B3. Data Types

(a) The set of Booleans.
(b) The finite sets.
(c) Atomic formulae and the traffic light example.
(d) The disjoint union of sets.
(e) The $\Sigma$-set.
(f) The set of natural numbers.
(g) Lists.
(h) Universes.
(i) Algebraic data types.
(a) The Set of Booleans

\[
\text{Case Bool} \quad \begin{array}{c}
\text{if cond : } C \\
\text{then } \text{if } \text{true : } C \\
\text{if } \text{false : } C
\end{array} \\
\Rightarrow \text{if cond : } C
\]\n
Elimination Rule

Introduction Rules

Formation Rule
Equality Rules

The Set of Boolans (Cont.)
In the above:

- \texttt{tt}, \texttt{ff} are the \textbf{constructors} of \texttt{Bool}.

\begin{align*}
\text{\texttt{cond} } & \leftarrow \\
(\text{\texttt{cond} : Bool}) & \leftarrow \\
(\texttt{if} \; \texttt{cond} & \leftarrow \\
(\texttt{ec} : \texttt{cond}) & \leftarrow \\
(\texttt{tt} : \texttt{cond}) & \leftarrow \\
\text{\texttt{CaseBool} : Bool} & \leftarrow \\
\texttt{Set} & \leftarrow
\end{align*}

- \texttt{CaseBool} : \texttt{Bool}.

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

\begin{align*}
& \text{\texttt{cond}} \leftarrow \\
& (\texttt{cond} : \texttt{Bool}) \leftarrow \\
& (\texttt{if} \; \texttt{cond} \leftarrow \\
& (\texttt{ec} : \texttt{cond}) \leftarrow \\
& (\texttt{tt} : \texttt{cond}) \leftarrow \\
& \texttt{CaseBool} : \texttt{Bool} \leftarrow \\
& \texttt{Set} \leftarrow
\end{align*}

\begin{itemize}
\item \texttt{tt}, \texttt{ff} are the \textbf{constructors} of \texttt{Bool}.
\item For "\texttt{condition}".
\item \texttt{ec} stands for "\texttt{if-case}, \texttt{if-case}".
\item \texttt{tt} stands for \texttt{true}, \texttt{ff} stands for \texttt{false}.
\end{itemize}
Remarks (Cont.)

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

\[ y(p : \text{Bool}) \]

So we obtain functions from \text{Bool} into other sets without having to write

\[
\begin{align*}
\text{if } \text{ec} = \text{ff} & \rightarrow f = \text{CaseBool}^{\text{ff}}
\text{if } \text{ff} = \text{ff} & \rightarrow f = \text{CaseBool}^{\text{ff}}
\text{if } \text{ec} = \text{tt} & \rightarrow f = \text{CaseBool}^{\text{tt}}
\end{align*}
\]

Notice that we then get for \text{C : Bool} → \text{Set, } \text{ic} : \text{C tt, } \text{ec} : \text{C ff}.

\[ \text{f} \]
This is similar to the definition of for instance (+) in curried form in Haskell.

Shorter than writing \( \lambda x. x + 3 \).
Remarks (Cont.)

- Note that we have the following order of the arguments of `CaseBool`:
  - 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 `CaseBool` is a “then…else…if” – the condition (if ...) is the last one.
AND is the conjunction:

\[
\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} = \lambda q, c : \text{Bool}. \\text{case Bool} (\lambda q, c : \text{Bool}. c \iff q)
\]

Example

\[
\text{ AND } = \iff
\]
Derivation of AND:

First we derive \( \text{AND} : \text{Bool} \Rightarrow \text{Bool} \).

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]

\[ \text{AND} : \text{Bool} \Rightarrow \text{ Bool} \]
We derive

Example (Cont.)

(C) Anton Setzer 2003 (except for pictures)
Similarly follows

\begin{align*}
& b : \text{Bool},
& c : \text{Bool} 
& \Rightarrow \text{Bool} = \\
& (\lambda (b' : \text{Bool}). \text{Bool}) \quad \text{ff} : \text{Type}
\end{align*}
Using part of the proof above, we derive

\[ q : \text{Bool}, c : \text{Bool} \]
\[ \vdash q : \text{Bool}, c : \text{Bool} \]
\[ \vdash \texttt{Context} \]

- We derive

\[ q : \text{Bool}, c : \text{Bool} \]
\[ \vdash q : \text{Bool}, c : \text{Bool} \]
\[ \vdash \texttt{Context} \]

- Using part of the proof above, we derive

Example (Cont.)
Example (Cont.)

We derive \( q \in \text{Bool}, c \in \text{Bool} \) using part of the proof above:

\[
\begin{align*}
q &: \text{Bool} \\
\text{context} &\iff \text{Context}
\end{align*}
\]

\ldots
Finally we obtain our judgment (we stack the premises of the rule because of lack of space):
We can extend and elimination and equality rules, having as result Type Elimination Rule into Type:

\[
\text{C} : \text{Bool} \leftarrow \text{Type}
\]

\[
\begin{array}{c}
\text{Case Bool} \quad \text{ic ec ec} = \text{ec} \\
\text{C} : \text{Bool} \leftarrow \text{Type}
\end{array}
\]

Equality Rules into Type:

\[
\begin{array}{c}
\text{C} : \text{Bool} \leftarrow \text{Type} \\
\text{Case Bool} \quad \text{ic ec ec} = \text{ec} \\
\text{C} : \text{Bool} \leftarrow \text{Type}
\end{array}
\]

\[
\begin{array}{c}
\text{C} : \text{Bool} \leftarrow \text{Type} \\
\text{Case Bool} \quad \text{ic ec ec} = \text{ec} \\
\text{C} : \text{Bool} \leftarrow \text{Type}
\end{array}
\]
We can extend this into an elimination rule into Kind or other higher types.

Elimination into Type (cont.)
We introduce \texttt{Bool} by simply listing its constructors (similarly to Haskell):

\[
data \texttt{Bool} = \texttt{tt} \mid \texttt{ff}
\]

\text{e.g. for defining \texttt{True} (which is used later):}

\text{\texttt{True} = \texttt{tt}}

\begin{itemize}
\item With this syntax, each constructor can occur at most once in a data type.
\item This introduces as well constants \texttt{tt} :: \texttt{Bool} and \texttt{ff} :: \texttt{Bool}.
\end{itemize}
The definition of Bool above is treated in Agda as an abbreviation for the following three more fundamental Agda definitions:

```agda
data _⊥_ : Set → Bool where
  tt : _⊥_ Bool
  ff : _⊥_ Bool

data _⊤_ : Bool → Set where
  ⊤ : _⊤_ Bool
```

The definition of Bool above is treated in Agda as an abbreviation for the following three more fundamental Agda definitions:
More about this later.

Key word depending on arguments.

This syntax is the only one allowed, if one defines a set using the data
another set with constructors tt or ff.

The definition of Bool as above doesn’t prevent the definition of
replace tt@Bool by tt.

If it is clear that the element in question is of type Bool, then one can
So tt and ff have to be defined separately.

Introduces Bool as a set having constructors tt Bool and ff Bool.

\[
\text{data } \text{tt} = \text{Set } \text{Bool} 
\]

The definition of Bool as

\[
\text{Bool in Agda (Cont.)}
\]
Internally, tt will always be represented as $\texttt{tt@Bool}$, similarly for $\texttt{f}$. So Agda evaluates tt to $\texttt{tt@Bool}$.

This can be seen when using for instance \texttt{agda-compute-WHNF}, compute weak head normal form.

Boolean in Agda (cont.)
Case Distinction

Elimination in Agda is based on case distinction.

Assume we want to define

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

so we have the goal:

- \( f \) \text{ tt } = \text{ tt } \)
- \( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)

\( f \) \text{ tt } = \text{ tt } \)

\( f \) \text{ ff } = \text{ ff } \)
Case Distinction (Cont.)

We cannot type into the goal \( x \) and choose the menu item "\texttt{aga-case}". We can then type into the goal \( x \) and choose the menu item "\texttt{aga-case}".

We then type into the goal \( x \) and choose the menu item "\texttt{aga-case}". We can then type into the goal \( x \) and choose the menu item "\texttt{aga-case}".

The goal expands to:

\[
\{ \{ i \ i \} \leftarrow (\texttt{tt}) \ \\
\{ i \ i \} \leftarrow (\texttt{ff}) \ \} \ \\
\text{case } x \text{ of} \ \\
\text{Bool} :: \\
(\text{Bool} :: x) \ f
\]

This introduces a case distinction by the constructor used for introducing \( x \) could have been introduced as \( \texttt{tt} \) or \( \texttt{ff} \).
The value of $x$ in the first goal can be tested as follows:

- Position the cursor in the first goal and choose (Goal-) menu item
- Then type into the mini-buffer $x$.
- Then $\text{compute-WHNF}$.

More precisely this means that a term is reduced until it starts with a constructor (or is a variable).

"Compute weak head normal form" means essentially "compute the result of reducing that term." "Compute the result of reducing that term." "\text{Compute-WHNF}" means essentially "\text{agda-compute-WHNF}".

- One gets the answer

The value of $x$ in the first goal can be tested as follows:
Similarly, one finds that in the second goal $x$ is $\texttt{tt}$.

\begin{itemize}
  \item
  
  \begin{align*}
  & \text{if context}\{ \text{use goal-menu "agda-context"}: } \\
  & \text{alternatively, check the cursor being in that goal, the context}
  \end{align*}
\end{itemize}
Now we can solve the new goals by inserting

\[
\begin{aligned}
\{ \text{tt}, \text{ff} \} & \leftarrow (\text{ff}) \\
\text{ff} & \leftarrow (\text{tt})
\end{aligned}
\]

of case \( x \) is the negation of \( x \): 

\[
\begin{aligned}
\text{case } x \text{ of } \ &= \\
\text{Bool} & :: (\text{Bool} :: x) \ f
\end{aligned}
\]

We obtain a function:

\[
\begin{aligned}
\text{tt} & \text{ into the second one;} \\
\text{ff} & \text{ into the first one.}
\end{aligned}
\]
Testing the Defined Function

We can test our function by using "agda-compute-WHNF".
Testing the Defined Function

The result shown is \( f \).

