(C) Anton Setzer 2003 (except for pictures)

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 Σ-set.
(f) The set of natural numbers.
(g) Lists.
(h) Universes.
(i) Algebraic data types.

(C) Anton Setzer 2003 (except for pictures)

(a) The Set of Booleans

Formation Rule

$$\mathrm{Bool} : \mathrm{Set}$$

Introduction Rules

$$\mathrm{tt} : \mathrm{Bool} \qquad\qquad \mathrm{ff} : \mathrm{Bool}$$

Elimination Rule

$$\frac{C : \mathrm{Bool} \to \mathrm{Set} \quad ic : C\ \mathrm{tt} \quad ec : C\ \mathrm{ff} \quad cond}{\mathrm{Case}_{\mathrm{Bool}}\ C\ ic\ ec\ cond : C\ cond}$$

(C) Anton Setzer 2003 (except for pictures)

The Set of Booleans (Cont.)

Equality Rules

$$\frac{C : \mathrm{Bool} \to \mathrm{Set} \quad ic : C\ \mathrm{tt} \quad ec : C\ \mathrm{ff}}{\mathrm{Case}_{\mathrm{Bool}}\ C\ ic\ ec\ \mathrm{tt} = ic : C\ \mathrm{tt}}$$

$$\frac{C : \mathrm{Bool} \to \mathrm{Set} \quad ic : C\ \mathrm{tt} \quad ec : C\ \mathrm{ff}}{\mathrm{Case}_{\mathrm{Bool}}\ C\ ic\ ec\ \mathrm{ff} = ec : C\ \mathrm{ff}}$$

(C) Anton Setzer 2003 (except for pictures)

Remarks

- In the above
  - tt stands for true, ff stands for false.
  - ic stands for "if-case", ec for "else-case".
  - con for "condition".
- We can write the elimination rule in a **more compact** but

  - $$\begin{aligned}\text{Case}_{\text{Bool}} : (C : \text{Bool} \to \text{Set}) \\ \to (ic : C\ \text{tt}) \\ \to (ec : C\ \text{ff}) \\ \to (cond : \text{Bool}) \\ \to C\ cond\end{aligned}$$.

- tt, ff are the **constructors** of Bool.

Remarks (Cont.)

- Notice that we then get for $C : \mathrm{Bool} \to \mathrm{Set}$, $ic : C\ \mathrm{tt}, ec$
  - $f := \mathrm{Case}_{\mathrm{Bool}}\ C\ ic\ ec$,
    $: (cond : \mathrm{Bool}) \to C\ cond$
  - $f\ \mathrm{tt} = \mathrm{Case}_{\mathrm{Bool}}\ C\ ic\ ec\ \mathrm{tt} = ic : C\ \mathrm{tt}$,
  - $f\ \mathrm{ff} = \mathrm{Case}_{\mathrm{Bool}}\ C\ ic\ ec\ \mathrm{ff} = ec : C\ \mathrm{ff}$.
- So we obtain functions from Bool into other sets **withou**
  $\lambda(\mathbf{b} : \mathbf{Bool}). \cdots$.
- That's why we choose the argument to eliminate from as

Remarks (Cont.)

- This is similar to the definition of for instance $(+)$ in **curri**
  - $(+) : \mathrm{int} \rightarrow \mathrm{int} \rightarrow \mathrm{int}$.
  - $(+)\ 3$ is the function which takes an integer and adds t
    - **Shorter** than writing $\lambda x.3 + x$.

(C) Anton Setzer 2003 (except for pictures)

Remarks (Cont.)

- Note that we have the following **order of the arguments**
  - 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 $\mathrm{Case}_{\mathrm{Bool}}$ is a "$\mathrm{then}\ _\mathrm{else}\ _\mathrm{if}$ " – the **condi**
  last one.

(C) Anton Setzer 2003 (except for pictures)

Example

$$\begin{array}{rcl} \textbf{AND} & := & \lambda(b, c : \mathrm{Bool}).\mathrm{Case}_{\mathrm{Bool}}\ (\lambda(b' : \mathrm{Bool}).\mathrm{B} \\ & : & \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool} \end{array}$$

- AND is the **conjunction**:
  - $\mathrm{AND}\ \mathrm{tt}\ c = c$.
    Correct since $\mathrm{tt} \wedge c = c$.
  - $\mathrm{AND}\ \mathrm{ff}\ c = \mathrm{ff}$.
    Correct since $\mathrm{ff} \wedge c = \mathrm{ff}$.

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

(C) Anton Setzer 2003 (except for pictures)

Example (Cont.)

