case } n \text{ of} \\
(Z) &\rightarrow \text{one} \\
(S \, n') &\rightarrow \text{case } n' \text{ of} \\
(Z) &\rightarrow \text{one} \\
(S \, n'') &\rightarrow \text{fib} \, n' + \text{fib} \, n''
\end{align*}
\]
Limitations of Termination Checker

Assume we define the **predecessor function**

\[
\text{pred} \ (n :: \mathbb{N})
\]

\[
:: \mathbb{N}
\]

\[
= \begin{cases}
(\mathbb{Z}) & \rightarrow \mathbb{Z} \\
(S \ n') & \rightarrow n'
\end{cases}
\]

i.e

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

\[ f \ (n :: N) :: N = \text{case } n \text{ of} \]

\[ (Z) \rightarrow Z \]

\[ (S \ n') \rightarrow f \ (\text{pred } n) \]

terminates always

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

However, \textit{check-termination fails}.
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.

- Omit treatment of simultaneous recursion.
Limitations of Termination Checker

- Unfortunately, Agda does currently not deal with **simultaneous recursion**
  - i.e. the situation, where we decrease in one case w.r.t. one variable, in another case w.r.t. another variable.

- In order to deal with this situation, one has to **rearrange proofs**.

- On next slide there is an example of a proof which results in non-termination, although each recursive call descends.
  - Refers to a definition of \((<) :: (n, m :: N) \rightarrow \text{Set}\) which will be introduced below.
Example

\[
\text{mono } (n, k, m :: N)(p :: n < k) :: (n + m) < (k + m) \\
= \text{ case } n \text{ of } \\
\quad (Z) \rightarrow \text{ case } k \text{ of } \\
\quad \quad (Z) \rightarrow \text{ case } p \text{ of } \{ \} \\
\quad \quad (S \ k') \rightarrow \text{ case } m \text{ of } \\
\quad \quad \quad (Z) \rightarrow \text{ true} \\
\quad \quad \quad (S \ m') \rightarrow \text{ mono } n \ k \ m' \ p \\
\quad \quad (S \ n') \rightarrow \text{ case } k \text{ of } \\
\quad \quad \quad (Z) \rightarrow \text{ case } p \text{ of } \{ \} \\
\quad \quad \quad (S \ k') \rightarrow \text{ case } m \text{ of } \\
\quad \quad \quad \quad (Z) \rightarrow \text{ mono } n' \ k' \ m \ p \\
\quad \quad \quad \quad (S \ m') \rightarrow \text{ mono } n \ k \ m' \ p
\]
The following version will be accepted by the termination checker:

(this version corresponds exactly to induction on $m$)

\[
\text{mono } (n, k, m :: \mathbb{N}) \\
(p :: n < k) \\
:: n + m < k + m \\
= \text{ case } m \text{ of} \\
(Z) \rightarrow p \\
(S \, m') \rightarrow \text{mono } n \, k \, m' \, p
\]
Amendment of Non-Termin. Version

- If one cannot reduce a non-terminating version directly in one with only one descend, one can use auxiliary lemmata instead.

- For instance in the previous non-terminating version, if one doesn’t observe the previous much better solution, one can
  - replace the first reference to mono by a reference to a lemma.
  - (this change is not really necessary, since only the second reference is responsible for rejection by the termination checker)
  - and observe that the second reference can be replaced by \( p \).
Amendment of Non-Termin. Version

\[
\text{lemma } (m, k : \mathbb{N}) \\
\quad :: Z + m < S k + m \\
= \text{ case } m \text{ of } \\
\quad (Z) \rightarrow \text{ true} \\
\quad (S \, m') \rightarrow \text{ lemma } m' \, k
\]
mono \((n, k, m :: \mathbb{N})(p :: n < k) :: (n + m) < (k + m)\)

