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 \(2\)-set.
(f) The set of natural numbers.
(g) Lists.
(h) Universes.
(i) Algebraic data types.
(a) The Set of Booleans

**Formation Rule**

```
Bool : Set
```

**Introduction Rules**

```
tt : Bool
ff : Bool
```

**Elimination Rule**

```
C : Bool → Set, \( \text{if } ic : C \text{ tt } \text{ then } ec : C \text{ ff } \text{ cond : Bool } \)
```
Equality Rules

The Set of Booleans (cont.)
• \texttt{tt}, \texttt{ff} are the \textbf{constructors} of \texttt{Bool}.

\begin{align*}
& \text{cond } \leftarrow \\
& (\text{cond} : \text{Bool}) \leftarrow \\
& \quad (\text{if } \leftarrow \\
& \quad \quad (\text{ec } \leftarrow \\
& \quad \quad \quad (\text{tt } \leftarrow \\
& \quad \quad \quad \quad (\text{CaseBool} : (C : \text{Bool} : \text{Set}) \leftarrow \\
& \text{In the above}\end{align*}

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

\begin{itemize}
  \item \texttt{cond} for "condition".
  \item \texttt{if} for "if-case".
  \item \texttt{ec} for "else-case".
  \item \texttt{tt} stands for \texttt{true}.
  \item \texttt{ff} stands for \texttt{false}.
\end{itemize}
That's why we choose the argument to eliminate from as the last one.

\[ \forall (p : \text{Bool}) \]...

So we obtain functions from \text{Bool} into other sets without having to write\[ (b : \text{Bool}) \]

\[ \text{Set} \]

Notice that we then get for \text{C} : \text{Bool} \leftarrow \text{Set}, \text{ic} : \text{C} \text{ tt}, \text{ec} : \text{C} \text{ ff} \]...
This is similar to the definition of for instance (\(+)\) in \textit{curried form} in Haskell.\footnote{Critical Systems, CS-411, Lent Term 2003, Sec. B3}

* Shorter than writing \(\lambda x.3 + x\).

\[ (+) \rightarrow 3 \rightarrow \int \rightarrow \int \rightarrow \int. \]
Remarks (cont.)

NotethatwehavethefollowingorderoftheargumentsofCaseBool:

- \{thecondition\: \text{if} \text{:::} \text{else} \text{:::} \text{if} \} is thelastone.

\text{In some sense Case Bool is a \text{if} \text{else} \text{if} \}. \text{is the}

- Finally we put the element which we are eliminating.
- Then follow the cases, one for each constructor.
- First we have the set into which we eliminate.

CriticalSystems,CS-411,Lentterm2003,Sec.B3

B3-6
- and if it is a term, in italic blue,
- is a type in boldface red,

In the following we write `Bool` if it is a type in boldface red, and if it is a term, in italic blue.

\[ \text{Example} \]

\[ \text{AND is the conjunction:} \]

\[ \forall q, c : \text{Bool}. \text{case Bool} (q, c : \text{Bool}) \text{ if } q \text{ AND } c \]

\[ \text{Correct since } \text{AND } c = c. \]

\[ \text{Correct since } \text{AND } c = c. \]

\[ \text{Correct since } \text{AND } c = c. \]

\[ \text{Correct since } \text{AND } c = c. \]
Derivation of \( \text{AND} \):

\[
\begin{align*}
\text{Set} & \leftarrow \text{Bool} : \text{Set} \\
\text{Context} & \leftarrow \text{Type} \\
\text{Set} & \leftarrow \text{Set} \\
\text{Type} & \leftarrow \text{Type} \\
\text{Set} & \leftarrow \text{Bool} \\
\text{Type} & \leftarrow \text{Bool} \\
\text{Context} & \leftarrow \text{Type} \\
\end{align*}
\]

First we derive \( q, c : \text{Bool} \rightarrow (q : \text{Bool}) \cdot (c : \text{Bool}) \cdot \text{AND} \rightarrow \text{Bool} \).
Example (Cont.)

We derive •

(C) Anton Setzer 2003 (except for pictures)
b : \text{Bool}; c : \text{Bool}

q : \text{Bool}, c : \text{Bool} \iff \forall (p : \text{Bool}. \text{Type}) (\forall q : \text{Bool} = \text{Bool} \iff \text{Bool} \iff \text{Bool})

Similar case follows

Example (cont.)
Using part of the proof above, we derive

\[ \forall \text{Bool} \cdot \text{Bool} : \text{Bool} \]

- We derive

\[ \text{tt} \]

\[ \iff \forall \text{Bool} \cdot \text{Bool} : \text{Bool} \]

- Using part of the proof above, we derive

Example (Cont.)
Wederive:

\[
\begin{array}{c}
q : \text{Bool} \\
\text{context} \\
\end{array} \iff \begin{array}{c}
c : \text{Bool} \\
\end{array}
\]

\[
\begin{array}{c}
b : \text{Bool} \\
\end{array}
\]

... 

We derive \( q : \text{Bool}, c : \text{Bool} \iff q : \text{Bool} \)

Using part of the proof above:

Example (cont.)
Finally we obtain our judgment (we stack the premises of the rule because of lack of space):
We can extend the elimination and equality rules, having as result $\text{Type}$:

**Equality Rules into Type**

\[
\text{Case Bool} \begin{array}{c}
\text{ic} \quad \text{ec} = \text{ec} \\
\text{ic} \quad \text{ec} = \text{ec} \\
\end{array} \\
\Rightarrow \text{cond} : \text{Bool} \\
\]

**Elimination Rule into Type**

\[
\text{Case Bool} \begin{array}{c}
\text{ic} \quad \text{ec} \quad \text{cond} \\
\text{ic} \quad \text{ec} \quad \text{cond} \\
\end{array} \\
\Rightarrow \text{cond} : \text{Bool} \\
\]
We can extend this into an elimination rule into Kind or other higher types.
We introduce $\texttt{Bool}$ by simply listing its constructors (similarly to Haskell):

$$
\begin{align*}
\text{data } \texttt{Bool} &= \texttt{tt} \\
&\quad \mid \texttt{ff} \\
\end{align*}
$$

Examples for defining $\texttt{True}$ (which is used later): – i.e. we cannot define a second type having constructor $\texttt{tt}$.

With this syntax, each constructor can occur at most once in a data type.

$$
\begin{align*}
\texttt{tt} :: \texttt{Bool} \\
\texttt{ff} :: \texttt{Bool} \\
\end{align*}
$$

This introduces as well constants $\texttt{tt}$, $\texttt{ff}$.
The definition of \( \text{Bool} \) above is treated in Agda as an abbreviation for the following three more fundamental Agda definitions:

\[
\begin{align*}
\text{ff} : \text{Bool} & = \text{ff} \\
\text{tt} : \text{Bool} & = \text{tt} \\
\text{data} \text{ ff} \mid \text{tt} : \text{Set} & = \text{Bool}
\end{align*}
\]
More about this later.

Keyword 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`.

\[
\begin{align*}
\text{data } & \lll
\begin{array}{l}
\text{Set} \\
\text{Bool}
\end{array}

\text{The definition of } \text{Bool as }
\end{align*}
\]
`Bool in Agda (cont.)`

Internally, it will always be represented as `tt @ Bool`, similarly for `⊥`.

- So Agda evaluates `tt` to `tt @ Bool`.
- This can be seen when using for instance `"agda-compute-WHNF"`.

compute weak head normal form.
Case Distinction

Elimination in Agda is based on case distinction.

\[
\{ i \mid i \} = \begin{cases} 
\text{Bool} & \text{::} \, f \\
\text{Bool} : x) & f 
\end{cases}
\]

So we have the goal:
- \( \text{tt} = \text{tt} \) \( \ast \)
- \( \text{tt} = \text{ff} \) \( \ast \)

Assume we want to define

- Elimination in Agda is based on case distinction.
{;{i i} ← (ff)
{;{i i} ← (tt) }
case x of =
    Bool ::
        (Bool :: x) f

The goal expands to:

• The goal could have been introduced as tt or ff.