- Derivation of $\mathrm{AND} : \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}$:
  - First we derive $b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \lambda(b' : \mathbf{Bool}).\mathit{Bool}$

$$\mathit{Bool} : \mathrm{Set}$$
$$\mathbf{Bool} : \mathrm{Type}$$
$$b : \mathbf{Bool} \Rightarrow \mathrm{Context}$$
$$b : \mathbf{Bool} \Rightarrow \mathit{Bool} : \mathrm{Set}$$
$$b : \mathbf{Bool} \Rightarrow \mathbf{Bool} : \mathrm{Type}$$
$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{Context}$$
$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathit{Bool} : \mathrm{Set}$$
$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathbf{Bool} : \mathrm{Type}$$
$$b : \mathbf{Bool}, c : \mathbf{Bool}, b' : \mathbf{Bool} \Rightarrow \mathrm{Conte}$$
$$b : \mathbf{Bool}, c : \mathbf{Bool}, b' : \mathbf{Bool} \Rightarrow \mathit{Bool} :$$
$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \lambda(b' : \mathbf{Bool}).\mathit{Bool} : \mathbf{Bo}$$

(C) Anton Setzer 2003 (except for pictures)

Example (Cont.)

- We derive

$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathbf{Bool} = (\lambda(b' : \mathbf{Bool}).\mathit{Bool}$$

(using part of the derivation above):

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{b : \mathbf{Bool}, c : \mathbf{Bool}, b' : \mathbf{Bool} \Rightarrow \mathrm{Context}}}{b : \mathbf{Bool}, c : \mathbf{Bool}, b' : \mathbf{Bool} \Rightarrow \mathit{Bool} : \mathrm{Set}} \quad \dfrac{b : \mathbf{Bool}, c :}{b : \mathbf{Bool}, c :}}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow (\lambda(b' : \mathbf{Bool}).\mathit{Bool})\ \mathrm{tt} =}}{\dfrac{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathbf{Bool} = (\lambda(b' : \mathrm{Bool}).\mathit{Boo}}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathbf{Bool} = (\lambda(b' : \mathrm{Bool}).\mathit{Bool}}}$$

Example (Cont.)

- Similarly follows

$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathbf{Bool} = (\lambda(b' : \mathbf{Bool}).\mathit{Bool}$$

Example (Cont.)

– Using part of the proof above, we derive

$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow c : (\lambda(b' : \mathbf{Bool}).\mathit{Boo}$$

$$\dfrac{\dfrac{\dfrac{\cdots}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{Context}}}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow c : \mathbf{Bool}} \quad \dfrac{\cdots}{b{:}\mathbf{Bool}, c{:}\mathbf{Bool} \Rightarrow \mathbf{Bool} = (\lambda(}}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow c : (\lambda(b' : \mathbf{Bool}).\mathit{Bo}}$$

– We derive

$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{ff} : (\lambda(b' : \mathbf{Bool}).\mathit{Boo}$$

$$\dfrac{\dfrac{\dfrac{\cdots}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{Context}}}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{ff} : \mathbf{Bool}} \quad \dfrac{\cdots}{b{:}\mathbf{Bool}, c{:}\mathbf{Bool} \Rightarrow \mathbf{Bool} = (\lambda(}}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{ff} : (\lambda(b' : \mathbf{Bool}).\mathit{Bo}}$$

(C) Anton Setzer 2003 (except for pictures)

Example (Cont.)

- We derive $b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow b : \mathbf{Bool}$ using part of th

$$\frac{\begin{array}{c}\cdots \\ b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{Context}\end{array}}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow b : \mathbf{Bool}}$$

Example (Cont.)

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

$$\frac{\dfrac{\begin{array}{r} b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \lambda(b' : \mathbf{Bool}).\mathit{Bool} : \mathbf{B} \\ b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow c : (\lambda(b' : \mathbf{Bool}).\mathit{B} \\ b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{ff} : (\lambda(b' : \mathbf{Bool}).\mathit{} \\ b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow b : \mathbf{Bool} \end{array}}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{Case}_{\mathrm{Bool}}\ (\lambda(b' : \mathbf{Bool}).\mathit{Bo}}}{b : \mathbf{Bool} \Rightarrow \lambda(c : \mathbf{Bool}).\mathrm{Case}_{\mathrm{Bool}}\ (\lambda(b' : \mathbf{Bool}).\mathit{Bool})}$$

$$\overline{\lambda(b, c : \mathbf{Bool}).\mathrm{Case}_{\mathrm{Bool}}\ (\lambda(b' : \mathbf{Bool}).\mathit{Bool})\ c\ \mathrm{ff}\ b : \mathbf{Bo}}$$

