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

**Formation Rule**

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

**Introduction Rules**

\[ \text{tt} : \text{Bool} \quad \text{ff} : \text{Bool} \]

**Elimination Rule**

\[
\begin{align*}
C : \text{Bool} & \to \text{Set} \\
\text{ic} : C \text{ tt} & \quad \text{ec} : C \text{ ff} & \quad \text{cond} \\
\text{Case}_{\text{Bool}} C \text{ ic ec cond} & : C' \text{ cond}
\end{align*}
\]
The Set of Booleans (Cont.)

Equality Rules

\[ C : \text{Bool} \rightarrow \text{Set} \quad \text{ic} : C \ tt \quad \text{ec} : C \ ff \]
\[ \text{Case}_{\text{Bool}} C \ ic \ ec \ tt = ic : C \ tt \]
\[ C : \text{Bool} \rightarrow \text{Set} \quad \text{ic} : C \ tt \quad \text{ec} : C \ ff \]
\[ \text{Case}_{\text{Bool}} C \ ic \ ec \ ff = ec : C \ ff \]
Remarks

• In the above
  – \texttt{tt} stands for true, \texttt{ff} stands for false.
  – \texttt{ic} stands for “if-case”, \texttt{ec} for “else-case”.
  – \texttt{con} for “condition”.

• We can write the elimination rule in a \textbf{more compact} but less readable way:

  – \texttt{Case}_{\text{Bool}} : (C : \text{Bool} \to \text{Set}) \to (ic : C \texttt{tt}) \to (ec : C \texttt{ff}) \to (cond : \text{Bool}) \to C \text{ cond}

• \texttt{tt}, \texttt{ff} are the \textbf{constructors} of \texttt{Bool}.
• Notice that we then get for $C : \text{Bool} \rightarrow \text{Set}$, $ic : C \ tt$, $ec : C Fedicec$:

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

$$f \ tt = \text{Case}_{\text{Bool}} C ic ec \ tt = ic : C \ tt,$$

$$f \ ff = \text{Case}_{\text{Bool}} C ic ec \ ff = ec : C \ ff.$$

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

• That’s why we choose the argument to eliminate from as the last one.
• This is similar to the definition of for instance (+) in curried form:

- \((+) : \text{int} \rightarrow \text{int} \rightarrow \text{int}\).
- \((+) 3\) is the function which takes an integer and adds to it.
  * **Shorter** than writing \(\lambda x.3 + x\).
• 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 is the last one.
\textbf{Example}

\[
\text{\textbf{AND}} \ := \ \lambda(b, c : \text{Bool}).\text{Case}_{\text{Bool}} (\lambda(b' : \text{Bool}).\text{Bool}) : \text{Bool} \to \text{Bool} \to \text{Bool}
\]

- \text{AND} is the \textit{conjunction}:
  - \text{AND} \text{tt} \ c = c.
    Correct since \text{tt} \land c = c.
  - \text{AND} \text{ff} \ c = \text{ff}.
    Correct since \text{ff} \land c = \text{ff}.

- In the following we write \text{Bool}, if it
  - is a type in \textbf{boldface red},
  - and if it is a term, in \textit{italic blue}.
• Derivation of AND : \( \text{Bool} \to \text{Bool} \to \text{Bool} \):
  
  First we derive \( b : \text{Bool}, c : \text{Bool} \Rightarrow \lambda(b' : \text{Bool}).\text{Bool} \):

\[
\begin{array}{c}
\text{Bool} : \text{Set} \\
\hline
\text{Bool} : \text{Type} \\
\hline
b : \text{Bool} \Rightarrow \text{Context} \\
\hline
b : \text{Bool} \Rightarrow \text{Bool} : \text{Set} \\
\hline
b : \text{Bool} \Rightarrow \text{Bool} : \text{Type} \\
\hline
b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context} \\
\hline
b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Bool} : \text{Set} \\
\hline
b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Bool} : \text{Type} \\
\hline
b : \text{Bool}, c : \text{Bool}, b' : \text{Bool} \Rightarrow \text{Context} \\
\hline
b : \text{Bool}, c : \text{Bool}, b' : \text{Bool} \Rightarrow \text{Bool} : \text{Set} \\
\hline
b : \text{Bool}, c : \text{Bool} \Rightarrow \lambda(b' : \text{Bool}).\text{Bool} : \text{Bool}
\end{array}
\]
• We derive

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

(using part of the derivation above):

\[
\begin{align*}
\ldots \\
\frac{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context}}{b : \text{Bool}, c : \text{Bool}, b' : \text{Bool} \Rightarrow \text{Context}} \\
\frac{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context}}{b : \text{Bool}, c : \text{Bool}, b' : \text{Bool} \Rightarrow \text{Context}} & = (\lambda (b' : \text{Bool}).\text{Bool}).\text{Bool} \\
\frac{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context}}{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context}} & = (\lambda (b' : \text{Bool}).\text{Bool}).\text{Bool} \\
\frac{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context}}{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context}} & = (\lambda (b' : \text{Bool}).\text{Bool}).\text{Bool}
\end{align*}
\]
Example (Cont.)

- Similarly follows

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

\[ b : \text{Bool}, c : \text{Bool} \Rightarrow c : (\lambda (b' : \text{Bool}). \text{Bool}). \]

\[ \ldots \]

\[ b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context} \]

\[ b : \text{Bool}, c : \text{Bool} \Rightarrow c : \text{Bool} \]

\[ b : \text{Bool}, c : \text{Bool} \Rightarrow c : (\lambda (b' : \text{Bool}). \text{Bool}). \]

– We derive

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

\[ \ldots \]

\[ b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context} \]

\[ b : \text{Bool}, c : \text{Bool} \Rightarrow \text{ff} : \text{Bool} \]

\[ b : \text{Bool}, c : \text{Bool} \Rightarrow \text{ff} : (\lambda (b' : \text{Bool}). \text{Bool}). \]
Example (Cont.)

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

\[
\begin{align*}
    b : \text{Bool}, c : \text{Bool} & \Rightarrow \text{Context} \\
    \hline
    b : \text{Bool}, c : \text{Bool} & \Rightarrow b : \text{Bool}
\end{align*}
\]
Finally we obtain our judgement (we stack the premises because of lack of space):

\[
\begin{align*}
    b : \textbf{Bool}, c : \textbf{Bool} & \Rightarrow \lambda (b' : \textbf{Bool}). \textbf{Bool} : \textbf{Bool} \\
    b : \textbf{Bool}, c : \textbf{Bool} & \Rightarrow c : (\lambda (b' : \textbf{Bool}). \textbf{Bool}) : \textbf{Bool} \\
    b : \textbf{Bool}, c : \textbf{Bool} & \Rightarrow \texttt{ff} : (\lambda (b' : \textbf{Bool}). \textbf{Bool}) : \textbf{Bool} \\
    b : \textbf{Bool}, c : \textbf{Bool} & \Rightarrow b : \textbf{Bool} \\
    b : \textbf{Bool}, c : \textbf{Bool} & \Rightarrow \text{Case}_\textbf{Bool} (\lambda (b' : \textbf{Bool}). \textbf{Bool}) : \textbf{Bool} \\
    b : \textbf{Bool} & \Rightarrow \lambda (c : \textbf{Bool}). \text{Case}_\textbf{Bool} (\lambda (b' : \textbf{Bool}). \textbf{Bool}) : \textbf{Bool} \\
    \lambda (b, c : \textbf{Bool}). \text{Case}_\textbf{Bool} (\lambda (b' : \textbf{Bool}). \textbf{Bool}) : \textbf{Bool} & \quad \texttt{ff} : b : \textbf{Bool}
\end{align*}
\]
We can extend add elimination and equality rules, having as result

**Elimination Rule into Type**

\[
\frac{\begin{array}{c}
C : \text{Bool} \rightarrow \text{Type} \quad \text{ic} : C \text{ tt} \quad \text{ec} : C \text{ ff} \quad \text{cond} \\
\end{array}}{
\text{CaseBool } C \text{ ic ec cond : C cond}
\]

**Equality Rules into Type**

\[
\frac{\begin{array}{c}
C : \text{Bool} \rightarrow \text{Type} \quad \text{ic} : C \text{ tt} \quad \text{ec} : C \text{ ff} \\
\end{array}}{
\text{CaseBool } C \text{ ic ec tt = ic : C tt}
\]

\[
\frac{\begin{array}{c}
C : \text{Bool} \rightarrow \text{Type} \quad \text{ic} : C \text{ tt} \quad \text{ec} : C \text{ ff} \\
\end{array}}{
\text{CaseBool } C \text{ ic ec ff = ec : C ff}
\]
Elimination into Type (Cont.)