• x could have been introduced as tt or ff.

• This introduces a case distinction by the constructor used for introducing

• This introduces a case distinction by the constructor used for introducing

• We can then type into the goal x and choose the menu item „agda-case“.

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 (goal- menu item)
- Then type into the mini-buffer $x$.
- One gets the answer
- "Compute weak head normal form" essentially means that a term is reduced until it starts with a constructor (or a variable).
- "Compute the result of reducing that term" means essentially "agda-compute-WHNF"
- "Compute the result of reducing that term"

The value of $x$ in the first goal can be tested as follows:
Alternatively, check, the cursor being in that goal, the context:

- Similarly one finds that in the second goal \( x \) is \( \text{tt} \).

\[
\text{Bool} = \text{tt} \Rightarrow
\]

It contains:

- (use goal-menu "agda-context"):

Alternatively, check, the cursor being in that goal, the context:

Case Distinction (Cont.)
Now we can solve the new goals by inserting $x$ is the negation of $x$.

We obtain a function:

- $\text{tt} \rightarrow (\text{ff})$
- $\text{ff} \rightarrow (\text{tt})$

```haskell
f (Bool :: x) 
```

We obtain a function:

- $\text{tt}$ into the second one,
- $\text{ff}$ into the first one,
We can test our function by using "\texttt{\textasciitilde agda-compute-WHNF}\texttt{.}

Testing the Defined Function
Testing the Defined Function

1. Type in a dummy goal:

```
\{i \mid i\} = \text{test}
```

2. Choose "\texttt{agda-compute-WHNF}"

3. Move to the new goal:

```
\text{test} :: \text{Set}
```

4. And type into the mini-buffer \( f \).

5. The result shown is \( \texttt{\_\_\_\_\_} \).

So we

```
\text{Testings the Defined Function}
```
(b) The Finite Sets

bool can be generalized to sets having $n$ elements ($n$ a fixed natural number):

**Formation Rule**

$\text{Fin}_n : \text{Set}$

$A^n_k : \text{Fin}_n$

**Introduction Rules**

$C : \text{Fin}_n \rightarrow \text{Set}$

$s_0 : C \ A^n_0$

$s_1 : C \ A^n_1$

$\vdots$

$s_{n-1} : C \ A^n_{n-1}$

$a : \text{Fin}_n$

**Elimination Rule**

Case $n, C s_0 \ldots s_{n-1} a : C a$
Case $s_0, s_1, \ldots, s_{n-1}$

$$\forall u \forall C : \forall s = \forall u \forall s_{n-1} \ldots \forall s_0 \forall s_1$$

$(\forall C : \forall s_0, s_1, \ldots, s_{n-1})$
Sincethepremisesoftheequalityrulecaninmostcasestypedeterminedfrom
theintroductionandeliminationrules,wewillusuallyomitthem,andevertypethe
forinstanceforthepreviousrule:

\[ \forall u \forall v : \forall s = \forall u \forall v - 1 s_0 \cdots s_{u-v} \]

Since the premises of the equality rule can in most cases be determined from

Omitting Premises in Equality Rules
More compact elimination rules

\[
\begin{align*}
\text{Case } & \left( \text{Fin } \right) \\
(0^u \text{Fin } & : \text{Set} ) \leftarrow \\
\ldots \leftarrow \\
(1^u \text{Fin } & : \text{Set} ) \\
\end{align*}
\]
Similarly as for Bool we can write down elimination rules, where

\[
C : \text{Fin} \rightarrow \text{Type} \quad \text{instead of} \quad C : \text{Fin} \rightarrow \text{Set}.\]

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

**Formation Rule**

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

**Introduction Rules**

\[ \text{true} : \text{true} \]

**Elimination Rule**

\[ \text{true} : \text{true} \]

**Equality Rule**

\[ \text{true} : \text{true} \]

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

(© Anton Setzer 2003 (except for pictures))
Case True is computationally not very interesting.

- This equality is called Leibnitz equality.
  - true, i.e. they are identical to true.
  - This means that all elements of $x : \text{type True}$ are indistinguishable from True.
  - So there is no $C : \text{True : True}$ ! Set $s.t. C \text{true} \in C$ true is inhabitable, but $C x$ is not.

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

From a logic point of view, it expresses:

$$\forall C \text{true} \in C \text{true}$$

- However, in Agda we might not be able to derive
  - $\text{CaseTrue} \in C$ the untyped function $\lambda x.C$.

- Rules for True (cont.)
Rules for False

**Formation Rule**

\[
\text{Set False} \to \exists \text{Set False}
\]

**Elimination Rule**

\[
\frac{f \in \text{Case False}}{\text{Set False}} \to \exists \text{Set False}
\]

**Introduction Rule**

There is no Equality Rule

---

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

B3-31

(C) Anton Setzer 2003 (except for pictures)
A false formula like \( 0 = 1 \) or "Swannsea lies in Germany" implies everything.

- E.g. A false formula like "0 = 1" or "Swannsea lies in Germany" always implies absurdity. (From the absurdity follows anything.)

Case False expresses: From an element \( f \) 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 be applied to. Otherwise one obtains for instance elements of False.

- That’s why it’s important to carry out the termination check in Agda.
- However that doesn’t reduce to canonical form. \[ f = f \text{ by } \text{False} \]
- If we had full recursion, we could define \( f \text{ by } \text{False} \).
- Applies of course only if we are working in a terminating type theory.
- That’s why it’s important to carry out the termination check in Agda.
Finite sets can be introduced by giving one constructor for each element.

\[ \{ \text{\&\&} \leftarrow \text{blue} \}, \{ \text{\&\&} \leftarrow \text{green} \}, \{ \text{tt} \leftarrow \text{red} \} \]

\[
\text{is-red} \text{ :: Bool} \text{ :: } \text{Colour} \text{ :: } (\text{c :: Colour}) \]

- And we can define for instance
- With this we obtain \text{red :: Colour}

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

E.g.

E.g.

Finite sets in Agda
In Agda we can define the empty set as a "data\-set\wth no constructors":
\[
\text{data False} = \text{False}
\]

If we want to solve \( g(x::\text{False})::\text{Bool} = f!!g \) we can insert into the goal \( x \) and choose menu\-item \"Agda\-case\".

\[
\begin{array}{l}
\{ i \mid i \} = \\
\text{Bool} :: \\
\text{False} :: x \\
\end{array}
\]

\( b \)
The result is $\text{False in Agda (cont.)}$.
Example for the Use of False

Assume the type of trees:

\[
\text{IsOak} \quad \begin{array}{ll}
\text{oak} & \text{False} \\
\text{pine} & \text{True}
\end{array}
\]

\[
\text{IsOak} :: \text{Tree} \rightarrow \text{Set}
\]

\[
\text{data Tree} = \text{pime} \mid \text{oak}
\]

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". Let's see this in action:

```plaintext
f (\{ pine \} of \{ oak \}) = \forall (t : IsOak t) (d : Tree t) f
```

Example for the use of False (Cont.)
In order to use \( f \), we have to know that \( t \) is a pine tree.

\[ \text{Note that we don't have to invent a result of } f \text{ in case } t \text{ is a pine tree.} \]

\[ \text{In order to use } f \text{ we have to know that } t \text{ is an oak tree.} \]

\[ \text{Example for the Use of False (Cont.)} \]

\[ \text{Because we know that } t \text{ is a pine tree, we can use } f \text{ without inventing a result for it.} \]
Similarly we can introduce a stack, together with a predicate

\textbf{Example 2 for the Use of False}

Again we don't have to provide a result, in case \( s \) is empty.

\[
\begin{align*}
\ldots &= \\text{Stack} :: \\
&\quad \text{NotEmpty} :: d \\
&\quad \text{Stack} :: s \\
pop\end{align*}
\]

Now we can define

if \( s \) is the empty stack:

\[
\text{NotEmpty} :: s = \text{False}
\]

\text{Set} : \text{NotEmpty} :: \text{Stack} :: s \leftarrow \text{Set}

\[
\text{Set} \leftarrow \text{NotEmpty} :: \text{Stack} :: s
\]
The definition of `true` in Agda is straightforward:

\[
\begin{align*}
\begin{cases}
\text{data } \text{false} = \text{true} \\
\text{case } x \text{ of } \text{true} = \text{false} \\
(\text{true} :: x) \ b
\end{cases}
\end{align*}
\]

- Case distinction will require to solve the case `true`.
- The definition of `true` in Agda is straightforward.

\[\text{true in Agda}\]
We have already introduced two formulae:

Atomic Formulae

The Traffic Light Example
A formula is type-theoretically false, if from it we can derive everything. Furthermore, from any proof of False we can derive everything.