(C) Anton Setzer 2003 (except for pictures)

Elimination into Type

We can extend add elimination and equality rules, havin

Elimination Rule into Type

$$\frac{C : \mathrm{Bool} \rightarrow \mathrm{Type} \quad ic : C\ \mathrm{tt} \quad ec : C\ \mathrm{ff} \quad cond}{\mathrm{Case}_{\mathrm{Bool}}\ C\ ic\ ec\ cond : C\ cond}$$

Equality Rules into Type

$$\frac{C : \mathrm{Bool} \rightarrow \mathrm{Type} \quad ic : C\ \mathrm{tt} \quad ec : C\ \mathrm{ff}}{\mathrm{Case}_{\mathrm{Bool}}\ C\ ic\ ec\ \mathrm{tt} = ic : C\ \mathrm{tt}}$$

$$\frac{C : \mathrm{Bool} \rightarrow \mathrm{Type} \quad ic : C\ \mathrm{tt} \quad ec : C\ \mathrm{ff}}{\mathrm{Case}_{\mathrm{Bool}}\ C\ ic\ ec\ \mathrm{ff} = ec : C\ \mathrm{ff}}$$

(C) Anton Setzer 2003 (except for pictures)

Elimination into Type (Cont.)

We can extend this into an elimination rule **into Kind or ot**

(C) Anton Setzer 2003 (except for pictures)

Bool in Agda

- We introduce Bool by simply **listing its constructors** (s
syntax):

$$\mathrm{data\ Bool} = \mathrm{tt} \mid \mathrm{ff}$$

- This introduces as well constants
  - $\mathrm{tt} :: \mathrm{Bool}$
  - $\mathrm{ff} :: \mathrm{Bool}$

- With this syntax, each constructor **can occur at most on**
  - i.e. we cannot define a second type having constructor
  e.g. for defining True (which is used later):

$$\mathrm{data\ True} = \mathrm{tt}$$

(C) Anton Setzer 2003 (except for pictures)

Bool in Agda (Cont.)

- The definition of Bool above is treated in Agda as an **ab** following three **more fundamental Agda definitions**:

$$
\begin{array}{rcl}
\mathrm{Bool} & :: & \mathrm{Set} \\
 & = & \mathrm{data\ tt} \mid \mathrm{ff} \\
\mathrm{tt} & : & \mathrm{Bool} \\
 & = & \mathrm{tt@Bool} \\
\mathrm{ff} & : & \mathrm{Bool} \\
 & = & \mathrm{ff@Bool}
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

Bool in Agda (Cont.)

- The definition of Bool as

$$\begin{array}{rcl} \mathrm{Bool} & :: & \mathrm{Set} \\ & = & \mathrm{data}\ \mathrm{tt} \mid \mathrm{ff} \end{array}$$

introduces Bool as a set **having constructors $\mathrm{tt@Bool}$**

– So tt, and ff **have to be defined separately**.
– If it is clear that the element in question is of type B
replace $\mathrm{tt@Bool}$ by $\mathrm{tt@_}$.
– The definition of Bool as above **doesn't prevent**
another set with constructors tt or ff.
– This syntax is the only one allowed, if one defines a s
keyword depending on arguments.
More about this later.

Bool in Agda (Cont.)

- **Internally**, tt will always be represented as $\mathrm{tt@Bool}$, simi
  - So Agda **evaluates tt to $\mathrm{tt@Bool}$**.
  - This can be seen when using for instance **"agda-c**
    compute weak head normal form.

Case Distinction

- Elimination in Agda is **based on case distinction**.
  - Assume we want to define
    - $f : \mathrm{Bool} \rightarrow \mathrm{Bool}$, s.t.
    - $f\ \mathrm{tt} = \mathrm{ff}$,
    - $f\ \mathrm{ff} = \mathrm{tt}$.
  - So we have the goal:

$$\begin{array}{lll} f & (x :: \mathrm{Bool}) & \\ & :: & \mathrm{Bool} \\ & = & \{!\ \ !\} \end{array}$$

(C) Anton Setzer 2003 (except for pictures)

Case Distinction (Cont.)

- We can then type into the goal x and choose the menu it
  - This **introduces a case distinction** by the constructor u x:
    x could have been introduced as tt or ff.