\[
= \text{case } n \text{ of } \\
\hspace{2em} (Z) \rightarrow \text{case } k \text{ of } \\
\hspace{4em} (Z) \rightarrow \text{case } p \text{ of } \{ \} \\
\hspace{4em}(S \ k') \rightarrow \text{case } m \text{ of } \\
\hspace{6em}(Z) \rightarrow \text{true} \\
\hspace{6em}(S \ m') \rightarrow \text{lemma } m' \ k' \\
\hspace{2em}(S \ n') \rightarrow \text{case } k \text{ of } \\
\hspace{4em}(Z) \rightarrow \text{case } p \text{ of } \{ \} \\
\hspace{4em}(S \ k') \rightarrow \text{case } m \text{ of } \\
\hspace{6em}(Z) \rightarrow p \\
\hspace{6em}(S \ m') \rightarrow \text{mono } n \ k \ m' \ p
\]
Example: Addition

Definition of $+$ in Agda:

$$(+) \ (n, m :: N)$$

$\ :: \ N$

$= \ case \ m \ of$

$\ (Z) \rightarrow n$

$\ (S \ m') \rightarrow S \ (n + m')$

The definition expresses:

$$n + 0 = n$$

$$n + (m' + 1) = (n + m') + 1$$
Note that (+) is used **infix**, i.e. we write \( n + m \) for 
\((+) n m\).

If \( m = S m' \), the definition of \((+) n m\) refers to \((+) n m'\),
\((+) n m'\) is **defined before** \((+) n m\) since 
\( m' \) is introduced before \( m \).
Example: Multiplication

Definition

\[(*) \quad (n, m :: \mathbb{N}) \quad :: \quad \mathbb{N} \quad = \quad \text{case } m \text{ of} \]

\[\begin{align*}
(Z) & \quad \rightarrow \quad Z \\
(S \ m') & \quad \rightarrow \quad n \times m' + n
\end{align*}\]

The definition expresses:

\[n \cdot 0 = 0\]
\[n \cdot (m' + 1) = (n \cdot m') + n\]
Example: Multiplication (Cont.)

- Again $\ast$ is treated infix.
- Agda has built in that $\ast$ binds more than $\mathord{+}$.
  - $n \ast m' + n$ is treated as $(n \ast m') + n$.
- Note that the definition of $\ast$ requires, that $\mathord{+}$ is already defined.
Equality on N

The equality \((n == m) :: \text{Set}\) for \(n, m :: \mathbb{N}\) can be defined using the equations:

- \((Z == Z) = \text{True}'\).
- \((Z == S n) = (S n == Z) = \text{False}'\).
- \((S n == S m) = (n == m)\).
From this one can now derive a definition in Agda:

\[
(==) \quad (n, m :: N) \\
:: \quad \text{Set} \\
= \quad \text{case } n \text{ of} \\
\quad (Z) \quad \rightarrow \quad \text{case } m \text{ of} \\
\quad \quad (Z) \quad \rightarrow \quad \text{True}' \\
\quad (S \ m') \quad \rightarrow \quad \text{False}' \\
\quad (S \ n') \quad \rightarrow \quad \text{case } m \text{ of} \\
\quad \quad (Z) \quad \rightarrow \quad \text{False}' \\
\quad (S \ m') \quad \rightarrow \quad (n' == m')
\]
Alternatively, one could have defined first a Boolean valued equality

\[ \text{EqNBool} : \mathbb{N} \to \mathbb{N} \to \text{Bool} \]

on \( \mathbb{N} \) and then defined

\[ n == m = \text{atom}(\text{EqNBool} \ n \ m) . \]
Reflexivity of ==

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

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

- **Type theoretically** this means that we have to define a function `refl`:

\[
\text{refl } (n : \mathbb{N}) \\
:: \quad n == n \\
= \quad \{ ! ! \}
\]
Reflexivity of $==$ (Cont.)

This can now be shown using case distinction:

\[
\text{refl } (n : \text{N}) \\
:: \ n == n \\
= \ \text{case } n \ \text{of} \\
\quad \ (Z) \ \to \ \{! \ !\} \\
\quad \ (S \ n') \ \to \ \{! \ !\}
\]
Reflexivity of $==$ (Cont.)