We can extend this into an elimination rule into Kind or other...
• We introduce Bool by simply listing its constructors (similarly to Haskell syntax):
  \[
  \text{data } \text{Bool} = \text{tt} \mid \text{ff}
  \]

• This introduces as well constants
  - \text{tt} :: \text{Bool}
  - \text{ff} :: \text{Bool}

• With this syntax, each constructor can occur at most once, i.e. we cannot define a second type having constructor \text{tt}, e.g. for defining True (which is used later):
  \[
  \text{data True} = \text{tt}
  \]
• The definition of Bool above is treated in Agda as an abbreviation for the following three **more fundamental Agda definitions**:

\[
\begin{align*}
\text{Bool} &::= \text{Set} \\
&= \text{data tt} \mid \text{ff} \\
\text{tt} &::= \text{Bool} \\
&= \text{tt@Bool} \\
\text{ff} &::= \text{Bool} \\
&= \text{ff@Bool}
\end{align*}
\]
• The definition of Bool as

$$\text{Bool} :: \text{Set}$$

$$= \text{data} \ tt \mid ff$$

introduces Bool as a set **having constructors** \(tt@\text{Bool}\) and \(ff@\text{Bool}\).

– So \(tt\) and \(ff\) **have to be defined separately**.
– If it is clear that the element in question is of type \(\text{Bool}\), replace \(tt@\text{Bool}\) by \(tt@\_\) .
– The definition of Bool as above **doesn’t prevent** the definition of another set with constructors \(tt\) or \(ff\).
– This syntax is the only one allowed, if one defines a set using the **data** keyword depending on arguments.

More about this later.
Internally, \texttt{tt} will always be represented as \texttt{tt@Bool}, similarly for \texttt{ff}.

So Agda evaluates \texttt{tt} to \texttt{tt@Bool}.

This can be seen when using for instance “\texttt{agda-compute-WHNF},” compute weak head normal form.
Elimination in Agda is based on case distinction.

Assume we want to define

- \( f : \text{Bool} \rightarrow \text{Bool} \), s.t.
  - \( f \text{ tt} = \text{ ff} \),
  - \( f \text{ ff} = \text{ tt} \).

So we have the goal:

\[
f (x :: \text{Bool}) :: \text{Bool} = \{! \ !\}
\]
• We can then type into the goal $x$ and choose the menu item “agda-case”.
  
  – This introduces a case distinction by the constructor used for introducing $x$:

    $x$ could have been introduced as `tt` or `ff`.

• The goal expands to:

$$f \ (x :: \text{Bool}) :: \text{Bool} \ \Rightarrow \ \text{case } x \ \text{of}$$

$$\begin{cases} \text{tt} & \rightarrow \{! \ !\}; \\ \text{ff} & \rightarrow \{! \ !\}; \end{cases}$$
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-)
    “agda-compute-WHNF”
  * “**Compute weak head normal form**” means essentially
    “compute the result of reducing that term”.
  - More precisely this means that a term is reduced until
    a constructor (or is a variable).
  - Then type into the mini-buffer **x**.
  - One gets the answer
    \( tt\_@\_ \).
Case Distinction (Cont.)

• Alternatively, check, the cursor being in that goal, the context:
  – (use goal-menu “agda-context”):
  It contains

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

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

• We obtain a function:

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

• \( f \ x \) is the\underline{ negation of} \( x \).
Testing the Defined Function

- We can test our function by using "agda-compute-WHNF".

- We have to create a goal for this.
  - The reduction machinery is context dependent.
  - The context depends on where in the buffer we are.
    - See the above example where $x$ was depending on the goal $tt$ or $ff$.
  - Not every place in the buffer is a good place.
  - Good places for context are goals, and that’s the only place Agda allows us to compute the weak head normal form.
Testing the Defined Function

• So we
  
  – type in a dummy goal:

\[
\text{test} \quad :: \quad \text{Set} \\
= \quad \{! \quad !\}
\]

  – move to the new goal
  – choose “agda-compute-WHNF”,
  – and type into the mini-buffer \( f \; \text{tt} \).

• The result shown is \( \text{ff @} \_ \).
(b) The Finite Sets

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

Formation Rule

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

Introduction Rules

\[ A^n_k : \text{Fin}_n \]  
(for \( k = 0, \ldots, n - 1 \))

Elimination Rule

\[ C : \text{Fin}_n \to \text{Set} \]
\[ s_0 : C A^n_0 \]
\[ s_1 : C A^n_1 \]
\[ \ldots \]
\[ s_{n-1} : C A^n_{n-1} \]
\[ a : \text{Fin}_n \]

\[ \text{Case}_n C s_0 \ldots s_{n-1} a : C a \]
Equality Rules

\[ C : \text{Fin}_n \rightarrow \text{Set} \]
\[ s_0 : C A^n_0 \]
\[ s_1 : C A^n_1 \]
\[ \ldots \]
\[ s_{n-1} : C A^n_{n-1} \]

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

(for \( k = 0, \ldots, n - 1 \)).
Omitting Premises in Equality Rules

Since the premises of the equality rule can in most cases be determined from the introduction and elimination rules, we will usually omit them, for instance for the previous rule:

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

We sometimes even omit the type:

\[ \text{Case}_n \, C \, s_0 \ldots s_{n-1} \, A_k^n = s_k \]
More compact elimination rules

- \( \text{Case}_n : (C : \text{Fin}_n \to \text{Set}) \)
  \[\to (s_0 : C^n_0)\]
  \[\to \ldots\]
  \[\to (s_{n-1} : C^n_{n-1})\]
  \[\to (a : \text{Fin}_n)\]
  \[\to C^n a\]
• Similarly as for \( \text{Bool} \) we can write down \textit{elimination rules}
\[ C : \text{Fin}_n \rightarrow \text{Type} \] (instead of \( C : \text{Fin}_n \rightarrow \text{Set} \)).

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

True is the special case Fin$_n$ for $n = 1$:

**Formation Rule**

True : Set

**Introduction Rules**

true : True

**Elimination Rule**

\[
\frac{C : \text{True} \rightarrow \text{Set} \quad c : C \text{ true} \quad t : \text{True}}{\text{Case}_{\text{True}} \ c \ t : C \ t}
\]

**Equality Rule**

\[
\frac{C : \text{True} \rightarrow \text{Set} \quad c : C \text{ true}}{\text{Case}_{\text{True}} \ c \text{ true} = c : C \text{ true}}
\]
• **Case**$_{\text{True}}$ is **computationally not very interesting**.
  
  – Case$_{\text{True}}$ $c$ is the untyped function $\lambda x.c$.
  – However, in Agda we might not be able to derive

    \[
    \lambda(t : \text{True}).c : (t : \text{True}) \to \text{C } t
    \]

• From a **logic point of view**, it expresses:
  From an element of $\text{C } \text{true}$ we obtain an element of $\text{C } t$ for every $t : \text{True}$.

  – So there is no $\text{C } : \text{True} \to \text{Set}$ s.t. $\text{C } \text{true}$ is inhabited for some other $x : \text{True}$.
  
  – This means that all elements of $x$ of type True are **indistinguishable**, i.e. they are **identical to true**.
  
  – This equality is called **Leibnitz equality**.
Rules for False

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

**Formation Rule**

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

**There is no Introduction Rule**

**Elimination Rule**

\[
\frac{C : \text{False} \to \text{Set} \quad f : \text{False}}{\text{Case}_\text{False} \quad f : C \ f}
\]

**There is no Equality Rule**
• False has **no elements.**

• It is formula which is **always false.**
  
  – As well called **absurdity.**

• **Case**\text{False} expresses: **from an element f of False we obtain an element of any set** (which might depend on \( f \)).
  