- The goal expands to:

$$
\begin{array}{lll}
f & (x :: \mathrm{Bool}) & \\
 & :: & \mathrm{Bool} \\
 & = & \mathrm{case}\ x\ \mathrm{of} \\
 & & \quad \{\ (\mathrm{tt}) \rightarrow \{!\ \ !\}; \\
 & & \quad\ \ \ (\mathrm{ff}) \rightarrow \{!\ \ !\};\}
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

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 essent **"compute the result of reducing that term"**.
      - More precisely this means that a term is reduced u 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 co**
  - (use goal-menu **"agda-context"**):

  It contains

  $x :: \mathrm{Bool} = \mathrm{tt}@_.$

- Similarly one finds that in the second goal x is $\mathrm{ff}@_$.

(C) Anton Setzer 2003 (except for pictures)

Case Distinction (Cont.)

- Now we can solve the new goals by inserting
  - ff into the first one,
  - tt into the second one.
- We obtain a function:

$$\begin{array}{lll} f & (x :: \mathrm{Bool}) & \\ & :: & \mathrm{Bool} \\ & = & \mathrm{case}\ x\ \mathrm{of} \\ & & \qquad \{\ (\mathrm{tt}) \rightarrow \mathrm{ff}; \\ & & \qquad\ \ \ (\mathrm{ff}) \rightarrow \mathrm{tt}; \} \end{array}$$

- $f\ x$ is the **negation of x**.

(C) Anton Setzer 2003 (except for pictures)

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 th
  - Not every place in the buffer is a good place.
  - **Good places for context are goals**, and that's **the** Agda allows us to **compute the weak head normal fo**

Testing the Defined Function

- So we
  - type in a dummy goal:

$$\begin{array}{lcl} test & :: & \mathrm{Set} \\ & = & \{! \quad !\} \end{array}$$

  - move to the new goal
  - choose **"agda-compute-WHNF"**,
  - and type into the mini-buffer f tt.
- The result shown is $\mathrm{ff@_}$.

(C) Anton Setzer 2003 (except for pictures)

(b) The Finite Sets

Bool can be generalized to sets having n elements (n a fixe

Formation Rule

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

Introduction Rules

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

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

Elimination Rule

$$\frac{\begin{array}{c} C : \mathrm{Fin}_n \longrightarrow \mathrm{Set} \\ s_0 : C\ \mathrm{A}^n_0 \\ s_1 : C\ \mathrm{A}^n_1 \\ \cdots \\ s_{n-1} : C\ \mathrm{A}^n_{n-1} \\ a : \mathrm{Fin}_n \end{array}}{\mathrm{Case}_n\ C\ s_0\ \ldots\ s_{n-1}\ a : C\ a}$$

(C) Anton Setzer 2003 (except for pictures)

The Finite Sets (Cont)

Equality Rules

$$\frac{\begin{array}{c} C : \mathrm{Fin}_n \longrightarrow \mathrm{Set} \\ s_0 : C\ \mathrm{A}^n_0 \\ s_1 : C\ \mathrm{A}^n_1 \\ \cdots \\ s_{n-1} : C\ \mathrm{A}^n_{n-1} \end{array}}{\mathrm{Case}_n\ C\ s_0\ \ldots\ s_{n-1}\ \mathrm{A}^n_k = s_k : C\ \mathrm{A}^n_k}$$

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

(C) Anton Setzer 2003 (except for pictures)

Omitting Premises in Equality Rules

Since the premises of the equality rule can in most cases b
the introduction and elimination rules, we will **usually omi**
for instance for the previous rule:

$$\mathrm{Case}_n\ C\ s_0\ \ldots\ s_{n-1}\ \mathrm{A}^n_k = s_k : C\ \mathrm{A}^n_k$$

We sometimes even **omit the type**:

$$\mathrm{Case}_n\ C\ s_0\ \ldots\ s_{n-1}\ \mathrm{A}^n_k = s_k$$

(C) Anton Setzer 2003 (except for pictures)

More compact elimination rules

- $$\begin{aligned}\mathrm{Case}_n : (C : \mathrm{Fin}_n \to \mathrm{Set}) \\ \to (s_0 : C\ \mathrm{A}^n_0) \\ \to \cdots \\ \to (s_{n-1} : C\ \mathrm{A}^n_{n-1}) \\ \to (a : \mathrm{Fin}_n) \\ \to C\ a\end{aligned}$$

.

(C) Anton Setzer 2003 (except for pictures)

Elimination into Type

- Similarly as for Bool we can write down **elimination rule** $\mathbf{C : Fin_n \to Type}$ (instead of $C : \mathrm{Fin}_n \to \mathrm{Set}$).