- Case $n = Z$ is trivial.
- Case $n = S\ n'$ can be solved using $\text{refl}\ n'$ (which is defined before $\text{refl}\ n'$).
Symmetry of ==

- **Symmetry** of \(==\) is the formula:

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

- **Type theoretically** this means that we have to define a function \(\text{sym}\):

\[
\begin{align*}
\text{sym} & \quad (n, m : N) \\
& \quad (p :: n == m) \\
& \quad :: m == n \\
& \quad = \quad \{! !\}
\end{align*}
\]
Symmetry of == (Cont.)

This can now be shown using case distinction:

\[
\text{sym } (n, m : \mathbb{N}) \\
(p :: n == m) \\
:: \ m == n \\
= \ \text{case } n \ \text{of} \\
\quad (\mathbb{Z}) \rightarrow \ \text{case } m \ \text{of} \\
\quad \quad (\mathbb{Z}) \rightarrow \ \{! \ !\} \\
\quad \quad (S \ m') \rightarrow \ \{! \ !\} \\
\quad (S \ n') \rightarrow \ \text{case } m \ \text{of} \\
\quad \quad (\mathbb{Z}) \rightarrow \ \{! \ !\} \\
\quad \quad (S \ m') \rightarrow \ \{! \ !\} 
\]
Symmetry of $==$ (Cont.)

- The **first goal** can be solved by using true (since $(Z == Z) = \text{True}'$).

- For the **second goal** we know $p$ is an element of $Z == \text{S } m'$ which is False'.
  
  Therefore if we make **case distinction on** $p$ we get

  $$\text{case } p \text{ of } \{ \}$$

  and have solved the second goal.

- Similarly the **third goal can be solved**.
Symmetry of $==$ (Cont.)

In the fourth goal, we have as type of goal $S \, m' == S \, n'$ which is identical to $m' == n'$. The type of $p$ is $S \, n' == S \, m'$ which is identical to $n' == m'$.

The goal can be solved by using $\text{sym} \, n' \, m' \, p$.

Note that we can use here $p$ since it is of type $n' == m'$. It is correct to use it since $n'$ is introduced before $n$.

Therefore

$\text{sym} \, n'$ can be defined before $\text{sym} \, n$.

This definition will be accepted by check-termination.
Example: $<$ on $\mathbb{N}$

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

$$(<) \ (n, m :: \mathbb{N})$$

:: Set

= case $m$ of

$(Z) \rightarrow$ False

$(S \ m') \rightarrow$ case $n$ of

$(Z) \rightarrow$ True

$(S \ n') \rightarrow$ $n' < m'$
Example: Tuples of Length \( n \)

- We define tuples (or vectors) of length \( n \) in Agda.
- Define first

\[
\text{data Nil} \; = \; \text{nil}
\]

\[
\text{Cons} \; (A, B :: \text{Set}) \; :: \; \text{Set} \\
= \; \text{data cons}(a :: A)(b :: B)
\]
Tuples of Length n

Now we can define

\[
\text{Tuple} \ (A :: \text{Set}) \\
(n :: N) \\
:: \text{Set} \\
= \text{case } n \text{ of} \\
\quad (Z) \rightarrow \text{Nil} \\
\quad (S \ m') \rightarrow \text{Cons } A \ (\text{Tuple } A \ m')
\]
Tuples of Length $n$

Therefore (with the obvious definition of two),

\[ \text{Tuple } A \, n = \text{Cons } A (\text{Cons } A \cdots (\text{Cons } A \text{ Nil}) \cdots )) \, . \]

$n$ times

The elements of Tuple $A \, n$ are

\[ \text{cons } a_1 (\text{cons } a_2 \cdots (\text{cons } a_n \text{ nil}) \cdots ) \]