- and type into the mini-buffer \( f \).
- choose "\texttt{agda-compute-WHNF}".
- move to the new goal.

\[
\{ i \mid i \}\ =
\text{Set}::\text{test}
\]

- type in a dummy goal:

So we
Bool can be generalized to sets having n elements (a fixed natural number):  

**The Finite Sets**

(b) The Finite Sets
Equality Rules

The Finite Sets (Cont)
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, and write instead:

\[ \forall y \in \mathbb{C} \wedge s_0 \wedge \cdots \wedge s_{n-1} \exists y \in \mathbb{C} \wedge s_0 \wedge \cdots \wedge s_{n-1} \]

We sometimes even omit the type:

\[ \forall y \in \mathbb{C} : \forall y \in \mathbb{C} \wedge s_0 \wedge \cdots \wedge s_{n-1} \]
More compact elimination rules

(define (a C)
  (v C) <- (vmin : v)
  (w C) <- (wmin : w)
  ...
  (C : Case)

Case:
Similarly as for Bool, we can write down elimination rules, where

\[ C : \text{Type} \xrightarrow{\text{Pin}} \text{Type} \]

This can be done for all sets defined later as well.
Equality Rule

\[
\frac{C \text{ true} \land C \text{ true}}{C \text{ true} \rightarrow \text{ Set } C \text{ true}}
\]

Elimination Rule

\[
\frac{C \text{ true} \land C \text{ true}}{C \text{ true} \rightarrow \text{ Set } C \text{ true} \land C \text{ true}}
\]

Introduction Rules

\[
\text{true} \rightarrow \text{true}
\]

Formation Rule

\[
\text{true} \rightarrow \text{Set } \text{true} \land \text{true} = \text{true}
\]

Rules for True

(C) Anton Setzer 2003 (except for pictures)
Rules for True (Cont.)

- This equality is called Leibnitz equality).

true, i.e. they are identical.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.

true.
There is no Equality Rule:

\[
\frac{\text{False} \vdash \text{False}}{\text{False} \vdash \text{False}}
\]

Elimination Rule:

\[
\text{False} \vdash \text{False} \rightarrow \text{False}
\]

There is no Introduction Rule:

\[
\text{False} \vdash \text{False} 
\]

Formation Rule:

\[
\text{False} \vdash \text{False} 
\]

Rules for False:

\[
\text{False} \vdash \text{False} 
\]

False is the special case \( F^n \) for \( n = 0 \):
Everythig.

E.g. A false formula like "0 = 1" or "Swansea lies in Germany" implies absurdity. From the absurdity follows anything.

- From a logical point of view this is "Ex Falsum quodlibet" (from the false formula express any claim).

\[ \text{Case}\{\text{false}\} \text{ expresses: from an element } f \text{ of False we obtain an element of any set (which might depend on } f)\].

- As well called absurdity.

- It is formula which is always false.

- False has no elements.
Case False has no computational meaning, since there is no element it can otherwise be applied to.

- That's why it's important to carry out the termination check in Agda.

However that doesn't reduce to canonical form.

- If we had full recursion, we could define $f : \text{False} \rightarrow \text{False}$ by $f = f$.

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

- Otherwise one obtains for instance elements of False.

(C) Anton Setzer 2003 (except for pictures)
Finite sets can be introduced by giving one constructor for each element.

- And we can define, for instance:

\[
\begin{align*}
\text{is\text{-}red} \ (c \ : \ \text{Colour}) & = \ \text{case } \ c \ \text{of} \\
\quad & \quad \quad \text{Bool} \\
\quad & \quad \quad (\_ \ : \ \text{Colour}) \quad \Rightarrow \quad \text{red} \\
\end{align*}
\]

- With this we obtain \( \text{red} \ : \ \text{Colour} \)

\[
\begin{align*}
\text{data} \ \text{Colour} \ = \ \text{blue} \mid \ \text{red} \mid \ \text{green}
\end{align*}
\]
In Agda we can define the empty set as a "data-set with no constructors":

```
data False = False
```

If we want to solve

```
{ i | i } =
  Bool ::
    (False :: x) b
```

we can insert into the goal \(x\) and choose menu-item "Agda-case".

(\) Anton Setzer 2003 (except for pictures)
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:

\[
\begin{align*}
\text{IsOak} & \quad \text{oak} = \text{True} \\
\text{IsOak} & \quad \text{pine} = \text{False}
\end{align*}
\]

\[
\text{Set} \quad \text{isOak} \leftrightarrow \text{Tree} \rightarrow \text{Set}
\]

data \quad \text{Tree} = \text{pine} | \text{oak}

Below we will show how to introduce a function
If we want to define a function from trees, which are oak trees, into another set, we can do so by requiring an additional argument "IsOak". If we want to define a function from trees, which are oak trees, into another set, we can do so by requiring an additional argument "IsOak".

\[
\{ \ldots \leftarrow \text{oak} \}
\]

\[
\{ \} \leftarrow \text{pine}
\]

\[
\text{case } t \text{ of } \begin{cases}
\text{pine} & \text{case } p \text{ of } \\
\text{oak} & \forall \quad \text{IsOak } t \\
(\text{Tree } :: d) & (\text{Tree } :: t) \quad f
\end{cases}
\]
Example for the Use of False (Cont.)

In order to use $f$ we have to know that $t$ is an oak tree.

Note that we don’t have to invent a result of $f$ in case $t$ is a pine tree.

– i.e., we have to provide an argument which expresses the fact that we know this.

In order to use $f$ we have to know that $t$ is an oak tree.

Example for the Use of False (Cont.)
Similarly we can introduce a stack, together with a predicate.

\[
\text{NotEmpty} :: \text{Stack} \ni \text{NotEmpty}::d \quad \text{NotEmpty}::s \\
\text{pop} \quad \text{Stack}::s \\
\]

Again we don’t have to provide a result, in case s is empty.

\[
... = \text{Stack}::\text{NotEmpty}::s::d \\
\text{pop} \quad \text{Stack}::s::\text{NotEmpty}::\text{False} \\
\]

Now we can define

\[
\text{if } s \text{ is the empty stack:} \\
\text{NotEmpty}::s::\text{False} \\
\]

\[
\text{St.} \\
\text{NotEmpty}::\text{Stack}::\text{NotEmpty}::\text{Set} \\
\]

Example 2 for the use of False
The definition of True in Agda is straightforward:

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

Case distinction will require to solve the case \( \text{true} \):

\[
g (x :: \text{True}) :: \text{Bool} = \begin{cases} \text{case } x \text{ of } \{ \text{true} \rightarrow {!} ! \}; & \end{cases}
\]
We have already introduced two formulae:

\begin{itemize}
\item Atomic Formulae
\item The Traffic Light Example (c) Atomic Formulae and
\end{itemize}

Truth in type theory means provability.

A formula is type-theoretically true, if it is provable, i.e.:

- True is therefore type-theoretically true:

- There is a proof of it (true).

- True is inhabited.

- True.
derive \( \text{False} \) (i.e. a contradiction).

A formula is type-theoretically false, if from a proof of it we can derive everything. This is equivalent to the following:

- From this implies that we can derive \( \text{False} \) and from \( \text{False} \) we can derive everything.

Furthermore, from any proof of \( \text{False} \) we can derive everything.

Therefore, there is no proof of \( \text{False} \).

\( \text{False} \) is not inhabited.

- \( \text{False} \).
There are formulae in type theory which are neither type-theoretically true nor type-theoretically false.

This means that we can neither prove them, nor derive from a proof a

True and False as above are formulae corresponding to the truth values

- There are formulae in type theory for which neither of these two holds.
- Falsity in type theory means that we know that it cannot be true.
- Truth in type theory means that we know that it is true.
- Contradiction.

- There are formulae in type theory, which are neither type-theoretically true nor type-theoretically false.

Atomic Formulae (Cont.)
We can map truth values to their corresponding formulas:

\[
\begin{align*}
\text{atom can be defined using case distinction.} & \\
\text{atom} \quad & \text{if} \\
\text{true} & = \text{atom}
\end{align*}
\]

\[
\begin{align*}
\text{false} & = \text{atom} \\
\text{true} & = \text{atom}
\end{align*}
\]

\[
\text{Bool} \rightarrow \text{Set} = \text{atom}
\]
atom \texttt{ff} = \texttt{False}

atom \texttt{tt} = \texttt{True}

\[
\begin{align*}
\text{atom} & : \text{Set} \\
\text{atom} & \vdash \text{Bool} \\
\end{align*}
\]

This corresponds to the following rules (which are not needed)

atom (Cont.)
\[
\{ \text{False:} \leftarrow (\emptyset) \\
\text{True:} \leftarrow (\{tt\}) \} \\
\text{case } q \text{ of} \\
\text{Set ::} \\
\text{(Bool :: q) atom}
\]
Decidable Predicates

Using atom we can now define decidable predicates on sets.

Example: a means state a is safe.

- E.g. a function $f : A \rightarrow \text{Bool}$.

Example: the set of states a railway controller can choose.

- E.g. the set of states of a system $A$. Assume we have a set of states of a system $A$.

Assume we have a function $f : A \rightarrow \text{Bool}$.

Example: states a is safe.

- E.g. a means state a is safe.

Negative example: not a means state a is not safe.

- E.g. a function $g : A \rightarrow \text{Bool}$.
Decidable Predicates (Cont.)

Let now $g : A \rightarrow \text{Set}$, $g(a) = \text{atom}(a)$.

- e.g. for all $a : A$, $a$ is safe.
- e.g. for all $a : A$, $f(a)$ is true.
- For all $a : A$ we have $g(a)$ is inhabited.

Now, the existence of a $b \leftarrow (\forall : a)(\exists : y)$ means: $a \in b$.