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

(C) Anton Setzer 2003 (except for pictures)

Rules for True

True is the special case Fin_n for $n = 1$:

Formation Rule

$$\mathrm{True} : \mathrm{Set}$$

Introduction Rules

$$\mathrm{true} : \mathrm{True}$$

Elimination Rule

$$\frac{C : \mathrm{True} \to \mathrm{Set} \quad c : C\ \mathrm{true} \quad t : \mathrm{True}}{\mathrm{Case}_{\mathrm{True}}\ c\ t : C\ t}$$

Equality Rule

$$\frac{C : \mathrm{True} \to \mathrm{Set} \quad c : C\ \mathrm{true}}{\mathrm{Case}_{\mathrm{True}}\ c\ \mathrm{true} = c : C\ \mathrm{true}}$$

(C) Anton Setzer 2003 (except for pictures)

Rules for True (Cont.)

- $\mathrm{Case}_{\mathrm{True}}$ is **computationally not very interesting**.
  - $\mathrm{Case}_{\mathrm{True}}\ c$ is the untyped function $\lambda x.c$.
  - However, in Agda we might not be able to derive

$$\lambda(t : \mathrm{True}).c : (t : \mathrm{True}) \to C\ t$$

- From a **logic point of view**, it expresses:
  From an element of $C\ \mathrm{true}$ we obtain an element of $C\ t$ **f**
  - So there is no $C : \mathrm{True} \to \mathrm{Set}$ s.t. $C\ \mathrm{true}$ is inhabit
    inhabited for some other $x : \mathrm{True}$.
  - This means that all elements of x of type True are **indis**
    true, i.e. they are **identical to true**.
  - This equality is called Leibnitz equality).

(C) Anton Setzer 2003 (except for pictures)

Rules for False

False is the special case Fin_n for $n = 0$:

Formation Rule

$$\mathrm{False} : \mathrm{Set}$$

There is no Introduction Rule

Elimination Rule

$$\frac{C : \mathrm{False} \to \mathrm{Set} \qquad f : \mathrm{False}}{\mathrm{Case}_{\mathrm{False}}\ f : C\ f}$$

There is no Equality Rule

n

False

- False has **no elements**.
- It is formula which is **always false**.
  - As well called **absurdity**.
- $\mathrm{Case}_{\mathrm{False}}$ expresses: **from an element f of False we o**
  of any set (which might depend on f).
  - From a logic point of view this is **"Ex falsum quo**
    absurdity follows anything).
    E.g. A **false formula** like "$0 = 1$" or "Swansea lies in
    everything.

(C) Anton Setzer 2003 (except for pictures)

False (Cont.)

- $\mathrm{Case}_{\mathrm{False}}$ has **no computational meaning**, since there is
be applied to.
  - Applies of course only if we are working in a **terminati**
  - If we had **full recursion**, we could define $f : \mathrm{False}$ by f
    However that f doesn't reduce to canonical form.
  - That's why it's important to carry out the **terminatio**
    otherwise one obtains for instance elements of False

(C) Anton Setzer 2003 (except for pictures)

Finite Sets in Agda

- **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 $\text{red} :: \text{Colour}$
  - And we can define for instance

$$\begin{array}{lll} \text{is_red} & (c :: \text{Colour}) & \\ & :: & \text{Bool} \\ & = & \text{case } c \text{ of} \\ & & \quad \{ \; (\text{red}) \;\; \rightarrow \;\; \text{tt}; \\ & & \quad\;\;\; (\text{green}) \;\; \rightarrow \;\; \text{ff}; \\ & & \quad\;\;\; (\text{blue}) \;\; \rightarrow \;\; \text{ff}; \} \end{array}$$

False in Agda

- In Agda we can define the empty set as a "data"-set **with**

$$\text{data False} =$$

- If we want to solve

$$\begin{array}{lll} g & (x :: \text{False}) \\ & :: & \text{Bool} \\ & = & \{! \quad !\} \end{array}$$

we can insert into the goal x and choose menu-item **"ag**

(C) Anton Setzer 2003 (except for pictures)

False in Agda (Cont.)

- The result is

$$\begin{array}{lll} g & (x :: \mathrm{False}) & \\ & :: & \mathrm{Bool} \\ & = & \mathrm{case}\ x\ \mathrm{of}\ \{\ \ \} \end{array}$$

- If we make case distinction on x there is **no case to ch**
don't have to define anything.

(C) Anton Setzer 2003 (except for pictures)

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

$$\text{IsOak} :: \text{Tree} \to \text{Set}$$

s.t.

$$\begin{array}{rcl} \text{IsOak pine} & = & \text{False} \\ \text{IsOak oak} & = & \text{True} \end{array}$$

Example for the Use of False (Cont.)