* False is not inhabitable.
* False is therefore type-theoretically false.

(elimination rules for False).

Furthermore, this is equivalent to the following:

Since this implies that we can derive False and from False we can derive everything.

Therefore, from any proof of False we can derive everything.

A formula is type-theoretically false, if from it we can derive False (i.e. a contradiction).

False is not inhabitable.

Atomic Formulae (Cont.)
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 contradiction.

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

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

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

\[
\begin{align*}
\text{atom} & : \text{Set} \\
\text{atom} & \text{False} = \text{False} \\
\text{atom} & \text{True} = \text{True} \\
\end{align*}
\]
atom tt = True
atom False = False

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

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

atom (cont.)
\begin{align*}
\{ \text{False} \} & \leftarrow (\emptyset) \\
\{ \text{True} \} & \leftarrow (\top) \\
\text{case } q \text{ of} \\
& \quad \text{Set} :: \\
& \quad (\text{Bool} :: q) \quad \text{atom}
\end{align*}
Decidable Predicates

Using atom we can now define decidable predicates on sets.

Assume we have a function $f : A \rightarrow \text{Bool}$. E.g., $fa$ means: state $a$ is safe.

- E.g. $a \in f$ means: state $a$ is reachable.
- E.g. the set of states a railway controller can choose.
- Assume we have a set of states of a system $A$.
Decidable Predicates (Cont.)

Let now $g : A \to \text{Set}$, $ga = \text{atom}(fa)$.

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

Now, the existence of a $h : (\forall a : A) \xrightarrow{\forall} (\forall : a) : y$ means:

- $\forall a \in A : \forall : a$ is safe
- $\forall a \in A : \forall : a$ is safe
The Traffic Light Example

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

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

The set of physical states of the system is given by a pair determining the colour of A (and therefore A') and of B (and B'):
The set of control states is a set of states of the system, a controller of the system can choose.

- **Complete set of control states** consists of:
  - All red - All signals are red.
  - All green - Signal A is green, signal B is red.
  - A green - Signal A (and A') is green, signal B is red.
  - B green - Signal B is green, signal A is red.

(usually can only be achieved in small examples)

- In our example, all safe states will be captured
- Each of these states should be safe.

A complete set of control states consists of:

- In our example, all safe states will be captured
- Each of these states should be safe.

The set of control states is a set of states of the system, a controller of the system can choose.
We therefore define

\[ \text{data Control-State} = \text{Allred | Agreen | Bgreen} \]

\[ \text{(cont.)} \]
We define the state of signals $A$, $B$ depending on a control state:

\begin{align*}
\text{toSigA}(\text{ControlState}) \colon \text{Colour} &= \begin{cases} 
\text{red} & \text{Allred}
\end{cases} \\
\text{toSigB}(\text{ControlState}) \colon \text{Colour} &= \begin{cases} 
\text{red} & \text{Allred}
\end{cases}
\end{align*}
Now we can define the physical state corresponding to a control state:

\[ \text{phys-state} \{ \text{sig}B = \text{toSigB}(s); \text{sig}A = \text{toSigA}(s) \} = \text{Phys-State} :: \text{Phys-State}(s :: \text{Control-State}) \]
When a physical state is safe:

- It is safe iff not both signals are green.

- We define now when a physical state is safe:

  We define a predicate depending on two signals:

  \[
  \begin{align*}
  \text{green} & \leftarrow \text{red} \; \text{red} \; \text{green} \; \text{red} \\
  \text{true} & \leftarrow \text{green} \; \text{green} \; \text{true} \; \text{true} \\
  \end{align*}
  \]

- We first define a predicate depending on two signals:

  - Boolean function.

- We define now a corresponding predicate directly, without defining first a

  Critical Systems, CS 411, Lent Term 2003, Sec. B3

B3-52
complex examples (due to the lack of the \( \mu \)-rule). – In the current example this wouldn’t have caused problems, but in more complex examples it does (due to the lack of the \( \mu \)-rule).

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

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

\[
\text{Cor} \times \text{sig}_A \times \text{sig}_B = \text{Set} :: \text{Set} \times \text{sig}-\text{State} (\text{Cor} :: \text{Phys-State})
\]

Now we define

\textbf{Safety Predicate (cont.)}
Now we show that all control states are safe:

```plaintext
{ true \leftarrow (\text{Bgreen}) \\
\text{true} \leftarrow (\text{Agreen}) \\
\text{true} \leftarrow (\text{Bgreen}) } 
= \text{case } s \text{ of } \\
(\text{Cor(phys-state } s) :: \text{cor-proof } (s :: \text{Control-State}) }
```

Safety of the System
The first element was an element of $\text{Cor}(\text{Phys-state Allred})$, which reduces to True. Similarly for the other two elements. This works only because each control state corresponds to a correct physical state. If this hadn't been the case, we wouldn't have gotten instances where the goal to solve is False, which we can't solve.
If one makes a mistake which results in an unsafe situation

Safety of the System (Cont.)

By the termination check, (e.g. solve this goal as cor-proof Agreen), 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.

•

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

B3-55
(d) The Disjoint Union of Sets

**Introduction Rules**

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

**Formation Rule**

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

**Elimination Rule**

\[
\begin{align*}
\text{sl} : (a : A) \rightarrow C \quad & (\text{inl} \ A B a) \\
\text{sr} : (b : B) \rightarrow C \quad & (\text{inr} \ A B b)
\end{align*}
\]
Equality Rules

The Disjoint Union of Sets (cont.)

\[
\begin{align*}
\text{Split} \quad & (\text{inl} A B a) \\
\text{Split} \quad & (\text{inr} A B a) \\
\text{Plus} \quad & (\text{inl} A B a) \\
\text{Plus} \quad & (\text{inr} A B a)
\end{align*}
\]
A more compact notation is:

\[ \begin{align*}
\text{Disjoint Union using the Logical Framework} \\
\text{Plus-Split: (Set : Set)} \\
\text{Inl: (Set : Set)} \\
\text{Inr: (Set : Set)} \\
\end{align*} \]
Disjoint Union in Agda

The disjoint union can be defined as a "data"-set having two constructors:

\[
\begin{align*}
\text{data } \text{inl} & \quad \varphi \in \text{Set} \\
\text{data } \text{inr} & \quad \psi \in \text{Set}
\end{align*}
\]

\[
\text{Set} = (\text{Set} \times \text{Set}) + (\text{Set} \times \text{Set})
\]
Thenotation $(+)$ means that $+$ can be used in $x$.

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

- $(B + V) \leftarrow B :: (B + V)$
- $(B + V) \leftarrow V :: (B + V)$
- Now we have, if $A,B :: \text{Set}$:
  - The notation $(+)$ means that $+$ can be used infix.

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

So if want to define for $A', B :: Set$ for instance.

We can type into the goal $c$ and choose menu "agda-case".

\[
\{i \ i\} =
\begin{array}{l}
\text{Bool} :: \\
(B + A :: c)f
\end{array}
\]

(Cont.)
We obtain

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

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

\[
\begin{align*}
\{ i, i \} & \leftarrow (q \text{ inr}) \\
\{ i, i \} & \leftarrow (q \text{ inl}) \\
\text{case } c \text{ of} & = \\
\text{Bool} & :: \\
(B + A :: c) & \mapsto f
\end{align*}
\]
It is usually more convenient to define concrete disjoint unions directly with more intuitive names for constructors, e.g.