- If $a$ is false (e.g. $a$ is unsafe), $g(a)$ is not inhabited.
- If $a$ is true (e.g. $a$ is safe), $g(a)$ is inhabited.

Let now $b : A \exists a : A$. Set, $b \leftarrow (\forall : a f(a)$.
Assume a road crossing, controlled by traffic lights:

- but A and A' always coincide, similarly B and B'.

Assume from each direction A, A', B, B' there is one traffic light.

The Traffic Light Example
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 $B'$):

$$\text{Phys-State} :: \text{Set}$$

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

\[ \text{Colour} = \text{red} \lor \text{green} \]

\[ \text{Colours} = \{ \text{red}, \text{green} \} \]

$\text{Sig}A :: \text{Colour}$

$\text{Sig}B :: \text{Colour}$
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).

- 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.
- Bred - signal B is red.

A complete set of control states consists of:

The Set of Control States
The Set of Control States (Cont.)

We therefore define

\[
\text{data Control State} = \text{Allred} | \text{Agreen} | B\text{Green}
\]
We define the state of signals A, B depending on a control state:

\[
\begin{align*}
\text{toSigA} &: \text{ControlState} \rightarrow \text{Colour} \\
\text{toSigB} &: \text{ControlState} \rightarrow \text{Colour}
\end{align*}
\]

\[
\begin{align*}
\text{toSigA} &= \begin{cases} 
\text{green} & \text{if \ Allred} \\
\text{red} & \text{if \ Agreen} \\
\text{red} & \text{if \ Bgreen}
\end{cases} \\
\text{toSigB} &= \begin{cases} 
\text{red} & \text{if \ Allred} \\
\text{red} & \text{if \ Agreen} \\
\text{green} & \text{if \ Bgreen}
\end{cases}
\end{align*}
\]
Now we can define the \textit{physical state corresponding to a control state}:

\begin{verbatim}
{ sigB = toSigB(s); sigA = toSigA(s); }

\text{Phys-State} :: \text{Phys-State}(s :: \text{Control-State})
\end{verbatim}

\textbf{Mapping Control States to Physical States}
Safety Predicate

When a physical state is safe:

\[
\begin{cases}
\text{False} & \rightarrow \text{(green)} \\
\text{True} & \rightarrow \text{(red)}
\end{cases}
\]

We define now a corresponding predicate directly, without defining first a Boolean function:

\[
\begin{cases}
\text{True} & \rightarrow \text{(green)} \\
\text{True} & \rightarrow \text{(red)}
\end{cases}
\]

- It is safe iff not both signals are green.
- We define now when a physical state is safe:

\[\text{Safety Predicate}\]
complex examples it does (due to the lack of the \( \eta \)-rule).

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

\[ \text{Cor} \cap \text{aux} \cdot \text{sig}A \cdot \text{sig}B \cdot \text{Set} :: \text{Set} \left( \text{Cor} :: \text{phys}\text{-state} \right) \]

- Now we define Safety Predicate (Cont.)
Now we show that all control states are safe:

```
{ true ← (Breen) 
  true ← (Agreen) 
  true ← (Awhile) }
```

Now we show that all control states are safe:

Safety of the System
The first element was an element of \textit{Cor\text{-}phys\text{-}state\text{-}Allred}, which reduces to \texttt{True}.

Similarly for the other two elements.

\begin{itemize}
\item This works only because each control state corresponds to a correct physical state.
\end{itemize}

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.

---

\textbf{Critical Systems, CS-4 II, Lenterm 2003, Sec. B3}

B3-54a

(C) Anton Setzer 2003 (except for pictures)
If one makes a mistake which results in an unsafe situation, the system cannot prove the correctness of the type-check.

By the termination check, (e.g., solve this goal as cor-proof A.green), but this would be rejected.

In fact we could type-theoretically solve this goal by using full recursion – then we can't solve this goal directly and cannot prove the correctness of type False.

(e.g., sets to B: A.green = green, then in the last step we obtain one goal)

If one makes a mistake which results in an unsafe situation, the system cannot prove the correctness of the type-check.
The Disjoint Union of Sets

**Introduction Rules**

\[
\begin{align*}
A & : \text{Set} \\
B & : \text{Set} \\
a & : A \\
\text{inl} & : A + B \\
b & : B \\
\text{inr} & : A + B
\end{align*}
\]

**Formation Rule**

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

**Elimination Rule**

\[
\begin{align*}
\text{split} & : A + B \to \text{Set} \\
\text{split} & : A \to \text{Set} \\
\text{split} & : B \to \text{Set}
\end{align*}
\]

\[
\begin{align*}
\text{split} & : A + B \to \text{Set} \\
\text{split} & : A \to \text{Set} \\
\text{split} & : B \to \text{Set}
\end{align*}
\]

\[
\begin{align*}
\text{split} & : A + B \to \text{Set} \\
\text{split} & : A \to \text{Set} \\
\text{split} & : B \to \text{Set}
\end{align*}
\]
The Disjoint Union of Sets (cont.)

Equality Rules

\[
\begin{align*}
\text{Plus-Split } A & \cup C \\
\text{Plus-Split } A & \cup C \\
\text{Plus-Split } A & \cup C \\
\end{align*}
\]
Disjoint Union using the Logical Framework

A more compact notation is:

\[
\begin{align*}
\text{Plus-Split:} & & \forall (B: \text{Set}) \rightarrow (B + \forall \rightarrow \text{Set}: \forall) \\
\text{inl:} & & (A;B: \text{Set}) \rightarrow A \rightarrow (A + B) \\
\text{inr:} & & (A;B: \text{Set}) \rightarrow B \rightarrow (A + B) \\
\text{Plus-Split:} & & \forall \rightarrow \text{Set}: \forall \\
\end{align*}
\]
The disjoint union can be defined as a data-set having two constructors inl (in-left) and inr (in-right):

\[
\text{data inl} \circ \text{Set} + \text{data inr} \circ \text{Set} = \text{Set}.
\]

\[
(\text{Set} : A :: B) + (\text{Set} :: B : A)
\]

Disjoint Union in Agda
Thenotation `+` means, that `+` can be used in `x`.

Now we have, if $A; B :: \text{Set}$:

\begin{align*}
\text{(inl@ (} A + B \text{)) ::} & A ! (A + B) \\
\text{(inr@ (} A + B \text{)) ::} & B ! (A + B) \\
\text{This can be checked using the menu "agda-inter-type" in a dummy goal.}
\end{align*}

\[ B + V \leftarrow B :: (B + V) \]
\[ B + V \leftarrow V :: (B + V) \]

Now we have, if $A; B :: \text{Set}$:

• The notation `(+)` means, that `+` can be used in infix.

Disjoint Union in Agda (cont.)
Elimination is again represented by case distinction.

So if we want to define, for $A : Set$ for instance $B : Set$, we can type into the goal $c$ and choose menu "agda-case".

$$
\{ i \mid i \} = \\
\text{Bool} :: \\
( B + A :: c) \mapsto f
$$
We obtain

\[ f (c :: A + B) = \begin{cases} \text{true} & \text{if } c = \text{inl } a \\ \text{false} & \text{if } c = \text{inr } b \end{cases} \]

and insert into the first goal e.g. true and the second one false

\[
\{ 1 \} \leftarrow (q \_ \text{init}) \\
\{ 1 \} \leftarrow (a \_ \text{init})
\]

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

```haskell
data Plant = tree (t :: Tree) | flower (f :: Flower)
```

Now one can define, for instance:

```haskell
isFlower (p :: Plant) :: Bool = case p of
  flower (f :: Flower) = True
  _ = False
```
The Z-Set

Formulation Rule

Introduction Rule

Elimination Rule

Sigma-Split Rule

\[ p \land \exists q : p \land q \lor \exists q : p \]

\[ \exists q : p \land q \lor \exists q : p \]

\[ q : \exists q : p \lor \exists q : p \]

\[ q : \exists q : p \lor \exists q : p \]

\[ q : \exists q : p \lor \exists q : p \]

\[ q : \exists q : p \lor \exists q : p \]

\[ q : \exists q : p \lor \exists q : p \]
The $\Sigma$-Split $A$ $B$ $C$ $s$ $(p \land B a q) = s a q (p \land B a q) \land q (p \land B a q)$

Equality Rule

The $2$-Set (cont)
The compact notation is:

\[ \exists A \subseteq B \leftarrow \\
(\forall B : \mathcal{P}) \leftarrow \\
(\forall : \forall) \leftarrow \\
(\forall : \mathcal{P}) \leftarrow \\
(\exists : \mathcal{P}) \leftarrow \\
(\exists : \mathcal{P}) \leftarrow \\

\text{The more compact notation is:}
The □-Set using the Logical Framework (cont.)
The dependent product and the \( \Sigma \)-set are very similar.

\[ p \mapsto (p \mapsto \nu \, s \mapsto p \cdot s \cdot C \cdot B \cdot \forall \, x \cdot (\forall \, y \cdot B \cdot \nu \, s \mapsto y) \cdot x) \]

\[ \forall \, x \cdot (\forall \, y \cdot B \cdot \nu \, s \mapsto y) \cdot x \]

\[ (h \cdot (x \cdot B \cdot h) \cdot x \cdot (\forall \, y \cdot B \cdot \nu \, s \mapsto y) \cdot x) \]

\[ (x \cdot (x \cdot B \cdot h) \cdot x \cdot (\forall \, y \cdot B \cdot \nu \, s \mapsto y) \cdot x) \]

On the other hand, from \( \forall \), \( \exists \) we can define \( \Sigma \)-\text{Split} as follows:

\[ \forall \, x \cdot (\forall \, y \cdot B \cdot \nu \, s \mapsto y) \cdot x \]

\[ (h \cdot (x \cdot B \cdot h) \cdot x \cdot (\forall \, y \cdot B \cdot \nu \, s \mapsto y) \cdot x) \]

\[ (x \cdot (x \cdot B \cdot h) \cdot x \cdot (\forall \, y \cdot B \cdot \nu \, s \mapsto y) \cdot x) \]
However the dependent product has the \( \Pi \)-rule (which is however not implemented in Agda).