  – From a logic point of view this is "**Ex falso quodlibet**" (from the absurdity follows anything).
  
  E.g. A **false formula** like "0 = 1" or "Swansea lies in Germany" implies **everything.**
Case_{False} has no computational meaning, since there is nothing to be applied to.

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

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

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

  \[
  \text{is\_red (c :: Colour)} :: \text{Bool} = \text{case c of}
  \]

  \[
  \{ \begin{array}{c}
  \text{(red)} \rightarrow \text{tt}; \\
  \text{(green)} \rightarrow \text{ff}; \\
  \text{(blue)} \rightarrow \text{ff}; \\
  \end{array} \}
  \]
False in Agda

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

  \[
  \text{data False =}
  \]

- If we want to solve

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

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

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

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

• Assume the **type of trees**:

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

• Below we will show, how to introduce a function

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

s.t.

\[
\text{IsOak pine} = \text{False} \\
\text{IsOak oak} = \text{True}
\]
• 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":

\[
f \quad (t :: \text{Tree}) \\
(p :: \text{IsOak } t) \\
:: \quad A \\
= \quad \text{case } t \text{ of} \\
\quad \{ \text{pine } \rightarrow \quad \text{case } p \text{ of } \{ \} ; \\
\quad \text{oak } \rightarrow \quad \cdots ; \} 
\]
Example for the Use of False (Cont.)

• In order to use \( f \) we have to know that \( t \) is an oak tree,
  – i.e. we have to provide an argument \( p \) which expresses that we know this.

• Note that we don’t have to invent a result of \( f \) in case

Critical Systems, CS_411, Lentterm 2003, Sec. B3
• Similarly we can introduce a stack, together with a predicate

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

s.t.

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

if \( s \) is the empty stack.

• Now we can define

\[
\text{pop} \ ((s :: \text{Stack}) \\
(p :: \text{NotEmpty } s) \\
:: \text{Stack} \\
= \cdots)
\]

• Again we **don't have to provide a result, in case \( s \) is empty.**
The definition of True in Agda is straightforward:

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

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

\[
g \ (x :: \text{True}) :: \text{Bool} = \text{case } x \text{ of } \{(\text{true}) \rightarrow \{! \;!\};\}
\]
 Atomic Formulae and the Traffic Light Example

Atomic Formulae

- We have already introduced two formulae:
  - True.
    - True is inhabited.
      - There is a proof of it (true).
      - True is therefore type-theoretically true: A formula is type-theoretically true, if it is provable.

Truth in type theory means provability.
– False.
  * False is **not inhabited**.
    · There is **no proof** of False.
      Furthermore, from any proof of False we can derive everything (elimination rules for False).
    · False is therefore **type-theoretically false**:
      A formula is **type-theoretically false**, if from it we can derive everything.
    · Since this implies that we can derive False and from False we can derive everything, this is equivalent to the following:
    · A formula is **type-theoretically false**, if from a proof of it we can derive False (i.e. a contradiction).
• 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 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 *true and false*.
• We can **map** truth values to their corresponding formula:

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

• This can be defined using **case distinction**.
atom (Cont.)

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

\[
\begin{align*}
b : \text{Bool} \\
\Rightarrow \text{atom } b : \text{Set}
\end{align*}
\]

\[
\text{atom } \texttt{tt} = \text{True}
\]

\[
\text{atom } \texttt{ff} = \text{False}
\]
atom \ (b :: \text{Bool}) :: \text{Set} = \text{case } b \text{ of}
\begin{align*}
\{ & (\text{tt}) \rightarrow \text{True;} \\
& (\text{ff}) \rightarrow \text{False;} \}
\end{align*}
Decidable Predicates

- Using `atom` we can now define **decidable predicates** on sets.

- Assume we have a **set of states** of a system $A$.
  - E.g. the set of states a railway controller can choose.

- Assume we have a function $f : A \rightarrow \text{Bool}$.
  - E.g. $f$ $a$ means: **state $a$ is safe**.
Decidable Predicates (Cont.)

• Let now \( g : A \rightarrow \text{Set}, \ g \ a = \text{atom}(f \ a) \).
  
  – If \( f \ a \) is true (e.g. \( a \) is safe), \( g \ a \) is inhabited.
  – If \( f \ a \) is false (e.g. \( a \) is unsafe), \( g \ a \) is not inhabited.

• Now, the existence of a \( h : (a : A) \rightarrow g \ a \) means:
  
  – For all \( a : A \) we have \( g \ a \) is inhabited,
  – ie. for all \( a : A \), \( f \ a \) is true,
  – e.g. for all \( a : A \), \( a \) is safe.
The Traffic Light Example

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

• Assume from each direction A, A’, B, B’ there is one traffic light,
  – but A and A’ always coincide, similarly B and B’.

Critical Systems, CS_411, Lentterm 2003, Sec. B3
The Set of Physical States

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

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

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

  \[
  \text{Phys\_State} \::\: \text{Set} \\
  = \text{sig}\{ \text{sigA} \::\: \text{Colour}; \text{sigB} \::\: \text{Colour}; \}
  \]
The Set of Control States

- The set of control states is a set of states of the system, a controller of the system can choose.
  - Each of these states should be safe.
  - In our example, all safe states will be captured (this can usually be only achieved in small examples).

- A complete set of control states consists of:
  - Allred – all signals are red.
  - Agreen – signal A (and A’) is green, signal B is red.
  - Bgreen – signal B is green, signal A is red.
• We therefore define

\[
\text{data Control\_State} = \text{Allred} \mid \text{Agreen} \mid \text{Bgreen}
\]
• We define the state of signals $A$, $B$ depending on a control state:

$$
toSigA \ (s :: \text{Control\_State})
= \begin{cases}
\text{Colour} & : \text{Colour} \\
(\text{Allred}) & \to \text{red}; \\
(\text{Agreen}) & \to \text{green}; \\
(\text{Bgreen}) & \to \text{red}; \\
\end{cases}
$$

$$
toSigB \ (s :: \text{Control\_State})
= \begin{cases}
\text{Colour} & : \text{Colour} \\
(\text{Allred}) & \to \text{red}; \\
(\text{Agreen}) & \to \text{red}; \\
(\text{Bgreen}) & \to \text{green}; \\
\end{cases}
$$
Now we can define the physical state corresponding to a control state:

\[
\text{phys\_state} \ (s :: \text{Control\_State}) :: \text{Phys\_State} \\
\quad = \ \text{struct} \{ \text{sigA} = \text{toSigA} \text{s} \} \\
\quad \quad \text{sigB} = \text{toSigB} \text{s}\n\]
• We define now **when a physical state is safe:**
  
  – It is **safe iff not both signals are green.**
  – We define now a corresponding predicate **directly**, without a Boolean function.
  – We first define a predicate depending on two signals:

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

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

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

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

• Now we show that all control states are safe:

\[
\text{cor\_proof } (s :: \text{Control\_State})
\]
\[
:: \text{Cor(phys\_state } s) = \text{ case } s \text{ of }
\]
\[
\{ (\text{Allred}) \rightarrow \text{true};
\]
\[
(A\text{green}) \rightarrow \text{true};
\]
\[
(B\text{green}) \rightarrow \text{true};
\]
Safety of the System (Cont.)

- The first element true was an element of Cor(phys_state), 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 would have gotten instances where the goal to solve is False, which we can’t solve.
Safety of the System (Cont.)

- If one makes a **mistake** which results in an unsafe situation (e.g. sets toSigB Agreen = green, then in the last step we obtain one goal of type False.
  - Then we can’t solve this goal directly and **cannot prove**.
  - (In fact we could type-theoretically solve this goal by using full recursion, e.g. solve this goal as **cor_proof Agreen**), but this would be rejected by the termination check.)
(d) The Disjoint Union of Sets

**Formation Rule**

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

**Introduction Rules**

\[
A : \text{Set} \quad B : \text{Set} \quad a : A \\
\frac{}{\text{inl} A \ B \ a : A + B}
\]

\[
A : \text{Set} \quad B : \text{Set} \quad b : B \\
\frac{}{\text{inr} A \ B \ b : A + B}
\]

**Elimination Rule**

\[
A : \text{Set} \quad B : \text{Set} \quad C : (A + B) \rightarrow \text{Set} \\
sl : (a : A) \rightarrow C (\text{inl} A \ B \ a) \\
sr : (b : B) \rightarrow C (\text{inr} A \ B \ b) \\
d : A + B \\
\frac{}{\text{Plus\_Split} A \ B \ C \ sl \ sr \ d : C \ d}
\]
The Disjoint Union of Sets (Cont.)