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

$$\text{Tuple}(A, 0) := \{\langle \rangle \} ,$$
$$\text{Tuple}(A, n + 1) := \{\langle a_1, \ldots, a_{n+1} \rangle \mid a_1, \ldots, a_{n+1} \in A\}.$$
Remarks on Tuples of Length \( n \)

If we define

\[
\begin{align*}
\text{nil} & := \langle \rangle , \\
\text{cons } a_1 \langle a_2, \ldots, a_{n+1} \rangle & := \langle a_1, \ldots, a_{n+1} \rangle ,
\end{align*}
\]

then this reads:

\[
\begin{align*}
\text{Tuple}(A, 0) & := \{\text{nil}\} , \\
\text{Tuple}(A, n + 1) & := \{\text{cons } a \ b \mid a \in A \land b \in \text{Tuple}(A, n)\} .
\end{align*}
\]
Remarks on Tuples of Length $n$

In the type theoretic definition we have constructors:

- $\text{nil} :: \text{Tuple} \ A \ Z$
- $\text{cons}@(\text{Tuple} \ A \ (S \ n)) :: A \rightarrow \text{Tuple} \ A \ n \rightarrow \text{Tuple} \ A \ (S \ n)$.

This is the type theoretic analogue of the previous definitions.
Example: Sum of n-Tuples

Define

\[\text{NTuple } (n :: N) :: \text{Set} = \text{Tuple N } n\]

NTuple \(n\) are tuples of natural numbers of length \(n\).
Componentwise Sum of n-Tuples

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

Using mathematical notation, this sum for instance as follows:

$$\langle 2, 3, 4 \rangle + \langle 5, 6, 7 \rangle = \langle 7, 9, 11 \rangle .$$
Componentwise Sum of n-Tuples

\[
\text{sumNTuple} \ (n :: N) \\
(aavec, bvec :: NTuple n) \\
:: \ NTuple n \\
= \ \text{case } n \ \text{of} \\
\quad (Z) \to \ \text{nil} \\
\quad (S \ n') \to \\
\qquad \text{case } aavec \ \text{of} \\
\qquad \quad (\text{cons } a \ aavec') \to \\
\qquad \quad \text{case } bvec \ \text{of} \\
\qquad \quad \quad (\text{cons } b \ bvec') \to \\
\qquad \quad \quad \text{cons@\_} (a + b) \\
\qquad \quad \quad (\text{sumNTuple } n' \ aavec' \ bvec')
\]
(i) Lists

We define the set of lists of elements of type $A$ in Agda.

We have two constructors:
- $\text{nil}$, generating the empty list.
- $\text{cons}$, adding an element of $A$ in front of a list

So we define lists as:

$$\text{list } (A :: \text{Set})$$
$$:: \text{Set}$$
$$= \text{data nil}$$
$$\mid \text{cons}(a :: A)(l :: \text{list } A)$$
Elimination Rule for Lists

Elimination rule uses list-recursion:
Assume
- $A : \text{Set}$
- $C :: \text{Set, depending on } l :: \text{list } A.$

Then we can define

$$f (\text{l :: list } A) :: C = \text{case } l \text{ of}$$

$$(\text{nil}) \rightarrow \{! !\}$$

$$(\text{cons } a \text{ } l') \rightarrow \{! !\}$$

and in the second goal we can make use of $f \text{ } l'$. 
Example: Length of a List

\[
\text{length } (l :: \text{list } N) :: N = \begin{cases} 
\text{nil} & \rightarrow \ Z \\
\text{cons } a \ l' & \rightarrow \ S \ (\text{length } l') 
\end{cases}
\]
Example: sumlist

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

\[
\text{sumlist} \quad (l :: \text{list } N) \\
:: \quad N \\
= \quad \text{case } l \text{ of} \\
\quad (\text{nil}) \quad \rightarrow \quad Z \\
\quad (\text{cons } n \ l') \quad \rightarrow \quad n + \text{sumlist } l'
\]
Interesting Exercise

Define

\[ \text{append} : (A : \text{Set}) \rightarrow (\text{list } A) \rightarrow (\text{list } A) \rightarrow \text{list } A \ , \]

s.t. \( \text{append } A \ l \ l' \) is the result of appending the list \( l' \) at the end of list \( l \).