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

isFlower :: d : Plant  
      = case d of
          _ : Bool    
            | f : Flower
```

Now one can define for instance:

```plaintext
{ if t : (f : flower) 
      then _ : (t : tree) 
      else _ : (f : flower) 
      }
```

Use of Concrete Disjoint Sets
The \( \exists \)-Set

\[
\frac{p \land B}{B \land \exists \: p \land B \land \exists \: p}
\]

**Introduction Rule**

\[
\frac{p \land B}{B \land \exists \: p \land B \land \exists \: p}
\]

**Formation Rule**

\[
\frac{A : \exists \land B}{A : \exists \land B}
\]

**Elimination Rule**

\[
\frac{A : \exists \land B}{A : \exists \land B}
\]
The $\Sigma$-Rule

(\( p \land q \)) \land (\neg p \lor q) = \neg q \lor (p \land q)

The $\Sigma$-Rule (cont.)
The $\exists$-Set using the Logical Framework

The more compact notation is:
The $\Sigma$-Set using the Logical Framework (Cont.)

- $\Sigma$-Split

\[
\begin{array}{c}
\Sigma \left( A : \text{Set} \right) \\
\rightarrow \left( B : A \rightarrow \text{Set} \right) \\
\rightarrow \left( C : \left( \Sigma AB \right) \rightarrow \text{Set} \right) \\
\rightarrow \left( s : \left( \alpha : A, b : B a \right) \rightarrow C \left( p A B a b \right) \right) \\
\rightarrow \left( d : \Sigma AB \rightarrow C d \right).
\end{array}
\]
The dependent product and the dependent $\Sigma$-set are very similar.

Both have similar introduction rules (for the $\Sigma$-set, the constructors have additional arguments $A$, necessary for bureaucratic reasons only).

$\forall x \in B \cdot (p x \cdot (p x'' s d)) = \top$

$\forall x \in A \cdot (p x \cdot (p x'' s d)) = \bot$

On the other hand, from $\top \land \top$, we can define $\Sigma$-Split as follows:

One can define the projections $\top \land \top$, using $\Sigma$-Split:

The dependent product and the $\Sigma$-set are very similar.
However, the dependent product has the \( \text{\textit{i1}} \)-rule (which is however not implemented in Agda).

Because of the lack of \( \text{\textit{i1}} \)-rule, \[ \exists \] works usually better than the dependent product in Agda.

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

\[
\begin{align*}
\text{(data \( B \cdot q \)) (\( A \cdot v \)) \ p} &= \\
\text{Set} &:: \\
\text{(Set \( \leftarrow \ A \cdot B \))} &:: \\
\text{Set} &:: \\
\text{(A \cdot Set)}
\end{align*}
\]

\( \text{can be defined as a } \text{"data-\set with constructor } p \text{"} \) ❖
Again one usually defines concrete \( \mathbb{2} \)-sets more directly.

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

\[
\text{The set of plants can then be defined as}
\]

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

\[
\text{depending on } \; \text{Plant-Group}, \; \text{sets Plants-in-Group(} \; g \text{) for}
\]

\[
\text{a set Plant-Group for Groups of plants (e.g. "tree", "flower",)}
\]

\[
\text{Example: Assume we have defined}
\]

\[
\text{Again one usually defines concrete \( \mathbb{2} \)-sets more directly.}
\]
\[
\{ g \leftarrow (bd \, g) \, \text{plant} \} \quad \text{case of}
\]
\[
d \quad \text{plant-group} ::
\]
\[
(\text{plant} :: d) \quad f
\]

● Not surprisingly, for elimination we use case distinction, e.g.: The 2-set in Agda (cont.)
We have seen how to represent atomic decidable formulae. Now treatment of complex formulae constructed using logical connectives.
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 $B$.

It is therefore a pair consisting 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 $B$.

$A \land B$ is true iff both $A$ is true and $B$ is true.
With this identification, the introduction rule for $\land$ allows to form a proof of $A \land B$ from a proof of $A$ and a proof of $B$:

<table>
<thead>
<tr>
<th>$B \lor A$</th>
<th>$B$</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B \lor A$</td>
<td>$B$</td>
<td>$A$</td>
</tr>
</tbody>
</table>

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

This is what is expressed by the ordinary introduction rule for $\lor$.
The elimination rule for \( \land \) allows to project a proof of \( A \land B \) to a proof of \( A \) and a proof of \( B \):

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

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

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

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

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

\[
\frac{B : (d)^1 \nu}{B \lor A : \nu}
\]

\[
\frac{A : (d)^0 \nu}{B \lor A : \nu}
\]

\( \lor \) and a proof of \( B \):

\( \lor \) elimination rule allows to project a proof of \( B \lor A \) to a proof of \( A \lor B \):

\( \lor \) elimination rule
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.

$A + B$.

A proof $b : A$. It is therefore an element in $B$ if $d$ is a proof of $A$ or an element in $B$ for a proof of $A \land B$ consists of a proof of $A$ or a proof of $B$,

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

$A \land B$ is true iff $A$ is true or $B$ is true.
With this identification, the introduction rules allow to form a proof of $A \lor B$ from a proof of $A$ or from a proof of $B$.

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

\[
\frac{\text{irr}}{B : p} \quad \text{inr}
\]

\[
\frac{B + A : p}{B : p} \quad \frac{B + A : p}{A : p}
\]
This means that we can derive $A \land B$ from $A$ and from $B$.

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

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

\(\text{Disjunction (Cont.)}\)}
The elimination rule for \( \lor \) 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 \( B \):

\[
\begin{align*}
\text{Disjunction (cont.)}
\end{align*}
\]
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 $\land$.

\[ \frac{A \land B}{\therefore C} \]

Note that in the ordinary elimination rule, from the premise of "derivable from $B" we get from $A$ we obtain "derivable from $C", similarly for "derivable from $C".

\[ (B \land C) \]

Critical Systems, CS 411, Lenttern 2003, Sec. B3

B3-78b
Implication

We write temporarily for logical implication, in order to distinguish it from the function type \( \! \).

Below we see that \( \subset \) can be identified with the function type \( \! \).

\[ \text{We can identify } A \land B \text{ with } A \rightarrow B. \]

Therefore the set of proofs of \( A \subset B \) is the function type \( A \rightarrow B \).

\[ \text{Therefore a proof of } A \subset B \text{ is a function, which takes a proof of } A \text{ and computes a proof of } B. \]

\[ \text{Therefore if there is a proof of } A, \text{ there must be a proof of } B. \]

\[ \forall A : B \text{ is true iff whenever } A \text{ is true then } B \text{ is true.} \]

\[ \forall A \subset B \text{ is true if and only if } A \text{ is true.} \]

\[ \text{Below we see that } \subset \text{ can be identified with the function type } \! \text{.} \]

\[ \text{We write temporarily } \subset \text{ for logical implication, in order to distinguish it from} \]
Implication (Cont.)

Then we can derive \( A \subset B \) without assuming \( p : A \).

\[
\frac{B \subset \forall d : A \Rightarrow \forall d}{B : b \iff \forall d}
\]

This means that, if we, from assumptions \( p : A \) can prove \( B \),

\[ (\text{i.e. we can make use of a context for proving } q) \]

We, from a proof of \( B \) depending on a proof of \( A \):

With this identification, the introduction rule for \( \subset \) allows to form a proof...
This is what is expressed by the ordinary introduction rule for $\supset$.

Implication (Cont.)
The elimination rule for $\rightarrow$ allows to apply a proof of $A \rightarrow B$ to a proof of $q$ of $A$ in order to obtain a proof of $B$ of $A$ in $\mathcal{V}$.

\[
\frac{B}{\mathcal{V} \vdash B \rightarrow A}
\]

This means that $A \rightarrow B$ holds.

\[
\frac{B : b \ d}{\mathcal{V} : b} \quad \frac{B \subset \mathcal{V} \vdash q}{\mathcal{V} : q}
\]

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

The elimination rule for $\subset$ allows to apply a proof of $B$ of $A$ in $\mathcal{V}$ to a proof of $d$ of $\mathcal{V} \subset B$.