Equality Rules

\[
\text{Plus\_Split } A \ B \ C \ sl \ sr \ (\text{inl } A \ B \ a) = sl \ a : C \ (\text{inl } A \ B)
\]

\[
\text{Plus\_Split } A \ B \ C \ sl \ sr \ (\text{inr } A \ B \ b) = sr \ b : C \ (\text{inr } A \ B)
\]
• A more compact notation is:
  - \((+): \text{Set} \to \text{Set} \to \text{Set}\), written infix.
  - \(\text{inl}: (A, B: \text{Set}) \to A \to (A + B)\).
  - \(\text{inr}: (A, B: \text{Set}) \to B \to (A + B)\).
  - \(\text{Plus}\_\text{Split}: (A, B: \text{Set})
    \to (C: (A + B) \to \text{Set})
    \to (sl: (a: A) \to C \ (\text{inl} A B a))
    \to (sr: (b: B) \to C \ (\text{inr} A B b))
    \to (d: A + B)
    \to C \ d\).
• The disjoint union can be defined as a “data”-set having `inl` (in-left) and `inr` (inright):

\[
(+) (A :: \text{Set}) \\
(B :: \text{Set}) \\
:: \text{Set} \\
= \text{data inl}(a :: A) | \text{inr}(b :: B)
\]
Disjoint Union in Agda (Cont.)

- The notation (+) means, that + can be used **infix**.

- Now we have, if $A, B :: \text{Set}$:
  - $\text{inl}(A + B) :: A \rightarrow (A + B)$
  - $\text{inr}(A + B) :: B \rightarrow (A + B)$
  - This can be checked using the menu “agda-infer-type”.
  - Note that we cannot assign a type to $\text{inr}$. 
  - (+) **cannot** be defined using the abbreviated data notation
    (which would be of the form
    data (+) = ⋯).
• Elimination is again represented by case distinction. So if want to define for \( A, B :: Set \) for instance

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

we can type into the goal \( c \) and choose menu “agda-case”. 

Critical Systems, CS_411, Lentterm 2003, Sec. B3
• We obtain

\[
f \quad (c :: A + B) \\
:: \quad \text{Bool} \\
= \quad \text{case } c \text{ of} \\
\begin{cases} 
\text{(inl} \ a) & \rightarrow \{! \ !\}; \\
\text{(inr} \ b) & \rightarrow \{! \ !\}; 
\end{cases}
\]

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

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

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

- Now one can define for instance

  \[
  \begin{array}{ll}
  \text{isFlower } (p :: \text{Plant}) & :: \text{Bool} \\
  = \text{case } p \text{ of} \\
  & \{ (\text{tree } t) \rightarrow \text{ff}; \\
  & (\text{flower } f) \rightarrow \text{tt}; \\
  \}
  \end{array}
  \]
(e) The $\Sigma$-Set

**Formation Rule**

$$\frac{A : \text{Set} \quad B : A \rightarrow \text{Set}}{\Sigma A B : \text{Set}}$$

**Introduction Rule**

$$\frac{A : \text{Set} \\
B : A \rightarrow \text{Set} \\
a : A \\
b : B \ a} {p \ A \ B \ a \ b : \Sigma A B}$$

**Elimination Rule**

$$\frac{A : \text{Set} \quad B : A \rightarrow \text{Set} \quad C : (\Sigma A B) \rightarrow \text{Set} \\
s : (a : A) \rightarrow (b : B \ a) \rightarrow C \ (p \ A \ B \ a \ b))} {\text{Sigma_SPLIT} \ A \ B \ C \ s \ d : C \ d}$$
The $\Sigma$-Set (Cont)

Equality Rule

Sigma_Split $A B C s (p A B a b) = s a b : C (p A B)$
The $\Sigma$-Set using the Logical Framework

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

Critical Systems, CS_411, Lentterm 2003, Sec. B3
The \( \Sigma \)-Set using the Logical Framework (Cont.)

- \( \text{Sigma-Split} \)
  : \( (A : \text{Set}) \)
  \( \rightarrow (B : A \rightarrow \text{Set}) \)
  \( \rightarrow (C : (\Sigma A B) \rightarrow \text{Set}) \)
  \( \rightarrow (s : (a : A, b : B a) \)
    \( \rightarrow C (p A B a b)) \)
  \( \rightarrow (d : \Sigma A B) \)
  \( \rightarrow C d \).
The dependent product and the \(\Sigma\)-set are very similar:

- Both have similar introduction rules (for the \(\Sigma\)-set, the constructors have additional arguments \(A, B\) necessary for bureaucratic reasons).
- One can define the projections \(\pi_0, \pi_1\) using \texttt{Sigma\_Split}:
  \[
  \begin{align*}
  \pi_0 &= \texttt{Sigma\_Split} \ A \ B \ (\lambda x. A) \ (\lambda (x : A). \lambda (y : B x). x ) \\
  \pi_1 &= \texttt{Sigma\_Split} \ A \ B \ (\lambda x. B \ \pi_0 (x)) \ (\lambda (x : A). \lambda (y : B x). y )
  \end{align*}
  \]

- On the other hand, from \(\pi_0, \pi_1\) we can define \texttt{Sigma\_Split} as follows:
  \[
  \lambda A, B, C, s, d. s \ \pi_0 (d) \ \pi_1 (d).
  \]
The Σ-Set and the Dependent Product

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

• Because of the lack of $\eta$-rule, $\Sigma$ works usually better than the dependent product in Agda.
  – I personally don’t use the dependent product of Agda.
The Σ-Set in Agda

- Σ can be defined as a “data”-set with constructor p:

\[
\begin{align*}
\text{Sigma} & \quad (A :: \text{Set}) \\
& \quad (B :: A \rightarrow \text{Set}) \\
& :: \quad \text{Set} \\
& = \quad \text{data } p \ (a :: A) \ (b :: Ba)
\end{align*}
\]
• Again one usually defines concrete $\Sigma$-sets more directly.

• **Example:** Assume we have defined
  
  – a set Plant\_Group for **groups of plants** (e.g. “tree”),
  
  – depending on $g ::$ Plant\_Group, sets Plants\_in\_group for **plants in that group**.

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

  \[
  \text{data Plant} = \text{plant} \ (g :: \text{Plant\_Group})(pg :: \text{Plants\_in\_group})
  \]
• Not surprisingly, for **elimination** we use **case distinction**, e.g.:

\[
\begin{align*}
  f & \quad (p :: \text{Plant}) \\
  :: & \quad \text{Plant}\_\text{group} \\
  = & \quad \text{case } p \text{ of} \\
    & \quad \{ \text{ (plant } g \text{ pg) } \rightarrow g; \}
\end{align*}
\]
We have seen how to represent atomic decidable formulae.

Now treatment of complex formulae constructed using logical connectives.
Conjunction

- \( A \land B \) is true iff both \( A \) is true and \( B \) is true.

- Therefore a proof of \( A \land B \) consists of a **proof of \( A \)** and **proof of \( B \)**.
  - It is therefore a pair \( \langle p, q \rangle \) consisting of a proof \( p \) of \( A \) and a proof \( q \) of \( B \).

- Therefore the set of proofs of \( A \land B \) is the set of pairs of \( A \) and \( B \), i.e. \( A \times B \).

- We can **identify** \( A \land B \) with \( A \times B \).
Conjunction (Cont.)

- 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 $B$:

\[
\begin{array}{c}
p : A \\
q : B \\
\langle p, q \rangle : A \land B
\end{array}
\]

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

- This is what is expressed by the **ordinary introduction rule**:

\[
\begin{array}{c}
A \\
B \\
\hline
A \land B
\end{array}
\]
• The **elimination rule** for $\land$ allows to project a proof of $A \land B$ to a proof of $A$ and a proof of $B$:

$$
\frac{p : A \land B}{\pi_0(p) : A} \quad \frac{p : A \land B}{\pi_1(p) : B}
$$

• 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 $\land$:

$$
\frac{A \land B}{A} \quad \frac{A \land B}{B}
$$
Disjunction

- $A \lor B$ is true iff $A$ is true or $B$ is true.