Because of the lack of \( \Pi \)-rule, \( \exists \) works usually better than the dependent product in Agda.

Personally I don’t use the dependent product of Agda much.

The \( \exists \)-set and the Dependent Product
The $\Sigma$-Set in Agda

$\Sigma$ can be defined as a "data"-set with constructor $p$:

$\Sigma (A :: Set) (B :: A \rightarrow Set) \quad \text{data} \quad p (a :: A) (b :: B a)$
The \( \exists \)-Set in Agda (Cont.)

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

\[
\text{data Plant} = \text{plant} (g :: \text{Plant \Group}) (\text{Plants \in \Group} g)
\]

- The set of plants can then be defined as

\[
\text{Plants in that \Group.}
\]

- Depending on \( g :: \text{Plant \Group} \), sets \( \text{Plants \in \Group} g \) for \( g \) (e.g. "tree", "flower")

Example: Assume we have defined a set \( \text{Plant \Group} \) for \( \text{groups of plants} \).

\( \exists \)-Set in Agda (Cont.)
Notsurprisingly,foreliminationweusecasedistinction,e.g.:

\[
\{ \text{g} \leftarrow \text{(Plant g) \_Plant} \} \quad \begin{array}{l}
case \ d \ \text{of} \\
\text{Plant-Group} :: \\
(\text{Plant} :: d) \quad \text{f}
\end{array}
\]

Not surprisingly, for elimination we use case distinction, e.g.:

The \(\exists\)-set in Agda (cont.)
We have seen how to represent atomic decidable formulae.

Now treatment of complex formulæ constructed using logical connectives.

We have seen how to represent atomic decidable formulæ.
We can identify $A \land B$ with $A \times B$. Therefore the set of proofs of $A \land B$ is the set of pairs of elements of $A$ and $B$, i.e. $A \times B$. Therefore a proof of $A \land B$ consists of a proof of $A$ and a proof of $B$. Therefore the set of proofs of $A \land B$ is the set of pairs of elements of $A$ and $B$, i.e. $A \times B$. Therefore a proof of $A \land B$ consists of a proof of $A$ and a proof of $B$. Therefore a proof of $A \land B$ consists of a proof of $A$ and a proof of $B$. Therefore a proof of $A \land B$ consists of a proof of $A$ and a proof of $B$. Therefore a proof of $A \land B$ consists of a proof of $A$ and a proof of $B$. Therefore a proof of $A \land B$ consists of a proof of $A$ and a proof of $B$. Therefore a proof of $A \land B$ consists of a proof of $A$ and a proof of $B$. Therefore a proof of $A \land B$ consists of a proof of $A$ and a proof of $B$. Therefore a proof of $A \land B$ consists of a proof of $A$ and a proof of $B$.
With this identification, the introduction rule for $\land$:

\[
\frac{B \lor A}{B} \quad \frac{B}{A}
\]

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

\[
\frac{B \land A : \langle b, d \rangle}{B : b} \quad \frac{A : d}
\]

This is what is expressed by the ordinary introduction rule for $\lor$. 

Critical Systems, CS 411, Lenterm 2003, Sec. B3
The elimination rule for \( \land \) allows to project a proof of \( A \land B \) to a proof of \( A \) and a proof of \( B \):

\[
\begin{align*}
\frac{B}{A \land B} & \quad \frac{A}{A \land B} \\
\end{align*}
\]

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

This is what is expressed by the ordinary elimination rule for \( \lor \):

\[
\begin{align*}
\frac{B \vdash (d) \text{I}}{B \lor A \vdash d} & \quad \frac{A \vdash (d) \text{O}}{B \lor A \vdash d} \\
\end{align*}
\]

\( A \lor B \vdash \text{a proof of } B \):  

\( A \lor B \vdash \text{a proof of } A \):  

The elimination rule for \( \lor \) allows to project a proof of \( A \lor B \) to a proof of \( A \) or a proof of \( B \).
We can identify $A \land B$ with $A + B$.

Therefore the set of proofs of $A \land B$ is the disjoint union of $A$ and $B$, i.e.

\[ A + B. \]

\[ \text{proof } b : B. \]

\[ \text{It is therefore an element in } [d \text{ for a proof of } A \text{ or an element in } [b \text{ for a proof of } B. \]

plus the information which one.

Therefore a proof of $A \land B$ consists of a proof of $A$ or a proof of $B$,

\[ A \land B \text{ is true iff } A \text{ is true or } B \text{ is true.} \]
Disjunction (Cont.)

With this identification, the introduction rules for \( \land \) allows to form a proof of \( A \land B \) from a proof of \( A \) or a proof of \( B \).

Omitting the premises \( A, B \) and omitting the arguments of \( \mathbf{inl} \) and \( \mathbf{inr} \) (which is needed only for bureaucratic reasons) we get:

\[
\begin{align*}
\mathbf{inl} & : \, A + B \\
\mathbf{inr} & : \, A + B
\end{align*}
\]

\[
\begin{align*}
\mathbf{Set} & : \, B \\
\mathbf{Set} & : \, B
\end{align*}
\]
This means that we can derive $A \land B$ from $A$ and from $B$.

This is what is expressed by the ordinary introduction rules for $\land$:

\[
\frac{B \land A}{B} \quad \frac{B \land A}{A}
\]
Disjunction (Cont.)

The elimination rule for 

\[ \begin{align*}
\text{Plus-Split} & \quad \mathcal{C} : p \\
\text{sl} & \quad \mathcal{C} \leftarrow \mathcal{B} : B \\
\text{sr} & \quad \mathcal{C} \leftarrow \mathcal{A} : A \\
\end{align*} \]

allows to form from an element of \( A + B \) an element of any set \( \mathcal{C} \) provided we can compute such an element from \( A \) and \( B \).

Omitting the dependency of \( \mathcal{C} \) on \( A + B \) and omitting the bureaucratic premises and arguments \( A \), \( B \) and \( \mathcal{C} \) we get:

\[ \begin{align*}
\text{Plus-Split} & \quad \mathcal{C} : p \\
\text{sl} & \quad \mathcal{C} \leftarrow \mathcal{A} : A \\
\text{sr} & \quad \mathcal{C} \leftarrow \mathcal{B} : B
\end{align*} \]

The elimination rule for 

\[ \text{sl Set} : \mathcal{A} \\
\text{sr Set} : \mathcal{B} \]

Critical Systems, CS-411, Lentterm 2003, Sec. B3, B3-78a

\[ \text{Omitting the dependency of } \mathcal{C} \text{ on } A \_ B \text{ and omitting the bureaucratic premises and arguments } A, B \text{ and } \mathcal{C} \text{ we get:} \]

\[ \begin{align*}
\text{Plus-Split} & \quad \mathcal{C} : p \\
\text{sl} & \quad \mathcal{C} \leftarrow \mathcal{B} : B \\
\text{sr} & \quad \mathcal{C} \leftarrow \mathcal{A} : A
\end{align*} \]

\[ \text{From } B : \]

\[ \text{Omitting the dependency of } \mathcal{C} \text{ on } A \_ B \text{ and omitting the bureaucratic premises and arguments } A, B \text{ and } \mathcal{C} \text{ we get:} \]

\[ \begin{align*}
\text{Plus-Split} & \quad \mathcal{C} : p \\
\text{sl} & \quad \mathcal{C} \leftarrow \mathcal{B} : B \\
\text{sr} & \quad \mathcal{C} \leftarrow \mathcal{A} : A
\end{align*} \]

\[ \text{From } B : \]

\[ \text{Omitting the dependency of } \mathcal{C} \text{ on } A \_ B \text{ and omitting the bureaucratic premises and arguments } A, B \text{ and } \mathcal{C} \text{ we get:} \]

\[ \begin{align*}
\text{Plus-Split} & \quad \mathcal{C} : p \\
\text{sl} & \quad \mathcal{C} \leftarrow \mathcal{B} : B \\
\text{sr} & \quad \mathcal{C} \leftarrow \mathcal{A} : A
\end{align*} \]

\[ \text{From } B : \]

\[ \text{Omitting the dependency of } \mathcal{C} \text{ on } A \_ B \text{ and omitting the bureaucratic premises and arguments } A, B \text{ and } \mathcal{C} \text{ we get:} \]

\[ \begin{align*}
\text{Plus-Split} & \quad \mathcal{C} : p \\
\text{sl} & \quad \mathcal{C} \leftarrow \mathcal{B} : B \\
\text{sr} & \quad \mathcal{C} \leftarrow \mathcal{A} : A
\end{align*} \]

\[ \text{From } B : \]

\[ \text{Omitting the dependency of } \mathcal{C} \text{ on } A \_ B \text{ and omitting the bureaucratic premises and arguments } A, B \text{ and } \mathcal{C} \text{ we get:} \]

\[ \begin{align*}
\text{Plus-Split} & \quad \mathcal{C} : p \\
\text{sl} & \quad \mathcal{C} \leftarrow \mathcal{B} : B \\
\text{sr} & \quad \mathcal{C} \leftarrow \mathcal{A} : A
\end{align*} \]

\[ \text{From } B : \]
This means that we can derive from $A \land B$ a formula $C$, if we can derive $C$ from $A$ and from $B$. This is what is expressed by the ordinary elimination rules for $\lor$.

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

\[
\frac{C \quad B \quad C}{A \lor C}
\]

Similarly for $\land$, we obtain \( A \lor C \leftarrow A \lor C \) from the premise $\lor$ derivable from $B$.

$B \lor C$.
We can identity $A \implies B$ with $A \leftarrow B$.

Therefore the set of proofs of $A \implies B$ is the function type $A \leftarrow B$.

Therefore a proof of $A \implies B$ is a function which takes a proof of $A$ and computes a proof of $B$.