Implication (Cont.)

(C) Anton Setzer 2003 (except for pictures)
Therefore we can identify $\neg A$ with $\neg\neg A$.

- If there is no proof of $A$, then we can prove $\neg A$.
- If from any proof of $A$ we can create a proof of absurdity, then there cannot be a proof of $A$. $A$ must be false.

(Where $\bot$ is absurdity or the set $\bot$ contains $A$.

$A$ has the same meaning as $\neg \bot$.

\[ \neg A \in A \cap \top \]\n
\[ \neg A \in A \cap \top \]\n
\[ \neg A \in A \cap \top \]\n
\[ \neg A \in A \cap \top \]
Since we have many types, we have to write when using quantifiers explicitly.

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

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

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 dependent function type $(x : A) \rightarrow B$.

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

We write therefore $\forall x : A. B$, $\exists x : A. B$. We write therefore $\forall x : A. B$, $\exists x : A. B$.

The type, the bound variable is ranging over.
This means that, if we can prove $B$ from $x:A$, then we get a proof of $\forall x:A:B$ which doesn't depend on $x:A$. 

\[
\frac{B \forall x:A : d \cdot (\forall x:A) \forall x:A : d \iff \forall x:A : B}
\]

With this identification, the introduction rule for $\forall$ allows to form a proof of $\forall x:A:B$ from a proof of $B$ depending on an element $x:A$ of $\forall x:A:B$. (Cont.)
This is what is expressed by the ordinary introduction rule for $\forall x A$.

$$\frac{A \forall x B}{B}$$

• This corresponds in type theory to the fact that $A$ does no longer occur in the context of the conclusion.

- The conclusion will no longer depend on free variables $x$.

- 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 therefore not depend on $x : A$.

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

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

\[ \frac{[v =: x] B : \forall d}{\forall : d} \frac{\forall A : x \forall : d}{B \forall : x} \]

The elimination rule for the dependent function type allows to apply a proof $d$ of $\forall A : x \forall : d$ to an element $A$ to obtain a proof of $B$. In order to obtain a proof of $A$, we apply the elimination rule.
This is what is expressed by the ordinary elimination rule for $\forall$. That $a : \forall A$, in more complex type theories we have to carry out this derivation.

For the simple languages used in ordinary logic, there is no need to derive:

$$\frac{\forall : a \quad B}{\forall : a}$$
Existential Quantification

We can identify $\exists x : A \cdot B$ with $(x : A) \times B$. Therefore the set of proofs of $\exists x : A \cdot B$ is the dependent product $(x : A) \times B$.

Therefore a proof of $\exists x : A \cdot B$ is a pair $(a, p)$ consisting of an element $a : A$ and a proof $p$ of $B[a/x]$. Therefore $\exists x : A \cdot B$ is true iff there exists an $a : A$ such that $B[a/x]$ is true.

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

(C) Anton Setzer 2003 (except for pictures)
Existential Quantification (Cont.)

With this identification, the introduction rule for $\exists x$ allows to form a proof $\exists x : A : B [x := a]$ from an element $a$. This is what is expressed by the ordinary introduction rule for $\exists:

\[
\frac{B.\forall : x \in E}{[v := x]B : \forall : v} \quad \forall : v
\]

With this identification, the existential quantification (Cont.)
The elimination rule for the dependent product allows to project a proof of \( A : B \) to an element \( x \) 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):

\[
\begin{align*}
\text{Assume:} & \quad [\forall x : \text{Set} \rightarrow (d(\forall x : A ; y : B)) \vdash c] \quad [x \mapsto p] ;
y \mapsto q.
\end{align*}
\]

\[\text{Then we have } c \quad [x \mapsto 0] ;
y \mapsto 1.\]

Critical Systems, CS-411, Lenterm 2003, Sec. B3
Therefore the conclusion does not depend on $x : A$ and $B$.

\[
\begin{array}{c}
\frac{C}{\exists x : A . B} \\
\cdot \\
\cdot \\
B \\
\forall x : A
\end{array}
\]

Rules: Therefore the rule in natural deduction follows from the type theoretic existential quantification (cont.)
function, which computes the $y$ from the $x$. For instance, if $p : A$:

\[ (x,y) \in B \]

we have $\exists y : B$. Similarly, from a proof of $\forall x : A$, we have a proof that $\forall x : A, \exists y : B$ holds.

\[ f(x) \in B \]

Therefore $f : A \to B$ is a function $A \to B$, and we have

\[ (x \mapsto f(x)) \in B \]

For instance, if $f \in A \to B$, then we have

\[ f(x) \in B \]

From type theoretic proofs we can directly extract programs.

Constructive (or Intuitionistic) Logic
\begin{align*}
\forall x : \text{Turing-Machine} \ (x \text{ halts } \land \neg (x \text{ halts}))
\end{align*}

\begin{itemize}
\item Therefore we cannot prove in type theory for a Turing machine whether it halts or not.

\item We cannot decide the Turing Halting Problem, i.e. we cannot decide which of the disjuncts holds.

\item Therefore we can decide, from a proof of a disjunction, which of the disjuncts holds.

\item We can derive as well a function which depends on \( p : A \lor B \) decides whether \( p = \text{inl}(a) \) or \( p = \text{inr}(b) \).
\end{itemize}

\textbf{Constructive Logic (Cont.)}
In classical logic we can prove the above, since we can derive $\neg A \land A$ for any formula $A$.

• In classical logic we can prove the above.

• In type theory, this law cannot hold, unless we don't want that all programs (tertium non datur) for any formula $A$.

The logic of type theory is intuitionistic (constructive) logic, in which

\[ \neg A \land \neg A \Rightarrow A \]

\[ \land \Rightarrow \neg \neg A \]

\[ A \land \neg A \Rightarrow \neg \neg \neg A \]

\[ \neg \neg \neg A \Rightarrow \neg A \]

can be evaluated.

In type theory, this law cannot hold, unless we don't want that all programs (tertium non datur) for any formula $A$. 

• The logic of type theory is intuitionistic (constructive) logic, in which

\[ \neg A \land \neg A \Rightarrow A \]

\[ \land \Rightarrow \neg \neg A \]

\[ A \land \neg A \Rightarrow \neg \neg \neg A \]

\[ \neg \neg \neg A \Rightarrow \neg A \]
In classical logic, \( \forall x : A : B \) is equivalent to \( \forall \forall \exists \forall \exists \), which trivially holds, since \( \forall \forall \exists \forall \exists \) implies \( \forall \). This requires \( (\exists \forall \exists \forall \exists \) which can be shown for all formulas built from decidable atomic formulas using \( (\exists \forall \exists \forall \exists \).  

If we take decidable atomic formulas only and replace \( \forall \exists \in \exists \exists \forall \exists \forall \exists \), then all formulas provable in classical logic are \( \forall \exists \in \exists \exists \forall \exists \forall \exists \) and replace \( \forall \exists \in \exists \exists \forall \exists \forall \exists \) and replace \( \forall \exists \in \exists \exists \forall \exists \forall \exists \).
Proof (using classical logic) of

$\exists x : A \land B \leftrightarrow \forall x : A \land B$

We have classically:

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

$\leftrightarrow A \land A$
We show intuitively (cont.):

\[
\neg x : A \dashv \vdash (\forall x : A \cdot B) \leftrightarrow (\forall x : A \cdot x \in E) \neg \leftrightarrow (\forall x \cdot A \cdot B)
\]

Now it follows (classically):

• By \( \forall x : A \cdot B \) we get \( \neg B \), therefore a contradiction.

Assume \( \forall x : A \cdot B \). Assume \( x \) s.t. \( B \) holds.

Therefore \( \forall x : A \cdot B \). Show \( \neg B \).

If we had \( B \), then we had \( \forall x : A \cdot B \), contradicting \( x \in E \). Therefore \( \forall x : A \cdot x \in E \leftrightarrow A \cdot B \).

We show intuitively (cont.):
Constructive Logic (Cont.)

Proofof $A \land \neg B$:

$$(A \land \neg B) \iff (A \land \neg B) \iff (A \land \neg B)$$

Now it follows (classically):