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

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

- Therefore the set of proofs of $A \lor B$ is the **disjoint union** $A + B$.

- We can **identify** $A \lor B$ with $A + B$. 
With this identification, the **introduction rules** for $\vee$ allow to form a proof of $A \vee B$ from a proof of $A$ or from a proof of $B$.

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

- Omitting the premises $A, B : \text{Set}$ and omitting them as arguments of $\text{inl}$ and $\text{inr}$ (which is needed only for bureaucratic reasons), we get:

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

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

- This is what is expressed by the ordinary introduction rules:

$$\frac{A}{A \lor B} \quad \frac{B}{A \lor B}$$
• The **elimination rule** for + allows to form from an element of any set $C$ provided we can compute such an element from $A$ and from $B$:

$$
\begin{array}{l}
A : \text{Set} \\
B : \text{Set} \\
C : (A \lor B) \rightarrow \text{Set} \\
sl : (a : A) \rightarrow C \ (\text{inl } A \ B \ a) \\
sr : (b : B) \rightarrow C \ (\text{inr } A \ B \ b) \\
d : A \lor B \\
\text{Plus_Split } A \ B \ C \ sl \ sr \ d : C \ d
\end{array}
$$

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

$$
\begin{array}{l}
d : A \lor B \\
sl : A \rightarrow C \\
sr : B \rightarrow C \\
\text{Plus_Split } sl \ sr \ d : C
\end{array}
$$
Disjunction (Cont.)

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

• This is what is expressed by the ordinary elimination rule:

$$
\begin{array}{ccc}
A & & B \\
\cdot & & \cdot \\
\cdot & & \cdot \\
& & \\
A \lor B & & C & & C \\
\end{array}
$$

• (Note that in the ordinary elimination rule, from the premise “$C$ derivable from $A$” we obtain “$A \rightarrow C$”, similarly for “$C$ derivable from $B$” we get $B \rightarrow C$.)
• We write temporarily $\supset$ for logical implication, in order to distinguish it from the function type $\rightarrow$.
  
  – Below we see that $\supset$ can be identified with $\rightarrow$.

• $A \supset B$ is true iff, whenever $A$ is true then $B$ is true.

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

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

• Therefore the set of proofs of $A \supset B$ is the function type $A \rightarrow B$.

• We can identify $A \supset B$ with $A \rightarrow B$. 
• With this identification, the **introduction rule for** \( \supset \) allows to form a proof of \( A \supset B \) from a proof of \( B \) depending on a proof \( p \) of \( A \):

\[
\frac{p : A \Rightarrow q : B}{\lambda(p : A).q : A \supset B}
\]

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

  **then we can derive** \( A \supset B \) **without assuming** \( p : A \).
This is what is expressed by the ordinary introduction rule:

\[
\begin{array}{c}
A \\
\vdots \\
B \\
\hline
A \supset B
\end{array}
\]
• The **elimination rule for** $\supset$ allows to apply a proof $p$ of $A \supset B$ of $q$ of $A$ in order to obtain a proof of $B$:

\[
\begin{array}{c}
p : A \supset B \\
q : A \\
\hline
p \quad q : B
\end{array}
\]

• This means that we can **derive from** $A \supset B$ and $A$ that $B$ holds.

• This is what is expressed by the **ordinary elimination rule**:

\[
\begin{array}{c}
A \supset B \\
\hline
B \\
A
\end{array}
\]
• \( \neg A \) has the same meaning as \( A \supset \bot \) 
  (where \( \bot \) is absurdity or the set False):

  – If there is no proof of \( A \), then we can prove \( A \supset \bot \).
  – 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.

• Therefore we can identify \( \neg A \) with \( A \rightarrow \text{False} \).
• Since we have many types, we have to write when using quantifiers the type, the bound variable is ranging over:
  We write therefore $\forall x : A.B$, $\exists x : A.B$.

• $\forall x : A.B$ is true iff, for all $x : A$ there exists a proof of $B$.

• 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 $(x : A) \rightarrow B$.

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

$$
x : A \implies p : B
\frac{}{\lambda(x : A). p : \forall x : A. B}
$$

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

Critical Systems, CS_411, Lentterm 2003, Sec. B3
Universal Quantification (Cont.)

• This is what is expressed by the **ordinary introduction rule** for \( \forall \):

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

where

- **x** might not occur free in any assumption of the proof.
  * This is guaranteed in type theory, since \( x : A \) must be the last element of the context, so any other assumptions must be located before it and can therefore **not depend on** \( x : A \).
- The **conclusion will no longer depend on** free variables.
  * This corresponds in type theory to the fact that \( x : A \) **do not occur in the context of the conclusion.**
Universal Quantification (Cont.)

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

\[
\frac{p : \forall x : A. B \quad a : A}{p a : B[x := a]}
\]

- This means that we can derive from $\forall x : A. B$ and an element $a : A$ that $B[x := a]$ holds.
• This is what is expressed by the **ordinary elimination rule**.
  
  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 derivation.

\[
\forall x : A. B \quad \frac{a : A}{B[x := a]}
\]
Existential Quantification

- \( \exists x : A. B \) is true iff there exists an \( a : A \) such that \( B[x := a] \) is true.

- Therefore a proof of \( \exists x : A. B \) is a pair \( \langle a, p \rangle \) consisting of an \( a : A \) and a proof \( p \) of \( B[x := a] \).

- Therefore the set of proofs of \( \exists x : A. B \) is the dependent product \( (x : A) \times B \).

- We can identify \( \exists x : A. B \) with \( (x : A) \times B \).
With this identification, the **introduction rule** for $\exists$ allows to form a proof of $\exists x : A.B$ from an element $a : A$ and a proof $p : B[x := a]$

$$a : A \quad p : B[x := a] \quad \langle a, p \rangle : \exists x : A.B$$

This is what is expressed by the **ordinary introduction rule**

$$a : A \quad B[x := a] \quad \exists x : A.B$$
Existential Quantification (Cont.)

- The **elimination rule** for the dependent product allows to project a proof of $\exists x : A.B$ to an element $\pi_0(p) : A$ and proof $\pi_1(p) : B[x:=\pi_0(p)]$.

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

- From this we can derive a rule which is essentially that used in natural deduction (in which one doesn’t have explicit proofs):
  - Assume:
    * $C : \text{Set}$, which does not depend on $x : A$,
    * $p : \exists x : A.B$ and
    * $x : A, y : B \Rightarrow c : C$.
  - Then we have $c[x := \pi_0(p), y := \pi_1(p)] : C$, **not depending on** $x:A$ or $y:B$. 
Therefore the rule in natural deduction follows from rules:

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

where the conclusion does not depend on \( x : A \) and \( B \).
From type theoretic proofs we can directly extract programs.

For instance, if \( p : \forall x : A. \exists y : B.C(x,y) \), then we have

- for \( x : A \) it follows \( b := \pi_0(p x) : B \) and \( \pi_1(p x) : C(x, b) \).
- Therefore \( f := \lambda x : A. \pi_0(p x) \) is a function \( A \to B \), and

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

i.e. we have a proof that \( \forall x : A. C(x, f(x)) \) holds.
- Therefore, from a proof of \( \forall x : A. \exists y : B.C(x, y) \), we have a function, which computes the \( y \) from the \( x \).
• We can derive as well a function which depending on \( p \):
  \[ A + B \]
  decides whether \( p = \text{inl}(a) \) or \( p = \text{inr}(b) \).

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

• Now:
  – Any function in type theory is recursive.
  – We **cannot decide the Turing Halting problem**, i.e. we cannot decide for a Turing machine whether it halts or not.
  – Therefore we **cannot prove in type theory**

  \[ \forall x : \text{Turing\_Machine}. (x \text{ halts} \lor \neg (x \text{ halts})) \]
• In classical logic we can prove the above, since we can derive $A \lor \neg A$ (tertium non datur) for any formula $A$.