- If we want to define a function from trees, which are oak set, we can do so by requiring **an additional argument** '

$$
\begin{array}{lll}
f & (t :: \mathrm{Tree}) & \\
  & (p :: \mathrm{IsOak\ t}) & \\
  & :: \quad A & \\
  & = \quad \mathrm{case}\ t\ \mathrm{of} & \\
  & \qquad \{\ \mathrm{pine} \ \rightarrow\ \mathrm{case}\ p\ \ \mathrm{of}\ \{\ \ \} & \\
  & \qquad \quad \mathrm{oak} \ \rightarrow\ \cdots ; \} &
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

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 expresse
    know this.
- Note that we **don't have to invent a result** of f in case

(C) Anton Setzer 2003 (except for pictures)

Example 2 for the Use of False

- Similarly we can introduce a **stack**, together with a predic

$$\text{NotEmpty} :: \text{Stack} \rightarrow \text{Set}$$

s.t.

$$\text{NotEmpty}\ s = \text{False}$$

if s is the empty stack.

- Now we can define

$$\begin{array}{ll} \text{pop} & (s :: \text{Stack}) \\ & (p :: \text{NotEmpty}\ s) \\ & ::\quad \text{Stack} \\ & =\quad \cdots \end{array}$$

- Again we **don't have to provide a result, in case s is e**

(C) Anton Setzer 2003 (except for pictures)

True in Agda

- The definition of True in Agda is **straightforward**:

$$\text{data False} = \text{true}$$

- Case distinction will require to **solve the case true**:

$$\begin{array}{lll} g & (x :: \text{True}) & \\ & :: & \text{Bool} \\ & = & \text{case } x \text{ of } \{(\text{true}) \rightarrow \{! \quad !\};\} \end{array}$$

(C) Anton Setzer 2003 (except for pictures)

(c) 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 proval

Truth in type theory means provability

Atomic Formulae (Cont.)

- False.
  * False is **not inhabited**.
    - There is **no proof** of False.
      Furthermore, from any proof of False we can
      (elimination rules for False).
    - False is therefore **type-theoretically false**:
      A formula is **type-theoretically false**, if from
      everything.
    - Since this implies that we can derive False and f
      derive everything, this is equivalent to the following
    - A formula is **type-theoretically false**, if from a
      derive False (i.e. a **contradiction**).

(C) Anton Setzer 2003 (except for pictures)

Atomic Formulae (Cont.)

- There are formulae in type theory, which are **neither type-** **nor type-theoretically false**.
  - This means that we can neither prove them, nor deri contradiction.
  - Truth in type theory means that we **know that it is tr**
  - Falsity in type theory means that we **know that it can**
  - There are formulae in type theory for which **neither of**
- True and False as above are formulae corresponding to **true and false**.

(C) Anton Setzer 2003 (except for pictures)

atom

- We can **map** truth values to their corresponding formula:

$$
\begin{aligned}
\mathrm{atom} &: \mathrm{Bool} \rightarrow \mathrm{Set} \\
\mathrm{atom}\ \mathrm{tt} &= \mathrm{True} \\
\mathrm{atom}\ \mathrm{ff} &= \mathrm{False}
\end{aligned}
$$

- This can be defined using **case distinction**.

(C) Anton Setzer 2003 (except for pictures)

atom (Cont.)

- This corresponds to the following rules (which are not nee

$$\frac{b : \mathrm{Bool}}{\mathrm{atom}\ b : \mathrm{Set}}$$

$$\mathrm{atom\ tt} = \mathrm{True}$$
$$\mathrm{atom\ ff} = \mathrm{False}$$

(C) Anton Setzer 2003 (except for pictures)

atom in Agda

$$
\begin{array}{llll}
\text{atom} & (b :: \text{Bool}) & & \\
 & :: \quad \text{Set} & & \\
 & = \quad \text{case } b \text{ of} & & \\
 & \qquad\qquad \{ \ (\text{tt}) & \rightarrow & \text{True;} \\
 & \qquad\qquad \phantom{\{} \ (\text{ff}) & \rightarrow & \text{False;} \}
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

Decidable Predicates

- Using atom we can now define **decidable predicates** on

- 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 $f : A \to \mathrm{Bool}$.
  - E.g. $f\ a$ means: **state a is safe**.