E.g., if \( a, b, c, d \) are elements of \( A \), and if we define \( \text{cons} := \text{cons@}(\text{list } A) \), \( \text{nil} := \text{nil@}(\text{list } A) \), then:

\[
\text{append } A \ (\text{cons } a \ (\text{cons } b \ \text{nil})) \ (\text{cons } c \ (\text{cons } d \ \text{nil})) \\
= \text{cons } a \ (\text{cons } b \ (\text{cons } c \ (\text{cons } d \ \text{nil})))
\]
(j) Universes

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

So we have

- $U : \text{Set}$,
- $T : U \rightarrow \text{Set}$ (the decoding function).

We consider in the following a universe closed under

- False, True, Bool, N,
- $+$,
- $\Sigma$,
- the dependent function type.
Rules for the Universe

Formation Rule

\[
\begin{align*}
  \text{U : Set} & \quad (U-F) \\
  a : \text{U} & \quad (T-F) \\
  T \ a : \text{Set} & \quad \text{\hspace{1cm} (T-F)}
\end{align*}
\]
Rules for the Universe

Introduction and Equality Rules

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

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

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

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

Introduction and Equality Rules (Cont.)

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

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

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

\[ T (\hat{\Sigma} a\ b) = \Sigma (T a) \ (\lambda x^T a . T (b \ x)) : \text{Set} \quad (T-Eq_{\hat{\Sigma}}) \]
Rules for the Universe

Introduction and Equality Rules (Cont.)

\[
\begin{align*}
\frac{a : U \quad b : T \ a \rightarrow U}{\widehat{\Pi} \ a \ b : U} \quad (U-I_{\widehat{\Pi}}) \\
T (\widehat{\Pi} \ a \ b) = (x : T \ a) \rightarrow T (b \ x) : Set \quad (T-Eq_{\widehat{\Pi}})
\end{align*}
\]
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 \textit{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 \textit{Set} and use \textit{T} to obtain the required function.
Applications of the Universe

Example: Define

\[
\begin{align*}
\hat{\text{atom}} & : \text{Bool} \to \text{U} , \\
\hat{\text{atom}} & := \text{Case}_{\text{Bool}} (\lambda x^{\text{Bool}}. \text{U}) \text{ True } \text{ False} , \\
\hat{\text{atom}} & : \text{Bool} \to \text{Set} , \\
\hat{\text{atom}} & : \lambda x^{\text{Bool}}. \text{T} (\hat{\text{atom}} x) ,
\end{align*}
\]

Then

- \( \hat{\text{atom}} \text{ tt } = \text{True} , \)
- \( \hat{\text{atom}} \text{ ff } = \text{False} . \)
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.)

mutual

\[
\begin{align*}
U & :: \text{Set} \\
= & \text{data Nhat} \\
& | \text{Falsehat} \\
& | \text{Truehat} \\
& | \text{Boolhat} \\
& | Nhat \\
& | \text{Sigmahat}(a :: U)(b :: T a \to U) \\
& | \text{Pihat}(a :: U)(b :: T a \to U)
\end{align*}
\]
Universes in Agda (Cont.)

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

\[
T \ (u :: U) :: Set = \text{case } u \text{ of}
\]
\[
(N \text{hat}) \to N \\
(True \text{hat}) \to True \\
(False \text{hat}) \to False \\
(Bool \text{hat}) \to Bool \\
(N \text{hat}) \to N \\
(\text{Sigma} \hat{a} \ b) \to \text{Sigma} (T \ a) \\
(\lambda (x :: T \ a) \to T \ (b \ x)) \\
(\text{Pio} \hat{a} \ b) \to (x :: T \ a) \to T \ (b \ x)
\]
(k) 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.

\[
A :: \text{Set} = \text{data } C_1 (a_{11} :: A_{11}) \cdots (a_{1n_1} :: A_{1n_1}) \\
| C_2 (a_{21} :: A_{21}) \cdots (a_{2n_2} :: A_{2n_2}) \\
| \cdots \\
| C_m (a_{m1} :: A_{m1}) \cdots (a_{mn_m} :: A_{mn_m})
\]
Meaning of “data”

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

$$C_i @ A :: (a_{i1} :: A_{i1})$$

$$\rightarrow \ldots$$

$$\rightarrow (a_{in_i} :: A_{in_i})$$

$$\rightarrow A$$

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 $A_{ij}$ we can make use of $A$.