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

  – The logic of type theory is intuitionistic (constructive), $A \lor \neg A$ and $\neg\neg A \rightarrow A$ don’t hold for all formulae $A$. 
• In classical logic,
  - \( \exists x : A \land B \) is equivalent to \( \neg \forall x : A \land \neg B \),
  - \( A \lor B \) is equivalent to \( \neg (\neg A \land \neg B) \).

• If we take decidable atomic formulae only and replace \( \exists x : A \land B \) and \( A \lor B \) by the above formulae, then all formulas provable in classical logic are derivable.
  - This requires \( (\neg \neg A) \rightarrow A \), which can be shown for all formulae built from decidable atomic formulae using \( \neg \), \( \rightarrow \), \( \land \), \( \forall \).
  - The formula \( A \lor \neg A \) translates into \( \neg (\neg A \land \neg \neg A) \), which trivially holds, since \( \neg A \) and \( \neg \neg A \) implies \( \bot \).

• In this sense, type theory contains classical logic, but has as well so called strong disjunction and existential quantification.
Proof (using classical logic) of

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

We have classically:

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

* If \( A \) is true, then \( \neg \neg A \rightarrow A \) holds.
* If \( A \) is false, then \( \neg \neg A \) is false, therefore \( \neg \neg A \rightarrow A \)
• We show intuitionistically \( \neg(\exists x : A.B) \leftrightarrow \forall x : A.\neg B \):
  
  - Assume \( \neg(\exists x : A.B) \), \( x : A \) and show \( \neg B \).
    If we had \( B \), then we had \( \exists x : A.B \), contradicting \( \neg(\exists x : A.B) \).
    \( \neg B \).
  
  - Assume \( \forall x : A.\neg B \). Show \( \neg(\exists x : A.B) \):
    Assume \( \exists x : A.B \). Assume \( x \) s.t. \( B \) holds.
    By \( \forall x : A.\neg B \) we get \( \neg B \), therefore a contradiction.

• Now it follows (classically):

\[
(\exists x : A.B) \leftrightarrow \neg\neg(\exists x : A.B) \leftrightarrow \neg\forall x : A.
\]
• Proof of $A \lor B \leftrightarrow \neg(\neg A \land \neg B)$:

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

    $$(A \lor B) \leftrightarrow \neg\neg(A \lor B) \leftrightarrow \neg(\neg A \land \neg B)$$
Constructive Logic (Cont.)

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

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

- **Strong disjunction and existential quantificaton** is expressed by the original type theoretic formulae.
(f) The Set of Natural Numbers

- The set $\mathbb{N}$ is the type theoretic representation of the set $\mathbb{N} := \{0, 1, 2, \ldots\}$.

- $\mathbb{N}$ can be generated by
  - starting with the empty set,
  - adding 0 to it, and
  - adding, whenever we have $x$ in it $x + 1$ to it.
The Set of Natural Numbers (Cont.)

- Let \( S \) be a type theoretic notation for the operation \( x \mapsto x + 1 \).

- Then the type theoretic rules are

\[
\begin{align*}
N & : \text{Set} \\
0 & : N \\
\frac{n : N}{S\ n : N}
\end{align*}
\]
Primitive Recursion

• **Primitive Recursion expresses:**
  Assume we have
  
  – \( a : \mathbb{N} \).
  – and, if \( n : \mathbb{N}, \ x : \mathbb{N} \) then \( g \ n \ x : \mathbb{N} \).

  Then we can define \( f : \mathbb{N} \rightarrow \mathbb{N} \), s.t.
  
  – \( f \ 0 = a \),
  – \( f \ (S \ n) = g \ n \ (f \ n) \).
• The **computation of** $f\ n$ proceeds now as follows:

  – Compute $n$.
  – If $n = 0$, then the result is $a$.
  – Otherwise $n = S(n')$.
    * We assume that we have determined already how to compute $f\ n'$.
    * Now $f\ n$ reduces to $g\ n'\ (f\ n')$.
    * $g\ n'\ (f\ n')$ can be computed, since we know how to compute $\cdot\ g$ $\cdot\ f\ n'$. 

Critical Systems, CS_411, Lentterm 2003, Sec. B3
Example

• The function \( f : \mathbb{N} \to \mathbb{N} \) with \( f(x) = 2 \cdot x \) can be defined recursively by:
  
  \[
  \begin{align*}
  &f(0) = 0, \\
  &f(S\ n) = S\ (S\ (f\ n)).
  \end{align*}
  \]

• Therefore take in the definition above:
  
  \[
  \begin{align*}
  &a = 0, \\
  &g\ n\ x = S\ (S\ x).
  \end{align*}
  \]
• We can generalize primitive recursion as follows:
  – First we can replace the range of $f$ by an arbitrary set $C$:
    * i.e. we allow for any set $C$
    \[
    f : \mathbb{N} \rightarrow C
    \]
  – Further, $C$ can now depend on $\mathbb{N}$.

• We obtain the following set of rules:
Rules for the Natural Numbers

Formation Rule

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

Introduction Rules

\[ 0 : \text{N} \]
\[ n : \text{N} \quad S \quad n : \text{N} \]

Elimination Rule

\[ C : \text{N} \rightarrow \text{Set} \]
\[ a : C \ 0 \]
\[ f : (x : \text{N}) \rightarrow C \ x \rightarrow C \ (S \ x) \]
\[ n : \text{N} \]
\[ P \ C \ a \ f \ n : C \ n \]

Equality Rules

\[ P \ C \ a \ f \ 0 = a \]
\[ P \ C \ a \ f \ (S \ n) = f \ n \ (P \ C \ a \ f \ n) \]
• Note that if we define in the elimination rule $g := P C f$

  – The conclusion of the elimination rule reads:

    $$g \, n : C \, n$$

    which means that

    $$\lambda(n : N).g \, n : (n : N) \to C \, n.$$  

  – The equality rules read:

    $$g \, 0 = a$$

    $$g \, (S \, n) = f \, n \, (g \, n)$$
The more compact notation is:

- $N : \text{Set}$,
- $0 : N$,
- $S : N \rightarrow N$,
- $P : (C : N \rightarrow \text{Set}) \rightarrow C \ 0 \rightarrow ((x : N) \rightarrow C \ x \rightarrow C \ (S \ x)) \rightarrow (n : N) \rightarrow C \ n$.
Natural Numbers in Agda

• N is defined using **data**:

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

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

• Therefore we have

\[
\begin{align*}
\text{Z} & :: \text{N} \\
\text{S} & :: \text{N} \to \text{N}
\end{align*}
\]
Elimination works via case distinction in Agda.

- If we want to introduce

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

\* \(A\) possibly depending on \(n\),
we can type into the goal \(n\) and use the menu agda-case.
We get

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

\[ \{ (Z) \rightarrow \{! !\}; \ (S \ n') \rightarrow \{! !\}; \} \]
• For solving the goals, we can now make use of $f$. That will be accepted by the type checker.

• However, if we use of full $f$, and then use menu item “agda-check-termination”, we might obtain an error-message.

• If we
  – do not make use of $f$ in the case $n=\mathbb{Z}$ and
  – only use of $f\ n'$ in case $n = S\ n'$.

  then agda-check-termination succeeds.
Elimination Rules for N in Agda (Cont.)

- If `agda-check-termination` succeeds, the definition should
  - (The lecturer hasn’t checked the algorithm).

- However, if `agda-check-termination` fails, the definition may not be correct.
Example of the Power of Termination Check

- The following definition of the \textbf{Fibonacci numbers} can’t be defined directly using the rules of type theory, but it \textit{can be defined} as follows and \texttt{agda-check-termination} accepts it:

\[
\text{fib} \ (n :: N) :: N = \text{case } n \text{ of } \begin{cases} 
(Z) & \rightarrow \text{one}; \\
(S \ n') & \rightarrow \text{case } n' \text{ of } \begin{cases} 
(Z) & \rightarrow \text{one}; \\
(S \ n'') & \rightarrow \text{fib } n'
\end{cases}
\end{cases}
\]
Example for Limitations of Termination Check

- Assume we define the **predecessor function**

\[
\text{pred} \ (n :: N) :: N = \text{case } n \text{ of} \\
\{ \ (Z) \rightarrow Z; \\
\ (S \ n') \rightarrow n' \}; \\
\]

i.e.

\[
\text{pred}(n) = \begin{cases} 
0 & \text{if } n = 0 \\
 n - 1 & \text{otherwise.} 
\end{cases}
\]
Example for Limitations of Termination Check

• Then the function

\[
\begin{align*}
&f \ (n :: N) \\
&:: \ N \\
= \ &\text{case } n \ \text{of} \\
&\{ \ (Z) \rightarrow Z; \\
&\ (S \ n') \rightarrow f \ (\text{pred } n); \}
\end{align*}
\]

terminates always

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

• However, \texttt{agda-check-termination} fails.
Limitations of the Termination Check (Cont.)