Therefore if there is a proof of $A$, there must be a proof of $B$.

Therefore a proof of $A \implies B$ is a function, which takes a proof of $A$ and computes a proof of $B$.

Therefore the set of proofs of $A \implies B$ is the function type $A \leftarrow B$.

We can identity $A \implies B$ with $A \leftarrow B$.

Below we see that $\subset$ can be identified with $\leftarrow$.

We write temporarily $\subset$ for logical implication, in order to distinguish it from implication.
With this identification, the introduction rule for $A$ allows to form a proof of $B$ from a proof of $A$.

\[ \frac{B \subset \forall : b \cdot (\forall : d) \forall}{B : b \leftarrow \forall : d} \]

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 $A \subset B$ without assuming $p : A$.

Implication (cont.)
This is what is expressed by the ordinary introduction rule for $\supset$:

\[
\begin{align*}
A \subseteq B \\
\therefore B \\
\therefore A
\end{align*}
\]

\( \text{Implication (cont.)} \)
The elimination rule for \( \vdash \) allows to apply a proof of \( A \rightarrow B \) to a proof of \( A \) in order to obtain a proof of \( B \). This means that we can derive from \( A \vdash B \) and \( A \vdash A \) that \( B \) holds.

This is what is expressed by the ordinary elimination rule for \( \vdash \):

\[
\frac{A \vdash B}{A \vdash B \vdash A}
\]

\[
\frac{A \vdash B}{A \vdash B \vdash A \vdash d}
\]

\[
\frac{A \vdash b \vdash d}{A \vdash b \vdash d \vdash A \vdash d}
\]

Implication (cont.)
Therefore we can identify $\neg A$ with $\neg \neg A = \text{False}$. 

A cannot be a proof of $A$, $A$ must be false. 

- If from any proof of $A$, we can create a proof of absurdity, then there is absurdity or the set $\bot$. 
- If there is no proof of $A$, then we can prove $A \in T$. 

(Where $\bot$ is absurdity or the set False: $\bot \in T$ has the same meaning as $A \in T$. 

$\neg A$ has the same meaning as $A \in T$. 

Negation
Since we have many types, we have to write when using quantifiers explicitly.

We can identify \( \forall x : A.B \) with \((x : A) \rightarrow B\).

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

There is the set of proofs of \( \forall x : A.B \) is the dependent function type

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

Therefore the set of proofs of \( \forall x : A.B \) is the dependent function type

which takes an \( x : A \) and computes an element of \( B\).

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

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

We write therefore \( \forall x : A.B \) there is the bound variable is ranging over:

Since we have many types, we have to write when using quantifiers explicitly.
This means that, if we, from $x:A$ can prove $B$, then we get a proof of \( \forall x: A \implies B \). From this we can deduce the introduction rule for $\forall$ allows to form a proof of $\forall x: A \implies B$ depending on an element $x: A$. 

\[
\frac{B \forall x: A \implies (\forall x: x) \forall}{B : d \iff \forall x: x}
\]
This is expressed by the ordinary introduction rule for $\forall$:

\[
\frac{\forall x : A \quad B}{B}
\]

This corresponds in type theory to the fact that $\forall x : A$ does no longer occur in the context of the conclusion.

- The conclusion will no longer depend on free variables $x$.
- Can therefore not depend on $x : A$.

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 $x$ might not occur free in any assumption of the proof.

Where

This is what is expressed by the ordinary introduction rule for $\forall$.

Universal Quantification (cont.)
The elimination rule for the dependent function type allows to apply a proof of $\forall x : A. B$ to an element of $A$ to obtain a proof of $B[x := a]$.

This means that we can derive from $\forall x : A. B$ and an element of $A$ that

$$
\frac{
[p := x]B : p \vdash d
}{
\forall : p \vdash \forall x : A \vdash B, x \vdash : d
}
$$

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

The elimination rule for the dependent function type allows to apply a proof of $\forall x : A. B$ to an element of $A$ to obtain a proof of $B[x := a]$. This means that we can derive from $\forall x : A. B$ and an element of $A$ that $B[x := a]$ holds.

Universal Quantification (cont.)
Universal Quantification (cont.)

This is what is expressed by the ordinary elimination rule for $\forall$.

Derivation:

For the simple languages used in ordinary logic, there is no need to derive that $a : A$, in more complex type theories we have to carry out this step.

$$\frac{\forall a A : B}{\forall \bar{a} B}$$
We can identify \( \exists x : A \land B \) with \( (x : A) \times B \). Therefore the set of proofs of \( \exists x : A \land B \) is the dependent product \( (x : A) \times B \).

Therefore a proof of \( \exists x : A \land B \) is a pair \( \langle a, p \rangle \) consisting of an element \( a : A \) and a proof \( p \) of \( B \). Therefore a proof of \( \exists x : A \land B \) is a pair \( \langle a, p \rangle \) consistent with an element \( a : A \) and a proof \( p \) of \( B \).

Therefore a proof of \( \exists x : A \land B \) is true iff there exists an \( a : A \) such that \( B \) is true.
With this identification, the introduction rule for $\exists$:

\[
\begin{align*}
  & \exists \forall \vdash x \exists \\
  & \frac{\exists \forall \vdash x \exists}{[\alpha =: x] \exists \vdash \alpha} \\
  & \frac{\exists \forall \vdash x \exists \vdash \alpha}{\exists \forall \vdash [\alpha =: x] \exists \vdash \alpha} \\
  & \exists \forall \vdash x \exists \vdash \exists \forall \vdash x \exists
\end{align*}
\]

This is what is expressed by the ordinary introduction rule for $\exists$.

With this identification, the introduction rule for $\exists$ allows to form a proof of $\exists$.

Existential Quantification (Cont.)
Existential Quantification (Cont.)

The elimination rule for the dependent product allows to project a proof\( p \) of \( \forall x : A : B \) to an element \( \{d\} : \forall \) of \( x \in A \) and proof \( \forall \) and \( \forall \) and root \( \forall \) and root \( \forall \) of \( x \in A \). From this we can derive a rule which is essentially that used in natural deduction (in which one doesn't have explicit proofs).

This kind of rule works only if we have explicit proofs.

- Assume:

\[ [(d) : \forall x : A : B] \]

- Then we have \( c \) [\( x := 0 \) (\( p \))]; \( y := 1 \) (\( p \))]: \( C \)

- \( \forall \) : \( \forall \) and root \( \forall \) and root \( \forall \) of \( x \in A \).
Therefore the conclusion does not depend on $x : A$ and $B$.

\[
\frac{C}{} \quad \frac{x}{\exists x : A \cdot B} \quad \frac{\cdot \ldots \cdot}{} \quad \frac{B}{\forall x : A}
\]

Rules:

Therefore the rule in natural deduction follows from the type theoretic existential quantification (cont.)
From type theoretic proofs we can directly extract a function, which computes the $y$ from the $x$.

Therefore, from a proof of

\[
\forall A : x A \vdash (x d) \nu y : y = f x
\]

for instance, if $A$ holds.

\[
\therefore \forall A : x A \vdash x \forall x : A \exists x, f x \forall (x, f) C \exists x, f x \forall (x, f) C
\]

extract a function $f : A \rightarrow B$, and we have

\[
(f, x) C \equiv \forall A : x A \vdash (x d) \nu y : y = f x
\]

From type theoretic proofs we can directly extract programs.

Constructive (or Intuitionistic) Logic

(C) Anton Setzer 2003 (except for pictures)
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. Therefore we cannot prove in type theory for a Turing machine whether it halts or not. We cannot decide the Turing Halting problem, i.e., we cannot decide whether $\forall x : \text{Turing-Machine} . (x \text{ halts} \lor \neg x \text{ halts})$. Therefore we cannot prove in type theory any function in type theory is recursive. Therefore we cannot prove in type theory $\forall x : \text{Turing-Machine} . (x \text{ halts} \lor \neg x \text{ halts})$. Therefore we cannot prove in type theory $\forall x : \text{Turing-Machine} . (x \text{ halts} \lor \neg x \text{ halts})$. Therefore we cannot prove in type theory $\forall x : \text{Turing-Machine} . (x \text{ halts} \lor \neg x \text{ halts})$. Therefore we cannot prove in type theory $\forall x : \text{Turing-Machine} . (x \text{ halts} \lor \neg x \text{ halts})$.
In classical logic we can prove the above, since we can derive $A \land \neg \neg A$ for any formula $A$. However, 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
  
  - $A \land A \not\rightarrow \neg A \not\rightarrow A$ and $\forall A \land \neg \forall A$ don't hold for all formulae $A$.

  - In type theory, this law cannot hold, unless we don't want that all programs can be evaluated.

  - In classical logic we can prove the above, since we can derive $A \land \neg \neg A$. 

In classical logic, \( \forall x : A \land B \) is equivalent to \( \forall x : A \land \neg \neg \neg B \).

\( B \land \forall x : A \land B \) is derivable.

\( \forall x : A \land B \) is derivable if we take decidable atomic formulae only and replace \( \forall A \).
We have classically:

\[ \neg A \rightarrow \neg\neg A \]

• Proof (using classical logic) of

\[ \exists x : A \rightarrow B \neg x : A \rightarrow B \]
We show intuitionistically:

\[(\forall x : A \rightarrow B) \leftrightarrow (\forall y : x \in E x : A \rightarrow B)\]

Now it follows (classically):

\[
\text{Assume } \forall x : A \rightarrow B \text{, then we had } B \text{, therefore a contradiction.}
\]

\[
\text{Assume } \forall x : x \in E (A \rightarrow B), \text{ assume } x \text{ st. } B \text{ holds.}
\]

\[
\text{If we had } B, \text{ then we had } \forall A : \forall B \rightarrow B, \text{ therefore a contradiction.}
\]

\[
\text{Therefore } \forall A : x \in E (A \rightarrow B) \rightarrow B.
\]

\[
\text{We show } (\forall x : A \rightarrow B) \leftrightarrow (\forall y : x \in E x : A \rightarrow B).
\]

Constructive logic (cont.)
Proof of $A \lor B$:

Now it follows (classically):

- Assume $A \land B$. If $A$ then $A \land B$, a contradiction with $\neg A$, similarly with $B$.
  - Assume $\neg A \lor B$. \iff $A \lor B$.
  - Assume $\neg B$, show $\neg A \lor B$. Similarly we get $\neg B$, therefore $\neg A \lor B$.