- Assume $A \land B$. If $A$ then a contradiction with $\neg A$, similarly with $B$.
- Assume $\neg A \land \neg B$, show $\neg (A \land \neg B)$.
  - Similarly we get $\neg B$, therefore $\neg A \land \neg B$.
- Assume $(\neg A \land \neg B) \iff (A \land \neg B)$.
  - We show intuitionistically:
    - Proof of $\neg A \land \neg B$:

  Critical Systems, CS 411, Lent Term 2003, Sec. B3

(C) Anton Setzer 2003 (except for pictures)
Constructive Logic (Cont.)

Weak disjunction and existential quantification is expressed by the formulae:

\[(A \land B)\]

and:\n
\[\exists x : A \land B\]
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,
  - adding \( x \) to it whenever we have \( x \) in it, \( x + 1 \) to it.

(f) The Set of Natural Numbers
The Set of Natural Numbers (Cont.)

\[
\begin{align*}
\text{N} & : u \\
\text{N} & : S \\
\text{N} & : 0 \\
\text{N} & \in \text{Set}
\end{align*}
\]

Then the type theoretic rules are

- Let \( S \) be a type theoretic notation for the operation \( x \rightarrow x + 1 \).
Primitive Recursion expresses:

Assume we have

\[ f_0 = a \]

\[ f_{S^n} = g_n(f_n) \]

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

\[ f(0) = a \]

\[ f(S^n) = g_n(f_n) \]

\( \forall n \in \mathbb{N} \)

\( f(0) = a \cdot f \)

\( f(S^n) = g_n(f_n) \cdot f \)

\( \forall n \in \mathbb{N} \)
The computation of $f_n$ proceeds now as follows:

- Compute $n$.
- If $n = 0$, then the result is $a$.
- Otherwise $u = S(u)$.
- Compute $u$.  

We assume that we have determined already how to compute $f_n 0$.

Now $f_n$ reduces to $g_n 0 (f_n 0)$.

$g_n 0 (f_n 0)$ can be computed, since we know how to compute $u f$.

$\mu f . b$.

$\mu f . (\mu f . \mu b)$.  

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

Primitive Recursion (Cont.)
The function $f : \mathbb{N} \to \mathbb{N}$ with $f(x) = 2x$ can be defined recursively by:

1. \( f(0) = 0 \)
2. \( f(S(n)) = S(S(f(n))) \)

Therefore take in the definition above:

1. \( a = 0 \)
2. \( g(x) = S(S(f(x))) \)

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

Example
We can generalize primitive recursion as follows:

1. First, we can replace the range of \( f \) by an arbitrary set \( C \).
   
   \[ C \leftarrow N : f \]

2. Further, \( C \) can now depend on \( N \).

3. I.e., we allow for any set \( C \).

We obtain the following set of rules:

- Generalized Primitive Recursion
(u f a Đ) u f = (u S) f a Đ

\[ p = 0 f a Đ \]

**Equality Rules**

\[
\begin{align*}
\frac{u Đ : u f a Đ}{N : u} \\
(x S) Đ \leftarrow x Đ \leftarrow (N : x) : f \\
0 Đ : a \leftarrow N : Đ
\end{align*}
\]

**Elimination Rule**

\[
\frac{N : u S}{N : u} \quad \frac{N : 0}{N} \quad \frac{\text{Set}}{N} \]

**Introduction Rules**

**Formation Rule**

**Rules for the Natural Numbers**
Rules for the Natural Numbers (cont.)

\[(u \land b) \land f = (u \land S) \land b\]
\[p = 0 \land b\]

The equality rules read:

\[\cdot u \land (N : u) : u \land b \cdot (N : u)\]

which means that

\[u \land : u \land b\]

The conclusion of the elimination rule reads:

\[f \land : b \land P = b\]

Note that if we define in the elimination rule \(f \land : b \land P = b\) then
Rules for $\mathbb{N}$ using the Logical Framework

The more compact notation is:

- $N : \text{Set}$,
- $0 : N$,
- $S : N \rightarrow N$,
- $P : (C : N \rightarrow \text{Set}) \rightarrow (C_0) \uparrow$,
- $(x : N) \rightarrow C_0 x \rightarrow C(Sx) \uparrow$
- $(n : N) \rightarrow C_n$. 

Critical Systems, CS 411, Lentterm 2003, Sec. B3
Natural Numbers in Agda

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

(Unfortunately, 0 is not an acceptable name in Agda).

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

Therefore we have

\[ \text{data} \ N \text{ is defined using data:} \]

\[ \text{Natural Numbers in Agda} \]
Elimination Rules for N in Agda

Elimination works via case distinction in Agda.

\[
\begin{align*}
\{ \langle i, i \rangle \} & \leftarrow (\mu s) \\
\{ \langle i, i \rangle \} & \leftarrow (\nu z) \\
\text{case } u \text{ of } V :: (N :: u) & \to f
\end{align*}
\]

We get

We can type into the goal \( u \) and use the menu Agda-case.

\[
\begin{align*}
\{ i, i \} & = \\
V :: (N :: u) & \to f
\end{align*}
\]

- If we want to introduce
For solving the goals, we can now make use of \( f \).

- If we make use of \( f \) in the case \( n = 5 \),
- do not make use of \( f \) in the case \( n = 2 \), and

Then \texttt{agda-check-termination} succeeds.

- only use of \( f \) in the case \( n = 5 \).

However, if we use \texttt{full \_f} and then use menu item „\texttt{agda-check-termination}” we might obtain an error message.

That will be accepted by the type checker.

\textbf{Elimination Rules for \( N \) in Agda (Cont.)}
If \texttt{agda-check-termination} succeeds, the definition should be correct.

However, if \texttt{agda-check-termination} fails, the definition might still be correct.

- The lecturer hasn't checked the algorithm.

\textbf{Elimination Rules for N in \texttt{Agda} (Cont.)}
The following definition of the Fibonacci numbers can’t be defined in Agda as directly using the rules of type theory, but it can be defined in Agda as follows and Agda-check-termination accepts it:

\[
\begin{align*}
\{ \text{'} \} & \quad \text{one} \leftarrow (\text{'} n S) \\
\text{one} & \leftarrow (Z) \\
\text{one} & \leftarrow (\text{'} n S) \\
\text{one} & \leftarrow (Z) \\
\text{case } n \text{ of } \text{one} & =
\begin{array}{c}
N \quad ::
\end{array}
\text{one} \leftarrow (Z S :: u) \\
\end{align*}
\]
Example for Limitations of Termination Check

Assume we define the predecessor function

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

\[
\{ \begin{array}{c}
u' \\ Z'
\end{array} \leftarrow \begin{array}{c}
\{ \begin{array}{c}
u \\ S
\end{array} \\
\{ \begin{array}{c}
Z \\
(\bar{S})
\end{array}
\end{array}
\right.
\]

\text{case } u \text{ of } \begin{array}{c}
\begin{array}{c}
N \\
(\bar{N} :: u)
\end{array}
\end{array}
\]

\text{pred}
Example for Limitations of Termination Check (Cont.)

Then the function

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

terminates always

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

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

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

- (it returns for all \( n : \mathbb{N} \) the value \( Z \)).
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 Agda-Check-Termination, which accepts exactly all in Agda definable functions, which terminate for all inputs.
Example: Addition

Definition of $+$ in Agda:

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

The definition expresses:

\[
\begin{cases}
I + (m + u) & \leftarrow (I + m) + u \\
0 + u & \leftarrow (Z)
\end{cases}
\]

\[
\text{case } m \text{ of } \leftarrow
\begin{align*}
\mathbb{N} & \hookrightarrow \\
(N :: m, u) & \leftarrow (+)
\end{align*}
\]

Definition of $+$ in Agda:
Example: Addition

Not that \((+)\) is used in infix, i.e. we write \(u + v\) for \((+\) \(u, v)\)

If \(m = S^m\), the definition of \((+)\) refers to \((+\) \(0, m)\).

\(-\) is defined before \((+)\) in \(m\) since \(m\) is introduced before \(m\).

\(n\) in \((+)\) \(u, m\) since \(m\) is introduced before \(m\).
Example: Multiplication

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

The definition expresses:

\[
\begin{cases}
    u + m \cdot u & \leftarrow (m \cdot S) \\
    Z & \leftarrow (Z)
\end{cases}
\]

\[
\text{case } m \text{ of } \\
    N :: (N :: u, u) \ (\star)
\]
Again is treated in 

\[ n \cdot m + n \text{ is treated as } (n \cdot m) + n. \]

Agda has built in that \* binds more than +. 

\[ \text{\* is treated infix.} \]

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

\[
\begin{align*}
(Z == Z) &= True, \\
(Z == S n) &= (n == S Z) = False, \\
(S n == S m) &= (n == m).
\end{align*}
\]

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

\[
\begin{align*}
(Z == Z) &= True, \\
(Z == S n) &= (n == S Z) = False, \\
(S n == S m) &= (n == m).
\end{align*}
\]
From this one can now derive a definition in Agda:

```
{::'i i} ← (\m S)
{::'i i} ← (Z)
\case m of
{::'i i} ← (\m S)
{::'i i} ← (Z)
\case m of
\case u of
Set ::
(N :: m,u) (==)
```
Task of Coursework 3, Question 1 (e) to solve this goal:

\[
\{ i \mid i \} = u \equiv u :: (N : u) \text{ Refl}
\]

Type theoretically this means that we have to define a function \text{Ref}: 

\[
u \equiv u. N : u
\]

Reflexivity of \(
\equiv
\) is the formula: 

Reflexivity of \(\equiv\)
Reexivity of \(\mathrel{=}\) (Cont.)

This can now be shown using case distinction:

\[
\begin{aligned}
\{\{i\}i\} & \leftarrow (\mu S)
\{\{i\}i\} & \leftarrow (Z)
\{\{i\}i\} & \leftarrow \begin{cases}
\text{case } u \in & \\
\text{ of } u & = \\
\text{ u } & \Rightarrow \\
\text{ N : u) & \text{ refl}
\end{cases}
\end{aligned}
\]

Reflexivity of \(\mathrel{=}\) (Cont.)
Case $n = \mathbb{S}$ can be solved using $\mathbb{R}^n$ (which is defined before $\mathbb{R}^n$).

Case $n = \mathbb{Z}$ is trivial.

Reflexivity of $== (cont.)$
Symmetry of $==$ is the formula:

$$\{i \ i\} = u == \mu \quad ::
(\mu == u :: d)
(N : \mu, \mu) \ sym$$

Typically, this means that we have to define a function $sym$:

$$u == \mu \leftarrow \mu == u \quad \forall \mu, N : \mu, \mu$$

Symmetry of $==$ is the formula:
This can now be shown using case distinction:

\[ \text{symmetry of } \equiv (\text{cont.}) \]
The first goal can be solved by using true (since $Z == Z = True$).

Similarly, the third goal can be solved.

For the second goal we know $p$ is an element of $Z == S$ which is False.

Therefore, if we make case distinction on $p$ we get

\[
\begin{cases}
\text{case } d\text{ of }
\end{cases}
\]

and have solved the second goal.
In the fourth goal, we have a type of goal $S n \equiv S m$, which is identical to $n \equiv m$. The goal can be solved by using $S n \equiv m$.

- The type of $d$ is $S n \equiv S m$ which is identical to $n \equiv m$.
- Therefore we can use here $p$ since it is of type $n \equiv m$.

Symmetry of $== $ (cont.)
Example: Tuples of Length n

Define First

\[
(B :: q) (A :: \text{cons}(A, B :: \text{Set} :: \text{Set}) :: \text{Set}) = \text{data cons}(a :: A) (b :: B :: \text{Set})
\]

\[
\text{data nil} = \text{nil}
\]
Example: Tuples of length $n$
In ordinary mathematics, we would define

\[ \{ V \in I^{1+u_\psi}, \ldots, I^{1+u_\psi} | \langle I^{1+u_\psi}, \ldots, I^{1+u_\psi} \rangle \} =: (I + u_\psi, \text{Vec}(V), V) \]

\[ \{ \langle \rangle \} =: (0, \text{Vec}(V), V) \]

Remarks on Tuples of Length n
\[ \{ (n, u) \in V \times V \mid (q, \text{cons}(\forall, u)) =: (q, \text{vec}(\forall)) + 1 \} \]

\[ \{ \text{nil} \} =: (0, \text{vec}(\forall)) \]

then this reads:

\[ \langle 1 + u^p, \ldots, 1^p \rangle =: (\langle 1 + u^p, \ldots, 1^p \rangle, \text{cons}(\forall, 1) \rangle \]

\[ \langle \rangle =: \text{nil} \]

• If we define

**Remarks on Tuples of Length n**
In the type theoretic definition, we have constructors

\[
\begin{align*}
\text{cons} & : \text{Vec } A \times (\text{Vec } A \times A) \\
\text{nil} & : \text{Vec } A
\end{align*}
\]

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

Define $\text{NVec} \ (n :: N) \triangleq \text{Vec} \ N \ n$
Example: Component-wise sum of tuples of length \( n \).

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.)