- Because of the **undecidability of the Turing halting problem**, it is undecidable whether a recursively defined function terminates.

- 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:

$$
(+)
\begin{array}{ll}
(n, m :: \mathbb{N}) & :: \mathbb{N} \\
= & \text{case } m \text{ of} \\
& \{ \begin{array}{ll}
(Z) & \rightarrow n; \\
(S m') & \rightarrow S (n + m');
\end{array} \}
\end{array}
$$

- The definition expresses:

$$
\begin{align*}
n + 0 &= n \\
n + (m' + 1) &= (n + m') + 1
\end{align*}
$$
Example: Addition

- Note that (+) is used **infix**, i.e. we write \( n + m \) for \((+) \) \(nm\).

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

- Definition

\[(*) \quad (n, m :: \mathbb{N}) :: \mathbb{N} = \text{case } m \text{ of} \]
\[
\begin{cases} 
(Z) & \rightarrow \mathbb{Z}; \\
(S \, m') & \rightarrow n \cdot m' + n;
\end{cases}
\]

- The definition expresses:

\[
n \cdot 0 = 0
\]
\[
n \cdot (m' + 1) = (n \cdot m') + n
\]
• Again * is treated **infix**.

• Agda has built in that * **binds more than** +.
  
  – \( n \times m' + n \) is treated as \((n \times m') + n\).

• Note that the definition of * requires, that + **is already defined**.
The equality \((n \equiv m) :: \text{Set}\) for \(n, m :: \mathbb{N}\) can be defined using the equations:

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

\[
(==) \quad (n, m :: \mathbb{N}) \\
:: \quad \text{Set} \\
= \quad \text{case } n \text{ of} \\
\{ \quad (Z) \quad \rightarrow \quad \text{case } m \text{ of} \\
\quad \{ \quad (Z) \quad \rightarrow \quad (S \ m') \quad \rightarrow \\
\quad (S \ n') \quad \rightarrow \quad \text{case } m \text{ of} \\
\quad \{ \quad (Z) \quad \rightarrow \quad (S \ m') \quad \rightarrow \\
\}
\]

• Task of coursework 3, Question 1 to fill in those goals.
Reflexivity of $==$

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

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

- **Type theoretically** this means that we have to define a function $\text{refl}$:

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

- This can now be shown using \textit{case distinction}:

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

- Case $n = Z$ is trivial.
- Case $n = S n'$ can be solved using refl $n'$ (which is defined before).
- Task of Coursework 3, Question 1 (e) to solve this goal.
Symmetry of $==$

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

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

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

$$\text{sym} \ (n, m : \mathbb{N}) \ (p :: n == m) \ :: \ m == n \ = \ \{! \ !\}$$
Symmetry of == (Cont.)

- This can now be shown using case distinction:

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

Critical Systems, CS_411, Lentterm 2003, Sec. B3
Symmetry of == (Cont.)

• The first goal can be solved by using true (since \( Z == Z \) = True).

• For the second goal we know \( p \) is an element of \( Z == S \)
  – Therefore if we make case distinction on \( p \) we get

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

  and have solved the second goal.

• Similarly the third goal can be solved.
Symmetry of == (Cont.)

- In the fourth goal, we have as type of goal $S \ m' == S \ n'$ which is identical to $m' == n'$.
  - The type of $p$ is $S \ n' == S \ m'$ which is identical to $n' == m'$.
- The goal can be solved by using $\text{sym} \ n' \ m' \ p$.
  - Note that we can use here $p$ since it is of type $n' == m'$.
    - It is correct to use it since $n'$ is introduced before $m'$.
    - Therefore $\text{sym} \ n'$ can be defined before $\text{sym} \ n$.
    - This definition will be accepted by agda-check-termination.
Example: Tuples (or Vectors) of Length $n$

- Define first

$$
data\ Nil = \text{nil}$$

$$
\text{Cons } (A, B :: \text{Set}) :: \text{Set} = \text{data cons}(a :: A)(b :: B)$$. 

Critical Systems, CS_411, Lentterm 2003, Sec. B3
Example: Tuples (or Vectors) of Length n

- Now we can define (we use \texttt{Vec} for vector)

\[
\text{Vec} \quad (A :: \text{Set}) \\
(n :: N) \\
:: \text{Set} \\
= \text{case } n \text{ of} \\
\quad \{(Z) \rightarrow \text{Nil}; \\
\quad (S \ m') \rightarrow \text{Cons } A \ (\text{Vec } A)
\]
• Therefore (with the obvious definition of two),

\[ \text{Vec } A \ n = \text{Cons } A \ (\text{Cons } A \ \cdots \ (\text{Cons } A \ \text{Nil}) \ \cdots) \]

\( n \text{ times} \)

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

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

for elements \( a_1, \ldots, a_n \) of \( A \).

• In ordinary mathematical notation, we would write \( \langle a_1, \ldots, a_n \rangle \) element.
Remarks on Tuples of Length $n$

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

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

- If we define

\[ \text{nil} := \langle \rangle , \]
\[ \text{cons}(a_1, \langle a_2, \ldots, a_{n+1} \rangle) := \langle a_1, \ldots, a_{n+1} \rangle . \]

then this reads:

\[ \text{Vec}(A, 0) := \{ \text{nil} \} , \]
\[ \text{Vec}(A, n + 1) := \{ \text{cons}(a, b) \mid a \in A \land b \in \text{Vec}(A, n) \} . \]
Remarks on Tuples of Length n (Cont.)

• In the type theoretic definition we have constructors

  – nil :: Vec A Z
  – cons@(Vec A (S n)) :: A → Vec A n → Vec A (S n).

• This is the type theoretic analogue of the previous definition.
Example: Sum of Tuples of Length \( n \)

- Define

\[
\text{NVec} \quad (n :: N) \\
:: \quad \text{Set} \\
= \quad \text{Vec} \ N \ n
\]

- \text{Nvec} \ n \ are \ tuples \ of \ natural \ numbers \ of \ length \ n.
Example: Componentwise 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 $n$

SumNVec $\langle n :: N \rangle$

$\langle avec, bvec :: NVec n \rangle :: Nvec n$

= case $n$ of

  $\{ \langle \langle Z \rangle \to \text{nil;}(S n') \to$

  case $avec$ of

    $\{ \langle \langle \text{cons a avec'} \rangle \to$

    case $bvec$ of

      $\{ \langle \text{cons b bvec'} \rangle \to$

        cons@

        $(a + b)$

        (SumNVec $n'$ $a$) \rangle \}$ \}$ \}$ \}$
We define the set of lists of elements of type \( A \) in Agda.

We have two constructors:

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

So we define lists as:

\[
\text{list} \quad (A :: \text{Set}) \\
\quad :: \quad \text{Set} \\
\quad = \quad \text{data} \quad \text{nil} \\
\quad \quad | \quad \text{cons}(a :: A) \quad (l :: \text{list} \ A)
\]
Elimination Rule for Lists

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

Then we can define

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

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

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

\[
\text{length } (l :: \text{list } N) :: N = \text{case } l \text{ of }
\begin{cases}
(\text{nil}) & \rightarrow Z; \\
(\text{cons } a \ l') & \rightarrow S (\text{length } l')
\end{cases}
\]
Example: Sum of the Elements of a List

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

- Define
  \[ \text{append} : (A : \text{Set}) \to (\text{list} A) \to (\text{list} A) \to \text{list} A \]
s.t. \( \text{append} \ A \ l \ l' \) is the result of appending the list \( l' \) at the end of list \( l \).