(C) Anton Setzer 2003 (except for pictures)

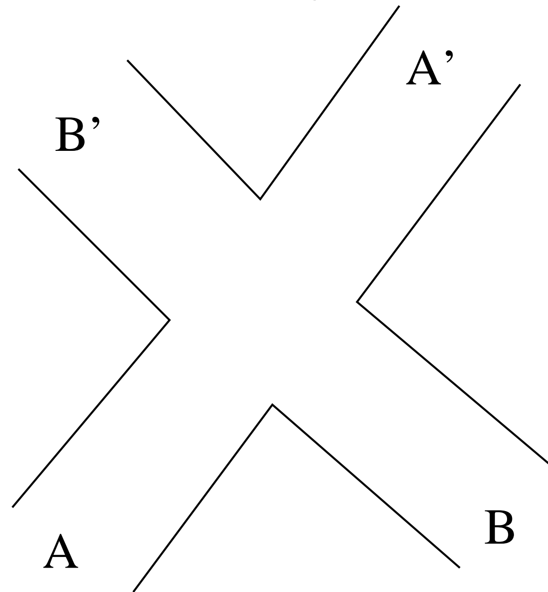
Decidable Predicates (Cont.)

- Let now $g : A \rightarrow \mathrm{Set}$, $g\ a = \mathrm{atom}(f\ a)$.
  - 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**.

(C) Anton Setzer 2003 (except for pictures)

The Traffic Light Example

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



- Assume from each direction A, A', B, B' there is one traf
  - but A and A' always coincide, similarly B and B'.

(C) Anton Setzer 2003 (except for pictures)

The Set of Physical States

- For simplicity assume that **each traffic light is either re**

$$\text{data Colour} = \text{red} \mid \text{green}$$

- The set of **physical states of the system** is given by a pa
  colour of A (and therefore as well A') and of B (and B')

$$\begin{array}{lll} \text{Phys_State} & :: & \text{Set} \\ & = & \text{sig}\{ \begin{array}{lll} \text{sigA} & :: & \text{Colour;} \\ \text{sigB} & :: & \text{Colour;} \end{array} \end{array}$$

The Set of Control States

- The set of **control states** is a set of states of the system, 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.

(C) Anton Setzer 2003 (except for pictures)

The Set of Control States (Cont.)

- We therefore define

$$\text{data Control_State} = \text{Allred} \mid \text{Agreen} \mid \text{Bg}$$

(C) Anton Setzer 2003 (except for pictures)

Mapping Control States to Physical Sta

- We define the **state of signals A, B depending on a co**

$$
\begin{array}{lll}
\mathrm{toSigA} & (s :: \mathrm{Control_State}) & \\
 & :: & \mathrm{Colour} \\
 & = & \mathrm{case}\ s\ \mathrm{of} \\
 & & \quad \{\ (\mathrm{Allred}) \ \longrightarrow\ \mathrm{red}; \\
 & & \quad\ \ \ (\mathrm{Agreen}) \ \longrightarrow\ \mathrm{green} \\
 & & \quad\ \ \ (\mathrm{Bgreen}) \ \longrightarrow\ \mathrm{red}; \}
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{toSigB} & (s :: \mathrm{Control_State}) & \\
 & :: & \mathrm{Colour} \\
 & = & \mathrm{case}\ s\ \mathrm{of} \\
 & & \quad \{\ (\mathrm{Allred}) \ \longrightarrow\ \mathrm{red}; \\
 & & \quad\ \ \ (\mathrm{Agreen}) \ \longrightarrow\ \mathrm{red}; \\
 & & \quad\ \ \ (\mathrm{Bgreen}) \ \longrightarrow\ \mathrm{green}
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

Mapping Control States to Physical Sta

- Now we can define the **physical state corresponding to**

$$
\begin{array}{lll}
\mathrm{phys_state} & (s :: \mathrm{Control_State}) & \\
 & :: & \mathrm{Phys_State} \\
 & = & \mathrm{struct}\{\ \mathrm{sigA} \ = \ \mathrm{toSigA} \\
 & & \qquad\qquad \mathrm{sigB} \ = \ \mathrm{toSigB}
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

Safety Predicate

- We define now **when a physical state is safe**:
  - It is **safe iff not both signals are green**.
  - We define now a corresponding predicate **directly**, with
    Boolean function.
  - We first define a predicate depending on two signals:

$$
\begin{array}{lll}
\mathrm{CorAux} & (a :: \mathrm{Colour}) & \\
 & (b :: \mathrm{Colour}) & \\
 & :: \quad \mathrm{Set} & \\
 & = \quad \mathrm{case}\ a\ \mathrm{of} & \\
 & \qquad \{\ (\mathrm{red}) \quad \longrightarrow \quad \mathrm{True}; & \\
 & \qquad\ \ \ (\mathrm{green}) \quad \longrightarrow \quad \mathrm{case}\ b\ \mathrm{of} & \\
 & \qquad\qquad\qquad \{\ (\mathrm{red}) & \\
 & \qquad\qquad\qquad\ \ \ (\mathrm{green}) &
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

Safety Predicate (Cont.)