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

- Now it follows (classically):
  - Assume $A \land B$. If $A$ then $A \land B$, a contradiction with $\neg A$, similarly with $B$.
  - Assume $\neg A \lor B$. \iff $A \lor B$.
  - Assume $\neg B$, show $\neg A \lor B$. Similarly we get $\neg B$, therefore $\neg A \lor B$.

- We show intuitionistically:
  - Proof of $A \lor B$:
    \iff $A \lor B$. (\cont)
Weak disjunction and existential quantification is expressed by the formulae:

\((\vdash A \&^* B)\) and \((\vdash \exists x : A. \neg B)\).

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.
The set \( \mathbb{N} \) is the type theoretic representation of the set \( \{0, 1, 2, \ldots\} \).

- \( \mathbb{N} \) can be generated by:
  - starting with the empty set,
  - adding 0 to it, and
  - adding \( x \) whenever we have \( x \) in it to it.

The set \( \mathbb{N} \) is the set of Natural Numbers.
The Set of Natural Numbers (cont.)

Let $S$ be a type theoretic notation for the operation $x \rightarrow x + 1$.

Then the type theoretic rules are

\[
\frac{N : u S}{N : u}
\]

\[
N : 0
\]

\[
\text{Set : Set}
\]
Primitive Recursion expresses:

Assume we have

Then we can define \( f : \mathbb{N} \rightarrow \mathbb{N} \) s.t.

\[
(u \ f) \ u \ b = (u \ S) \ f \ \\
\quad u = 0 \ f
\]

and, if \( N : n \ N \) then \( N : x \ N \), \( N : u \ N \) -

Then, we have
The computation of \( f_n \) proceeds as follows:

- \( \mu f. \)
- \( \mu b. \)

The computation of \( f_n \) proceeds as follows:

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)

We assume that we have determined already how to compute \( f_n \).

- \( \mu f. \)
- \( \mu b. \)
The function \( f : \mathbb{N} \to \mathbb{N} \) with \( f(x) = 2x \) can be defined recursively by:

\[
\begin{align*}
  (x \ S) \ S &= x \ u \ b - \\
  0 &= n - \\
  \text{Therefore take in the definition above:} & \quad \bullet \\
  ((u \ f) \ S) \ S &= (u \ S) f - \\
  0 &= (0) f - \\
  \text{Recursive by:} & \quad \bullet
\end{align*}
\]

The function \( x \cdot 2 = (x)f \) with \( \mathbb{N} \leftarrow \mathbb{N} : f \) can be defined primitive.

Example
We can generalize primitive recursion as follows:

- First we can replace the range of $\mathcal{f}$ by an arbitrary set $\mathcal{C}$.
- Further, $\mathcal{C}$ can now depend on $\mathbb{N}$.
- i.e. we allow for any set $\mathcal{C}$.

We obtain the following set of rules:

\[ \mathcal{C} \leftarrow \mathbb{N} \vdash \mathcal{f} \]
Rules for the Natural Numbers

Introduction Rules

\[ 0 : N \]
\[ n : N \Rightarrow Sn : N \]

Elimination Rules

\[ (u f \ a \ C \ d) u f = (u S) f \ a \ C \ d \]
\[ \varphi = 0 f \ a \ C \ d \]

Equality Rules

\[ u \ C : u f \ a \ C \ d \]
\[ N : u \]

\[ (x S) \ C \leftarrow x \ C \leftarrow (N : x) : f \]
\[ 0 \ C : \varphi \]
\[ \text{Set} \leftarrow N : C \]

Formation Rules

\[ \frac{N : u S}{N : u} \]
\[ \frac{N : 0}{\text{Set} : N} \]
\((u \, b) \, u \, f = (u \, \Sigma) \, b\)
\(\nu = 0 \, b\)

The equality rules read:

\[ u \, \sigma \leftarrow (N : u) : u \, b \cdot (N : u) \]

which means that

\[ u \, \sigma : u \, b \]

The conclusion of the elimination rule reads:

\(f \, \sigma \leftarrow b\)

Note that if we define in the elimination rule that

\(f\, \sigma : b\)
The more compact notation is:

\[
\begin{align*}
\cdot \ u \subset & \quad (N : u) \leftarrow \\
\quad (x S) \subset & \quad x \subset (N : x) \leftarrow \\
0 \subset & \quad (\text{Set} \leftarrow N : \text{Set}) : \ p - \\
\text{N'} & \quad N \leftarrow N : S - \\
\text{N'} & \quad N : 0 - \\
\text{N'} & \quad N : \text{Set} - \\
\end{align*}
\]
Natural Numbers in Agda

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

Therefore we have

\[ (N :: u)S | Z = N \]

data \( N \) is defined using \textbf{data}.
Elimination works via case distinction in Agda.

\[
\begin{align*}
\begin{cases}
\{i \ i\} &\leftarrow (\ u \ S) \\
\{i \ i\} &\leftarrow (Z) \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{case } u \text{ of } =
\begin{align*}
V :: (N :: u) & f
\end{align*}
\end{align*}
\]

We get

\[
\begin{align*}
\{i \ i\} &\leftarrow (\ u \ S) \\
\{i \ i\} &\leftarrow (Z) \\
\end{align*}
\]

\[
\begin{align*}
\text{case } u \text{ of } =
\begin{align*}
V :: (N :: u) & f
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\{i \ i\} &\leftarrow (\ u \ S) \\
\{i \ i\} &\leftarrow (Z) \\
\end{align*}
\]

\[
\begin{align*}
\text{case } u \text{ of } =
\begin{align*}
V :: (N :: u) & f
\end{align*}
\end{align*}
\]

If we want to introduce *A possibly depending on u,*

\[
\begin{align*}
\{i \ i\} &\leftarrow (\ u \ S) \\
\{i \ i\} &\leftarrow (Z) \\
\end{align*}
\]

\[
\begin{align*}
\text{case } u \text{ of } =
\begin{align*}
V :: (N :: u) & f
\end{align*}
\end{align*}
\]

- If we want to introduce

- Elimination works via case distinction in Agda.
For solving the goals, we can now make use of \( f \).

\[\text{That will be accepted by the type checker.}\]

\[\text{\text{Elimination Rules for } \mathbb{N} \text{ in Agda (Cont.).}}\]
If `agda-check-termination` succeeds, the definition should be correct. However, if `agda-check-termination` fails, the definition might still be correct. (The lecturer hasn’t checked the algorithm.)

If `agda-check-termination` succeeds, the definition should be correct.
The following definition of the Fibonacci numbers can’t be defined directly using the rules of type theory, but it can be defined in Agda as follows and Agda-check-termination accepts it:

\[
\begin{align*}
    \text{one} & \leftarrow (Z) \\
    n \cdot (\text{one}) & \leftarrow (\text{one}) \\
    n \cdot g & \leftarrow g
\end{align*}
\]

\[
\begin{align*}
    b(n) & \leftarrow \begin{cases} \\
    \text{one} & \text{if } n = 0 \\
    case n of \begin{cases} \\
    (\text{one}) & \text{if } n = 0 \\
    case n of \begin{cases} \\
    (Z) & \text{if } n = 0 \\
    \end{cases} & \text{if } n > 0 \\
    \end{cases} & \text{if } n > 0
\end{cases}
\end{align*}
\]

Example of the Power of Termination Check

(c) Anton Setzer 2003 (except for pictures)
Example for Limitations of Termination Check

\[
\begin{align*}
\text{Assume we define the predecessor function} \\
\begin{cases}
\text{otherwise:} \\ 0 = \begin{cases}
0 & \text{if } u \\
1 - u & \text{otherwise}
\end{cases} \\
\text{pred}(u) = \begin{cases}
0 & \text{if } n = 0 \\
0 & \text{otherwise}
\end{cases}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\{ & u \leftarrow (u \text{ s}) \\
& Z \leftarrow (Z) \} \\
\text{of case } u = \\
N \; :: \\
(N :: u) & \; \text{pred}
\end{align*}
\]

(C) Anton Setzer 2003 (except for pictures)
Example for Limitations of Termination Check (Cont.)

Then the function

\[ f(n) : \mathbb{N} \rightarrow \mathbb{N} = \begin{cases} \text{Z} & \text{if } n = \text{Z} \\ f(\text{pred}(n)) & \text{if } n \neq \text{Z} \end{cases} \]

However, the \texttt{agda-check-termination} fails.

- It returns for all \( n : \mathbb{N} \) the value \( \text{Z} \).
- The function always terminates.

\[
\{ (\text{pred}(u)) f \leftarrow (u \text{ S}) \}
\]

\[
\text{Z} \leftarrow (\text{Z})
\]

\[
\text{case } u \text{ of } \\
\mathbb{N} :: \\
(\mathbb{N} :: u) f
\]

(C) Anton Setzer 2003 (except for pictures)
Because of the undecidability of the Turing halting problem, it is undecidable whether a recursively defined function terminates or not. There is no extension of $\text{agda-check-termination}$, which accepts exactly all $\text{agda}$-definable functions, which terminate for all inputs.

Limitations of the Termination Check (Cont.)
\[ I + (\_\_m + u) = (I + \_\_m) + u \]
\[ u = 0 + u \]

The definition expresses:

\[
\{ \\
\text{\underbrace{\_\_m + u}_m} \quad 
\text{\underbrace{(Z)}_s} \\
\text{\underbrace{\_\_u}_u} \quad 
\text{\underbrace{(N :: m, u)}_{\text{case } m \text{ of }}} \\
\text{\underbrace{N ::} + } \\
\text{\underbrace{(N :: m, u)}_{\text{definition of } + \text{ in Agda}}} \\
\text{\underbrace{\text{Example: Addition}}} \\
\}
\]
Example: Addition

Addition is used in infix, i.e., we write $u + m$ for $(+) u m$.