```plaintext
\[
\sum_{i=1}^{n} (a_i + b_i)
\]
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{nil}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{nil}
```

```plaintext
\text{cons}
```

```plaintext
\text{nil}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```

```plaintext
\text{cons}
```
We define the set of lists of elements of type A in Agda.

\[
\text{list} \quad (A : \text{Set}) \quad :: \text{Set} \\
\text{nil} :: \text{List} \\
\text{cons} \quad (A :: \text{List} \quad A :: \text{Set} \quad \text{\_}) \\
\]

So we define lists as:

- **cons**, adding an element of A in front of a list
- **nil**, generating the empty list.

We have two constructors:

- **(g) Lists**
Elimination rule for lists

Elimination rule uses list-recursion:
Assume \( \{ i : i \} \) depending on \( l :: \) list
Then we can define
Assume Elimination rule uses list-recursion:
- \( C : Set \) depending on \( l :: \) list
- \( A : Set \)

\[
\begin{align*}
\{ \text{\( i \)} : \text{\( i \)} \} & \rightarrow (\text{\( \alpha \)} \text{\( l \)} \text{\( A \)} :: \text{\( l \)} \text{\( \alpha \)} \text{\( C \)} :: \text{\( l \)} \text{\( A \)} \text{\( f \)})
\end{align*}
\]
Example: Length of a list

```haskell
\{ S \leftarrow (\text{\textbf{cons}} \ a \ l) \\
Z \leftarrow (\text{\textbf{nil}}) \} \\
\text{\textbf{case}} \ l \ \text{of} \\
\text{\textbf{nil}} \Rightarrow N :: \text{\textbf{length}} (\text{\textbf{List}} N)```

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

\begin{align*}
\text{sumlist} & \colon \text{listN} \\
S & \leftarrow (\text{cons a l}) \\
Z & \leftarrow (\text{nil})
\end{align*}

\begin{align*}
\text{case } l & \text{ of} \\
N & :: (\text{sumlist list :: l})
\end{align*}
Interesting Exercise

Define append: \((A : \text{Set}) \rightarrow (\text{list } A) \rightarrow (\text{list } A) \rightarrow \text{list } A\) such that

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

and if we define cons := \text{cons } A \text{ nil}, then:

- \(\text{E.g., if } a, b, c, d \text{ are elements of } A, \)
  \(\text{append } A \text{ nil} \text{ nil} = \text{cons } A \text{ nil} \text{ nil} = \text{nil} \text{ nil}\)

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

Define append: \((A : \text{Set}) \rightarrow (\text{list } A) \rightarrow (\text{list } A) \rightarrow \text{list } A\)
A universe $U$ is a set, the elements of which are codes for sets.

We consider in the following a universe closed under

- the dependent function type.
  
  - $\exists$,
  
  - $\forall$,
  
  - $+$,
  
  - $\times$,
  
  - $\text{Fin}_0$, $\text{Fin}_1$, $\text{Bool}$,

  - $\text{Set}$ (the decoding function).

So we have

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

\[
\begin{align*}
\text{Fin}_0 : \text{Set} & = \text{Fin}_0 \\
\text{T}(	ext{Fin}_0) & = \text{Fin}_1 \\
\text{T}(	ext{Fin}_1) & = \text{Fin}_0
\end{align*}
\]

Formation Rule

Rules for the Universe
\[ \text{Set} : \left( (x \ q) \ 
 \right) \ \forall \ x (q \ x) = ((q \forall) \exists) \ \exists \]

\[ \cup : (q \forall) \exists \\
\cup \leftarrow (q) \exists : q \cup : \forall \\
\]

\[ \text{Set} : (q) \exists + (q) \exists = (q \mp q) \exists \\
\]

\[ \cup : q \mp q \\
\cup : q \cup : q \\
\]

for the Universe (Cont.)
\[ \text{Set} : (x \ q) \bot \leftarrow ((\forall) \bot : x) = ((q \forall) \bot) \bot \]

\[ \begin{align*}
\forall : (q \forall) & \bot \\
\forall & \leftarrow (\forall) \bot : q \\
\forall & : q
\end{align*} \]

for the Universe (Cont.)

Introduction/Equality Rules

(\text{C}) \ Anton Setzer 2003 (except for pictures)
There exist as well elimination rules and corresponding equality rules for the universe. They follow the principles present in previous rules. They are very long (one step for each constructor of $\cup$) and are not very much used.
Ordinary elimination rules don't allow to eliminate into $\text{Set}$. However, one can verify, that all sets needed are "elements of a universe", i.e. there are codes in the universe representing them.

Then one can eliminate into the universe instead of $\text{Set}$ and use $T$ to obtain the required function.

- However often, one can verify, that all sets needed are "elements of a universe".

Application of the Universe

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

\[
\begin{align*}
\text{atom}: \text{Bool}! U & \quad \text{atom} \not\in F_0. \\
\text{atom} = \text{Case} & \quad \text{atom} \in F_1.
\end{align*}
\]

Then

\[
\begin{align*}
\chi_{(\text{atom}: \text{Bool})} & : \text{atom} \not\in F_0, \\
\text{Set} & : \text{atom} \not\in F_0.
\end{align*}
\]

\[
\begin{align*}
\chi_{(\text{atom}: \text{Bool})} & : \text{atom} \not\in F_0, \\
\text{Bool} \cup & : \text{atom} \not\in F_0.
\end{align*}
\]

Applications of the Universe

(© Anton Setzer 2003 (except for pictures)
Universes in Agda

• $\cup$ and $\cap$ need to be defined simultaneously.

- Special construct `mutual`.
- Definitions possible.

- Usually Agda types checked definitions in sequence, so no reference to later

Everything in the scope of it is type checked simultaneously.

Scope determined by indentation.

(C) Anton Setzer 2003 (except for pictures)
Universes in Agda (Cont.)
T in the following is to be intended the same as \( U \).

Universes in Agda (cont.)
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.
In other words the elements of A are exactly those constructed by those constructors. Where a constructor always constructs new elements.

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

\[ C : A \]

\[ \vdash \exists A \text{ constructor} \]

\[ (a : A) \vdash A \]

\[ \vdash \exists a \in A \text{ constructor} \]

\[ \vdash \forall a : A \exists C(a) \]

\[ \vdash \forall a : A C(a) \]

\[ \vdash \forall a : A \exists C(a) \]

\[ \vdash \forall a : A C(a) \]

\[ \vdash \forall a : A \exists C(a) \]

\[ \vdash \forall a : A C(a) \]
In the types $A$ we can make use of $A$. However, it is difficult to understand $A$, if we have negative occurrences of $A$. Example:

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

What is the least set $A$ having a constructor $C$?

**Critical Systems, CS-411, Lentterm 2003, Sec. B3**
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.

(f 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.

* 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:

\[ Nlist :\text{Set} \]

\[ \text{cons}(a::N, l::Nlist) \]

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

\[ (1 \in Nlist) \]

\[ \text{cons}(a::N, l::Nlist) = \text{data nil} \]

\[ Nlist :\text{Set} \]

Because we can "construct" \( N \)-lists, the above is an acceptable definition.

- Closing it under \text{cons} whenever possible.
- Adding \text{nil} and \text{cons}.
- Starting with the emptyset.

So we can "construct" the set \( N \)-list by \( Nlist \) is not destroyed by this addition.

- If we add \text{cons} to \( N \), the reason for adding it (namely 1) is:

\[ Nlist \]

\[ \text{cons}(a::N, l::Nlist) \]

We have: if \( a \in N \) and \( l \in Nlist \), then we have \text{cons}(a::N, l::Nlist).

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

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

The definitions of finite sets, \( \mathbb{N} \), and \( \mathbb{N} \) were strictly positive.

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

- \( A \) is acceptable itself.
- Either types which don't make use of \( A \) are.

In general:

\[
\begin{align*}
C & : : \text{data} \left( a_{11} \ : \ A_{11} \right) \left( a_{12} \ : \ A_{12} \right) \left( a_{13} \ : \ A_{13} \right) \cdots \\
C & : : \text{data} \left( a_{21} \ : \ A_{21} \right) \left( a_{22} \ : \ A_{22} \right) \left( a_{23} \ : \ A_{23} \right) \cdots \\
\vdots & \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\
C & : : \text{data} \left( a_{m1} \ : \ A_{m1} \right) \left( a_{m2} \ : \ A_{m2} \right) \left( a_{m3} \ : \ A_{m3} \right) \cdots \\
\end{align*}
\]
One further example

This is a strictly positive data type.

```
Bintree::Set = data leaf
  branch (left::Bintree)
    right :: Bintree
```

The set of binary trees can be defined as follows:

```
Bintree :: Set = data leaf
```
An often used extension is to define several sets simultaneously inductively.

Example: the even and odd numbers:

\[
\begin{align*}
\text{Even} &::= \text{Set} \\
\text{Odd} &::= \text{Set}
\end{align*}
\]

\[
\begin{align*}
\text{data } S \ (n::\text{Even}) & = \\
\text{odd } ::\text{ Set} &
\end{align*}
\]

\[
\begin{align*}
\text{data } S \ (n::\text{Odd}) & = \\
\text{Even } ::\text{ Set} & \quad \text{mutual}
\end{align*}
\]

In such examples the constructors refer strictly positive to all sets which are to be defined simultaneously.
So again 0 can be „constructed“.

The reason for adding it to 0.

Construct \( \mathbb{N} \rightarrow 0 \). If out of it, adding this new element to 0 doesn’t destroy the last definition is unproblematic, since, if we have \( i : \mathbb{N} \rightarrow 0 \) and

\[
\begin{align*}
(0 & \rightarrow 0) \\
\text{true (false)} & | \\
\text{succ} (0:0) & = \text{data Leaf} \\
0 : : \text{Set} & = 0
\end{align*}
\]

Example (called „Kleene’s 0“)

\[
\begin{align*}
\text{where } A \text{ is one of the types introduced simultaneously:}
\end{align*}
\]

\[
\begin{align*}
\text{We can even allow } A \uparrow = B_1 \prec A \text{ or even } A \downarrow = B_1 \prec \cdots \prec B_1 \prec A
\end{align*}
\]
Elimination Rules for data

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 $a_1 \ldots a_n$, we can refer only to $f$ applied to elements used in $\mathcal{C}_0^\mathbb{N}$. 

$\quad \vdots$
Examples

For instance, in the Bintree example, when defining
\[ f \in \text{Bintree} \] 
by case-distinction, then the definition of

\[ f'(\text{branch} \ast \text{left right}) \]

by case-distinction, then the definition of

\[ f : \text{Bintree} \to \mathbb{N} \]

- in the example of \( O \), when defining

\[ g \in O \]

by case-distinction, then the definition of

\[ g(n : \mathbb{N}) \]

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

\[ g(\lim_{n : \mathbb{N}} f) \]

- in the Bintree example, when defining

\[ f \in \text{Bintree} \]

For instance