However, it is difficult to understand $A$, if we have negative occurrences of $A$.

Example:

$A :: Set$

$= \text{data } C \ (f :: A \rightarrow A)$

What is the least set $A$ having a constructor

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

$\rightarrow A$ ?
Strictly Positive Algebraic Types

- If we have constructed some part of $A$ already,
  find a function $f :: A \rightarrow A$, and
  add $\text{C@}_f$ to $A$,
  then $f$ might no longer be a function $A \rightarrow A$.
  ($f$ applied to the new element $\text{C@}_f$ might not be defined).

- In fact, “agda-check-termination” 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:

\[
\text{NList}:: \text{Set} = \text{data} \text{ nil} \mid \text{cons} (a :: \text{N}) (l :: \text{NList})
\]

The constructor \text{cons} of N-lists refers to NList, but in a positive way:

We have: if \( a :: \text{N} \) and \( l :: \text{NList} \), then we have \( \text{cons} a l :: \text{NList} \).
Strictly Positive Algebraic Types

If we add `cons@_ a l` to `NList`, the reason for adding it (namely `l :: NList` is not destroyed by this addition.

So we can “construct” the set `NList` by
- starting with the empty set,
- adding `nil@_` and
- closing it under `cons@_` whenever possible.

Because we can “construct” `NList`, the above is an acceptable definition.
Strictly Positive Algebraic Types

In general:

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

\[ = \text{data } C_1 \ (a_{11} :: A_{11}) \cdots (a_{1n_1} :: A_{1n_1}) \]

\[ | \ C_2 \ (a_{21} :: A_{21}) \cdots (a_{2n_2} :: A_{2n_2}) \]

\[ \cdots \]

\[ | \ C_m \ (a_{m1} :: A_{m1}) \cdots (a_{mn_m} :: A_{mn_m}) \]

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.

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

The set of binary trees can be defined as follows:

\[
\text{Bintree} :: \text{Set} \\
= \text{data leaf} \\
\mid \text{branch (left :: Bintree) (right :: Bintree)}
\]

This is a strictly positive algebraic type.
An often used extension is to define several sets simultaneously inductively.

Example: the even and odd numbers:

```
mutual

Even :: Set
    = data Z | S (n :: Odd)

Odd :: Set
    = data S (n :: Even)
```

In such examples the constructors refer strictly positive to all sets which are to be defined simultaneously.
We can even allow $A_{ij} = B_1 \rightarrow A$ or even
$A_{ij} = B_1 \rightarrow \cdots \rightarrow B_l \rightarrow A$, where $A$ is one of the types introduced simultaneously.

Example (called “Kleene’s O”):

$$O:: Set$$
$$\quad = \text{data leaf}$$
$$\quad \quad \mid \text{succ } (o :: O)$$
$$\quad \quad \mid \text{lim } (f :: N \rightarrow O)$$

The last definition is unproblematic, since, if we have $f :: N \rightarrow O$ 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 $f\,(C@\_\, a_1 \,\cdots\, a_n)$, 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

\[ f \left( \text{branch@}_\text{left right} \right) \]

can make use of \( f \text{left} \) and \( f \text{right} \).
Examples

In the example of $O$, when defining

$$g :: O \rightarrow A$$

by case-distinction, then the definition of

$$g \left( \lim @_f \right)$$

can make use of $g\left(f \; n\right)$ for all $n :: N$. 