If $m = m' 0$, then the definition of $(+) u m$ refers to $(+) u m'$ since $m'$ is introduced before $m$.

Note that $(+) u m$ is defined before $(+) u m'$.
Example: Multiplication

\[
\begin{align*}
   u + (\cdot m \cdot u) &= (1 + \cdot m) \cdot u \\
   0 &= 0 \cdot u
\end{align*}
\]

The definition expresses:

\[
\left\{ \begin{array}{c}
   u + \cdot m \cdot u \\
   Z
\end{array} \right. \left\{ \begin{array}{c}
   (\cdot m \cdot 0) \\
   (Z)
\end{array} \right. \\
\text{case } m \text{ of } N \text{ :: } (N :: u, m) \quad (*)
\]

Definition
Note that the definition of $\ast$ requires that $\ast$ is already defined.

$\neg \; n \ast m \; \ast n \; \ast$ treated as $(n \ast m) \ast n$.

Agda has built in that $\ast$ binds more than $+$.

Agda is treated infix.

Example: Multiplication (Cont.)
The equality \( n = m \) \( \in \mathbb{N} \) can be defined using the equations:

1. \( (Z = Z) = True \)
2. \( (Z = S \ n) = (n = Z) \)
3. \( (S \ n = S \ m) = (n = m) \)

**Critical Systems, CS** 411, Lenterm 2003, Sec. B3

(C) Anton Setzer 2003 (except for pictures)
From this one can now derive a definition in Agda:

\[
\begin{align*}
\forall \{i \in \mathbb{N} \mid m \neq 0\} & \quad \leftarrow (\forall m \quad \{i \in \mathbb{N} \mid m = 0\} \quad \leftarrow (\forall m \\
\forall \{i \in \mathbb{N} \mid m \neq 0\} & \quad \leftarrow (\forall m \quad \{i \in \mathbb{N} \mid m = 0\} \quad \leftarrow (\forall m \\
\forall \{i \in \mathbb{N} \mid m \neq 0\} & \quad \leftarrow (\forall m \quad \{i \in \mathbb{N} \mid m = 0\} \quad \leftarrow (\forall m
\end{align*}
\]

Equality on \( \mathbb{N} \) (Cont.)
Reexivity of ==

Type theoretically this means that we have to define a function `ref`:

\[ \{(i, i) \} = u == u :: (N : u) \text{ ref} \]

Reflectivity of == is the formula:

\[ u == u . N : u \]

Reflectivity of ==
This can now be shown using case distinction:

\[
\begin{align*}
\{\{i \ i\}\} & \leftarrow (\mu S) \\
\{i \ i\} & \leftarrow (Z) \\
\text{case } u \text{ of } & \\
\quad u == u : & \\
\quad (N : u) \quad \text{refl}
\end{align*}
\]
Task of Coursework 3, Question 1 (e) to solve this goal.

Case $n = 5^1$ can be solved using $\text{refl}_n$ (which is defined before $\text{refl}_n$).

Case $n = Z$ is trivial.

Reflexivity of $== (\text{cont.})$
Symmetry of \[\{i \ i\}\] is the formula:

\[u == m :: (w == u :: d) \land \text{sym} \]

Typically this means that we have to define a function \(\text{sym}\):

\[u == w \leftarrow w == u \land \text{sym} \]

The symmetry of \(==\) is the formula:

\[\text{symmetry of} ==\]
This can now be shown using case distinction:

\[ \text{symmetry of } \texttt{==} \text{ (cont.)} \]
Similarly the third goal can be solved.

and have solved the second goal.

\[
\begin{cases}
\text{case } p \text{ of } d & \text{get.}
\end{cases}
\]

Therefore if we make case distinction on \( p \) we get

For the second goal we know \( p \) is an element of \( Z \), which is False.

The first goal can be solved by using true since \( Z = \{ \) True, False... \( \} \) is True.

\( \text{Symmetry of } == \) (Cont.).
In the fourth goal, we have a type of goal $S\ n' = S\ m'$ which is identical to $m' = n'$.

- The type of $d$ is $S\ n'$ which is identical to $n' = m'$.
- The goal can be solved by using $S\ n' = m'$.
- Therefore $S\ n'$ can be defined before $S\ m'$.

Note that we can use here $p$ since it is of type $n' = m'$.

The goal will be accepted by agda-check-termination.

- It is correct to use it since $n'$ is introduced before $n$.
- Therefore $S\ n'$ can be defined before $S\ m'$.

This definition will be accepted by agda-check-termination.
Example: Tuples (or Vectors) of Length n

```latex
\begin{align*}
(B :: q)(A :: n) =
\text{data cons} & \in \text{Set} \\
\text{cons} (A :: B) & \in \text{Set} \\
\text{Nil} & \text{data}
\end{align*}
```

- Define first

(C) Anton Setzer 2003 (except for pictures)
Example: Tuples (or Vectors) of Length n

Now we can define (we use Vec for vector)

\[
\begin{align*}
\text{Cons } A \text{ Vec } A \text{ m} & \leftarrow (m) \\
\text{Nil} & \leftarrow (Z) \\
\text{Set } \cdot \text{ of } u & = \\
\text{Set } \left( N : u \right) & \left( N : \text{Set } A \right) \\
\text{Vec } & \\
\end{align*}
\]
Therefore (with the obvious definition of 

\( C_n \times \cdot \cdot \cdot \times C_n \)),

\[ \text{Vec } A^n = \text{Cons } A \cdot \cdot \cdot \cdot \text{Cons } A \cdot \cdot \cdot \cdot \\text{Nil} \].

The elements of \( \text{Vec } A^n \) are

\( a_1; \cdot \cdot \cdot ; a_n \) for such an element.

In ordinary mathematical notation, we would write \( \langle a_1; \cdot \cdot \cdot ; a_n \rangle \).
Remark on Tuples of Length $n$

In ordinary mathematics, we would define

\[
\begin{align*}
\{ v \in (1 + u)^n \mid \langle 1 + u, \ldots, 1, v \rangle \} &= (1 + u)^n \\
\{ \langle \rangle \} &= (0^n) \\
\end{align*}
\]
If we define

\[
\{ (q, a) \mid \forall q \in A \forall a \in V \} := (\forall A', V', (\forall V) A')
\]

\[
\\text{Vec}(A', V', V) := (0, V)
\]

then this reads:

\[
\langle 1 + u_d, \ldots, 1, 1 \rangle := (\langle 1 + u_d, \ldots, 1, 1 \rangle, 1) \text{cons}
\]

\[
\langle \rangle := \text{nil}
\]
In the type theoretic definition we have constructors.

\[ \text{nil} :: \text{Vec} A Z \]

\[ \text{cons} \circ (\text{Vec} A (S n)) :: (u \text{ Vec} A V) \leftarrow u \text{ Vec} A V \leftarrow V : (u \text{ Vec} A V) \leftarrow Z: \text{Vec} A V \]

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

Define \( N \text{Vec}_n \) as the set of tuples of natural numbers of length \( n \).

\[
N \text{Vec}_n = \text{Set} :: (\mathbb{N} :: n) \quad N \text{Vec}_n
\]
We define component-wise sum of tuples of length \( n \).

Using mathematical notation, this sum for instance as follows:

\[
\langle 2, 3, 4 \rangle + \langle 5, 6, 7 \rangle = \langle 7, 9, 11 \rangle.
\]
Example: Componentwise Sum of Tuples of Length \( n \) (cont.)

\[
\text{SumNVec } n \langle \text{avec } b\text{vec} \rangle = \begin{cases} 
\text{Nil} & \text{if } n = 0 \\
\text{cons} \left( \text{cons } q \text{vec} \right) & \text{otherwise}
\end{cases}
\]

\[
\text{case } q\text{vec} \text{ of} \\
\text{cons } a\text{vec} \text{ of} \\
\text{cons } b\text{vec} \text{ of} \\
\text{Nil} \\
\text{cons } (\text{sum } \text{vec } n \langle \text{avec } 0 \text{vec} \rangle) \\
\text{vec } n \langle \text{avec } 0 \rangle \\
\text{vec } (n :: u) \\
\text{SumNVec }
\]

(Except for Pictures)
We define the set of lists of type \( A \) in \( \text{Agda} \).

We have two constructors:

- \text{\texttt{nil}}, generating the empty list.
- \text{\texttt{cons}}(\( a \)), adding an element of \( A \) in front of a list.

So we define lists as:

\[
\begin{align*}
\text{list} : & \ \text{Set} \\
\text{cons}(a) \ (\text{\texttt{Set} ::} \text{\texttt{Set}}) : & \text{\texttt{Set}} \\
\text{data nil} & \end{align*}
\]
and in the second goal we can make use of $f$:

\[
\{ i \mid i \} \leftarrow (\text{cons } a \mid l) \\
\{ i \mid i \} \leftarrow (\text{null} l) \}
\]

Then we can define

\[
C :: \text{Set} :: l :: \text{List} \cdot A.
\]

Assume

\[
\text{Elimination rule uses List-recursion:}
\]
Example: Length of a list

```latex
\{ \text{s :: listN} \}
Z \leftarrow \text{cons a \text{ length l}} \text{ nil} \}
\text{length (l :: list :: l) = case l of}
\text{N :: length (l :: list :: l)}
```

{Example: Length of a list}
Example: Sum of the Elements of a List

\[
\begin{align*}
\{ & u + \text{sumlist} \ \text{of} \ \text{list} \ \text{nil} \} \\
Z & \leftarrow \begin{cases}
\text{nil} & \text{of} \ N \ \text{::} \\
\text{cons} \ u & \text{of} \ \text{list} \ \text{::} \ l
\end{cases}
\end{align*}
\]
Let’s define \( \text{append} \):\( (A : \text{Set}) \to (\text{list } A) \to (\text{list } A) \).\text{ s.t.} \text{append } A \, l \, l’ \text{ is the result of appending the list } l’ \text{ at the end of list } l. \text{ and if we define } \text{cons} : = \text{cons } (\text{list } A) \text{ nil } = \text{nil } (\text{list } A), \text{ then:} \)

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

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

Define \text{Interesting Exercise}
A universe $U$ is a set, the elements of which are codes for sets.