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

  \[
  \text{append} \ A \ (\text{cons} a (\text{cons} b \ \text{nil})) \ (\text{cons} c (\text{cons} d \ \text{nil}))
  = \text{cons} a (\text{cons} b (\text{cons} c (\text{cons} d \ \text{nil})))
  \]
• A universe $U$ is a set, the elements of which are codes for sets.

• So we have
  – $U : \text{Set}$,
  – $T : U \rightarrow \text{Set}$ (the decoding function).

• We consider in the following a universe closed under
  – $\text{Fin}_0$, $\text{Fin}_1$, $\text{Bool}$,
  – $\mathbb{N}$,
  – $+$,
  – $\Sigma$,
  – the dependent function type.
Rules for the Universe

**Formation Rule**

\[ U : \text{Set} \]

\[ a : U \]

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

**Introduction and Equality Rules**

\[ \hat{\text{Fin}}_0 : U \quad T(\hat{\text{Fin}}_0) = \text{Fin}_0 : \text{Set} \]

\[ \hat{\text{Fin}}_1 : U \quad T(\hat{\text{Fin}}_1) = \text{Fin}_1 : \text{Set} \]

\[ \hat{\text{Bool}} : U \quad T(\hat{\text{Bool}}) = \text{Bool} : \text{Set} \]
Introduction/Equality Rules for the Universe (Cont.)

\[
\begin{align*}
  a : U & \quad b : U \\
  a \mathrel{\hat{+}} b : U
  \\
  T(a \mathrel{\hat{+}} b) &= T(a) + T(b) : \text{Set}
\end{align*}
\]

\[
\begin{align*}
  a : U & \quad b : T(a) \rightarrow U \\
  \widehat{\Sigma}(a, b) : U
  \\
  T(\widehat{\Sigma}(a, b)) &= \Sigma T(a) \ (\lambda x. T (b \ x)) : \text{Set}
\end{align*}
\]
Introduction/Equality Rules for the Universe (Cont.)

\[
\begin{array}{c}
\frac{a : U \quad b : T(a) \to U}{\hat{\Pi}(a, b) : U} \\
\end{array}
\]

\[T(\hat{\Pi}(a, b)) = (x : T(a)) \to T(bx) : \text{Set} \]
There exist as well elimination rules and corresponding equality rules for the universe.

They are very long (one step for each of constructor of $U$) and are not very much used.

They follow the principles present in previous rules.
Applications of the Universe

- Ordinary elimination rules don’t allow to eliminate into Set.

- However often, one can verify, that all sets needed are “elements of a universe”,
  - i.e. there are codes in the universe representing them.

- Then one can eliminate into the universe instead of Set and use $T$ to obtain the required function.
• Example: Define

\[
\widehat{\text{atom}} : \text{Bool} \rightarrow U , \\
\widehat{\text{atom}} := \text{Case}_{\text{Bool}} (\lambda (x : \text{Bool}). U) \widehat{\text{Fin}}_1 \widehat{\text{Fin}}_0 \\
\text{atom} : \text{Bool} \rightarrow \text{Set} , \\
\text{atom} : \lambda (x : \text{Bool}). T (\widehat{\text{atom}} x) ,
\]

Then

- \( \text{atom} \, \text{tt} = \text{Fin}_1 \),
- \( \text{atom} \, \text{ff} = \text{Fin}_0 \).
Universe in Agda

- $U$ and $T$ need to be defined simultaneously.
  - Usually Agda type checks definitions in sequence, so not possible.
  - Special construct `mutual`.
    * Everything in the scope of it is type checked simultaneously.
    * Scope determined by indentation.
Universes in Agda (Cont.)

mutual
U :: Set
= data Nhat
| Finzerohat
| Finonehat
| Boolhat
| Sigmahat (a :: U)(b :: T a → U)
| Piha (a :: U)(b :: T a → U)
Universes in Agda (Cont.)

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

\[
\begin{align*}
T \ (u :: U) & :: \ Set \\
= \ & \text{case } u \text{ of} \\
& \{ (\text{Nhat}) \to N; \\
& \quad (\text{Finzerohat}) \to \text{Finzero}; \\
& \quad (\text{Finonehat}) \to \text{Finone}; \\
& \quad (\text{Boolhat}) \to \text{Bool}; \\
& \quad (\text{Sigmahat} \ a \ b) \to \Sigma (T \ a) (\lambda x :: T \ a) \to T (b \ x) \\
& \quad (\text{Pihat} \ a \ b) \to (x :: T \ a) \to T (b \ x) \}
\end{align*}
\]
(i) Algebraic Data Types.

- The construct “data” in Agda is much more powerful than what is covered by type theoretic rules.

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

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

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

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

$$\rightarrow \ldots$$

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

$$\rightarrow A$$

where a constructor always constructs new elements.

• In other words the elements of $A$ are exactly those constructed by constructors.
Strictly Positive Algebraic Data Types

- In the types $A_{ij}$ we can make use of $A$.
  - However, it is difficult to understand $A$, if we have negative occurrences of $A$.
  - Example:
    \[
    A :: \text{Set} \\
    = \text{data } C (f :: A \to A)
    \]
  - What is the least set $A$ having a constructor
    \[
    C@A :: (f :: A \to A) \\
    \to A
    \]
Strictly Positive Algebraic Data Types (Cont.)

- If we
  * have constructed some part of $A$ already,
  * find a function $f : : A \to A$, and
  * add $C@f$ to $A$,
then $f$ might no longer be a function $A \to A$.
($f$ applied to the new element $C@f$ might not be defined).
- In fact, “agda-check-termination” issues a warning, if we
  - We shouldn’t make use of such definitions.
A “good” definition is the set of lists of natural numbers, defined as follows:

$$Nlist :: Set$$

$$= data \text{ nil } \ |
\text{ cons } (a:: \ N) \ |
(\text{ l :: } Nlist)$$

The constructor $$\text{ cons }$$ of N-lists refers to $$Nlist$$, but in a positive way: We have: if $$a :: N$$ and $$l :: Nlist$$, then we have $$\text{ cons } a l :: Nlist$$.

- If we add $$\text{ cons }$$ a l to $$Nlist$$, the reason for adding (i.e., $$l :: Nlist$$) is not destroyed by this addition.
- So we can “construct” the set $$Nlist$$ by
  * starting with the emptyset,
  * adding $$\text{ nil }$$ and
  * closing it under $$\text{ cons }$$ whenever possible.

Because we can “construct” $$Nlist$$, the above is an acceptable definition.
Strictly Positive Algebraic Data Types (Cont.)

- In general:

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

is a strictly positive algebraic data type, if all \( A_{ij} \) are

- either types which don’t make use of \( A \)
- or are \( A \) itself.

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

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

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

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

- This is a strictly positive data type.
Strictly Positive Algebraic Data Types

Extensions

- An often used extension is to define several sets simultaneously.

- Example: the even and odd numbers:

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

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

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

• Example (called “Kleene’s O”):

\[
O:: \text{Set} \\
\quad = \text{data leaf} \\
\quad \quad | \text{succ } (o::O) \\
\quad \quad | \lim (f :: \mathbb{N} \to O)
\]

• The last definition is unproblematic, since, if we have $f :: \mathbb{N} \to O$, we can construct $\lim@_f$ out of it, adding this new element to $O$ doesn’t destroy the reason for adding it to $O$.

• So again $O$ can be “constructed”.
Elimination Rules for data

- Functions 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 we have $f \left( C_{a_1 \cdots a_n} \right)$, we can refer only to $f$ applied to elements $a_1 \cdots a_n$. 
• For instance
  – in the Bintree example, when defining
    \[
    f :: \text{Bintree} \rightarrow A
    \]
    by case-distinction, then the definition of
    \[
    f (\text{branch} @ \text{left right})
    \]
    can make use of \( f \text{ left} \) and \( f \text{ right} \).
  – In the example of \( 0 \), when defining
    \[
    g :: 0 \rightarrow A
    \]
    by case-distinction, then the definition of
    \[
    g (\text{lim} @ f)
    \]
    can make use of \( g (f \text{ n}) \) for all \( n :: N \).