- the dependent function type.
- $\exists$
- $+$
- $N$
- $\mathbb{P}$, $\mathbb{P}_0$, $\mathbb{P}_1$, $\text{Bool}$

We consider in the following a universe closed under

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

So we have

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

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

Introduction and Equality Rules

\[ \text{Fin}_0 : \text{UT}(\text{Fin}_0) = \text{Fin}_0 : \text{Set} \]
\[ \text{Fin}_1 : \text{UT}(\text{Fin}_1) = \text{Fin}_1 : \text{Set} \]
\[ \text{Bool} : \text{UT}(\text{Bool}) = \text{Bool} : \text{Set} \]

Rules for the Universe

Formulation Rule

(\text{Anton Setzer, 2003 (except for pictures)})
\[
\begin{align*}
\text{Set} : ((x \ q) \bot x) \ (n) \bot \exists x \ &= \ ((q \ n) \exists) \bot \\
\frac{\exists : (q \ n) \exists}{\exists \leftarrow (n) \bot : q} &\quad \frac{\exists : q}{\exists : n}
\end{align*}
\]

\[
\begin{align*}
\text{Set} : (q) \bot + (n) \bot &\ = (q + n) \bot \\
\frac{\exists : q + n}{\exists : q} &\quad \frac{\exists : n}{\exists : n}
\end{align*}
\]

for the Universe (Cont.)

Introduction/Equality Rules
\[
\text{set} : (x \, q) \perp \leftarrow ((v) \perp : x) = ((q', v) \perp) \perp \\
\end{array} \]

\[
\begin{array}{c}
\exists : (q', v) \perp \\
\hline
\exists \leftarrow (v) \perp : q \quad \exists : q
\end{array}
\]

for the Universe (Cont.)

Introduction/Equality Rules
There exist as well elimination rules and corresponding equality rules for the universe. These rules follow the principles present in previous rules. They are very long (one step for each constructor of $\cup$) and are not very much used.
Applications of the Universe

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

- Therefore, one can eliminate into the universe instead of $\text{Set}$ and use $T$ to obtain

  i.e., there are codes in the universe representing them.

  universe

  However, often, one can verify that all sets needed are elements of a

  Ordinary elimination rules don't allow to eliminate into $\text{Set}$. 

  Critical Systems, CS-411, Lenterm 2003, Sec. B3

B3-144
Example: Define

\[
\begin{align*}
\text{atom} & \equiv \text{Fin}_0, \\
\text{atom} \; \text{tt} & \equiv \text{Fin}_1,
\end{align*}
\]

Then

\[
\begin{align*}
\forall x : \text{Bool}, \; \text{T} (\text{atom} \; x) & : \text{atom} \\
\text{atom} \leftarrow \text{Set} & : \text{atom} \\
\text{Bool} \leftarrow \text{case Bool} (\forall x : \text{Bool}, \; \text{U} (\text{atom} \; x)) & : \text{atom} \\
\text{atom} \leftarrow \text{U} (\text{atom} \; \text{Bool}) & : \text{atom}
\end{align*}
\]
Universes in Agda

- Scope determined by indentation.
- Everything in the scope of it is type checked simultaneously.
- Special construct `mutual`.
- Definition possible.
- Usually Agda type checks definitions in sequence, so no reference to later
- \( \text{and } \text{I need to be defined simultaneously.} \)
Universes in 

\[
\begin{align*}
&\bigcup a \leftarrow \bigcap a \downarrow \bigcup a \downarrow \bigcap a \\
&\bigcup a \leftarrow \bigcap a \downarrow \bigcup a \downarrow \bigcap a \\
&\bigcup a \leftarrow \bigcap a \downarrow \bigcup a \downarrow \bigcap a \\
&\bigcup a \leftarrow \bigcap a \downarrow \bigcup a \downarrow \bigcap a
\end{align*}
\]

...
In the following is to be intended the same as $\bigcup$.

Universes in Agda (cont.)
Algebraic Data Types.

In general we can define now sets having arbitrarily many constructors with arbitrarily many arguments of arbitrary types.

By type theoretic rules.

The construct "data" in Agda is much more powerful than what is covered.

(\text{data}\ C \mid A \in \text{Set} = \bigcup_i A_i^j_1 \times \cdots \times A_i^j_n \times C_i^m_1 \times \cdots \times C_i^m_n)
Meaning of "data"

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

\[
C \cap A \quad :: \quad A
\]

In other words, the elements of \( A \) are exactly those constructed by those constructors.
In the types $A$, we can make use of a.

The least set $A$ having a constructor

What is the least set $A$ having a constructor

Example:

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

In the types $A$, we can make use of $A$. 

Let's consider the equation $\exists A \forall A : \text{Set} = \text{dataC}(\{ \exists A \forall A \})$. 

\begin{align*}
\text{Set} & : A \\
\text{Example} & : \set{A}
\end{align*}
Strictly Positive Algebraic Data Types (cont.)

- We shouldn't make use of such definitions.

  - In fact, "agda-check-termination" issues a warning, if we define A as above.
  - If applied to the new element C@ f might not be defined.
    - Then f might no longer be a function A → A.
    * Add C@ f to A, and
    * Find a function f :: A → A, and
    * Have constructed some part of A already.

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

\[
\text{Nlist} :: \text{Set} = \text{data nil} \cup \text{cons} \circ (\text{N} \times \text{Nlist})
\]

The constructor \text{cons} refers to \text{Nlist} by \text{Nlist}, but in a positive way:

\[
\text{Nlist} :: \text{Set} = \text{Set} \cup \text{data nil} \cup \text{cons} \circ (\text{N} \times \text{Nlist})
\]

Because we can "construct" \text{Nlist}, the above is an acceptable definition.

The constructor \text{cons} of \text{Nlist}s refers to \text{Nlist}, but in a positive way:

\[
\text{Nlist} :: \text{Set} = \text{Set} \cup \text{data nil} \cup \text{cons} \circ (\text{N} \times \text{Nlist})
\]

So we can "construct" the set \text{Nlist} by \text{Nlist} is not destroyed by this addition.

If we add \text{cons} to \text{Nlist}, the reason for adding it (namely \text{Nlist}) is not destroyed by this addition. If we add \text{cons} to \text{Nlist}, then we have \text{cons} to \text{Nlist}, but in a positive way.

\[
\text{Nlist} :: \text{Set} = \text{Set} \cup \text{data nil} \cup \text{cons} \circ (\text{N} \times \text{Nlist})
\]

Because we can "construct" \text{Nlist}, the above is an acceptable definition.
algebraic data types.

The definitions of finite sets, \( A \cup B \), \( A + B \), and \( \mathbb{N} \) are strictly positive.

And if \( A \) is a strictly positive algebraic data type, then \( A \) is acceptable.

- Or if \( A \) is itself.
- Either types which don't make use of \( A \) are is a strictly positive algebraic data type, if all \( A \) are

\[
\begin{align*}
\mathcal{C}_1 : & \quad \cdots \quad \mathcal{C}_m \quad \cdots \\
\mathcal{A}_1 : & \quad \cdots \quad \mathcal{A}_m \quad \cdots \\
\mathcal{A}_n : & \quad \cdots \quad \mathcal{A}_n \quad \cdots \\
\end{align*}
\]

And if \( A \) is acceptable.

Thedeinitionsof\( \mathbb{N} \), \( A \cup B \), and \( A + B \) were strictly positive.

In general:

Strictly Positive Algebraic Data Types (cont.)

(C) Anton Setzer 2003 (except for pictures)
One further example

The set of binary trees can be defined as follows:

\[
\text{Bintree} \triangleq \text{dataleaf} \cup \text{branch (left : Bintree)} \cup \text{branch (right : Bintree)}
\]

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

```
mutual

Even :: Set = data Z (n :: Even)
  Set :: odd

Odd :: Set = data S (n :: Odd)
  (n :: odd) :: S (n :: odd)

mutual Example: the even and odd numbers:

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

Extensions Strictly Positive Algebraic Data Types,
So again 0 can be “constructed”.

The reason for adding it to 0.

Construct \( \mathbb{N} \times 0 \), if out of it, adding this new element to 0 doesn’t destroy construct \( \mathbb{N} \times 0 \).

The last definition is unproblematic, since, if we have \( \mathbb{N} \) : \( \mathbb{N} \rightarrow 0 \) and

\[
\begin{align*}
(\mathbb{N} \times 0) & \\
\text{succ} & (0:0) \\
& = \text{data leaf}
\end{align*}
\]

Example (called “Kleene’s O”)

where \( A \) is one of the types introduced simultaneously.

We can even allow \( A \uparrow = B \uparrow < A \) or even \( A \uparrow = B \downarrow = B \downarrow < A \).

Extensions

Strictly Positive Algebraic Data Types,
Functions from strictly positive data 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

\[ f(a_1 \ldots a_n) \]

we can refer only to \( f \) applied to elements used in \( C a_1 \ldots a_n \).
can make use of $g(n)$ for all $n$.

$g(\lim_{n \to \infty} f)$

by case-distinction, then the definition of $g$:

$g(n) \quad \forall n < \infty$

by case-distinction, then the definition of $g$:

In the example of $O_n$, when defining $g$:

$\forall n < \infty$

by case-distinction, then the definition of $f$:

In the example $F$ \texttt{Bintree}$[\texttt{left} \leftarrow \texttt{right}]

by case-distinction, then the definition of $f$:

$\forall n < \infty$

For instance