– Now we define

$$\begin{array}{lll} \mathrm{Cor} & (s :: \mathrm{Phys_State}) & \\ & :: & \mathrm{Set} \\ & = & \mathrm{CorAux}\ s.\mathrm{sigA}\ s.\mathrm{sigB} \end{array}$$

- **Remark:** In some cases in order to define a function from s
a sig-set) into some other set, it is better first to **intro
function**, depending on the components of that product.
  – In the current example this wouldn't have caused prob
  complex examples it does (due to the lack of the η-rule

(C) Anton Setzer 2003 (except for pictures)

Safety of the System

- Now we show that **all control states are safe**:

$$
\begin{array}{lll}
\text{cor_proof} & (s :: \text{Control_State}) & \\
 & :: \quad \text{Cor}(\text{phys_state}\ s) & \\
 & = \quad \text{case}\ s\ \text{of} & \\
 & \qquad \{\ (\text{Allred}) & \rightarrow\ \text{true} \\
 & \qquad\ \ (\text{Agreen}) & \rightarrow\ \text{true} \\
 & \qquad\ \ (\text{Bgreen}) & \rightarrow\ \text{true}
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

Safety of the System (Cont.)

- The first element true was an element of **Cor(phys_sta** reduces to **True**.
- Similarly for the other two elements.
- This works only because **each control state correspo** **physical state**.
  - If this hadn't been the case, we would have gotten ir 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 situatio
  (e.g. sets toSigB Agreen = green, then in the last step w
  of type False.
  - Then we can't solve this goal directly and **cannot prove**
  - (In fact we could type-theoretically solve this goal by us
    (e.g. solve this goal as **cor_proof Agreen**), but this
    by the termination check.)

(C) Anton Setzer 2003 (except for pictures)

(d) The Disjoint Union of Sets

Formation Rule

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set}}{A + B : \mathrm{Set}}$$

Introduction Rules

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set} \qquad a : A}{\mathrm{inl}\ A\ B\ a : A + B}$$

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set} \qquad b : B}{\mathrm{inr}\ A\ B\ b : A + B}$$

Elimination Rule

$$\frac{\begin{array}{c} A : \mathrm{Set} \\ B : \mathrm{Set} \\ C : (A + B) \to \mathrm{Set} \\ sl : (a : A) \to C\ (\mathrm{inl}\ A\ B\ a) \\ sr : (b : B) \to C\ (\mathrm{inr}\ A\ B\ b) \\ d : A + B \end{array}}{\mathrm{Plus_Split}\ A\ B\ C\ sl\ sr\ d : C\ d}$$

(C) Anton Setzer 2003 (except for pictures)

The Disjoint Union of Sets (Cont.)

Equality Rules

$$
\begin{array}{lcl}
\text{Plus_Split}\ A\ B\ C\ sl\ sr\ (\text{inl}\ A\ B\ a) & = & sl\ a : C\ (\text{i} \\
\text{Plus_Split}\ A\ B\ C\ sl\ sr\ (\text{inr}\ A\ B\ b) & = & sr\ b : C\ (\text{i}
\end{array}
$$

Disjoint Union using the Logical Framework

- A **more compact notation** is:
  - $(+) : \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set}$, written infix.
  - $\mathrm{inl} : (A, B : \mathrm{Set}) \to A \to (A + B)$.
  - $\mathrm{inr} : (A, B : \mathrm{Set}) \to B \to (A + B)$.
  - $\mathrm{Plus_Split} : (A, B : \mathrm{Set})$

$$
\begin{aligned}
&\to (C : (A + B) \to \mathrm{Set})\\
&\quad \to (sl : (a : A) \to C\ (\mathrm{inl}\ A\ B\ a))\\
&\quad\quad \to (sr : (b : B) \to C\ (\mathrm{inr}\ A\ B\ b))\\
&\quad\quad\quad \to (d : A + B)\\
&\quad\quad\quad\quad \to C\ d\ \ .
\end{aligned}
$$

(C) Anton Setzer 2003 (except for pictures)

Disjoint Union in Agda

- The disjoint union can be defined as a "data"-set having inl (in-left) and inr (inright):

$$
\begin{array}{ll}
(+) & (A :: \mathrm{Set}) \\
& (B :: \mathrm{Set}) \\
& :: \quad \mathrm{Set} \\
& = \quad \mathrm{data}\ \mathrm{inl}(a :: A) \mid \mathrm{inr}(b :: B)
\end{array}
$$

Disjoint Union in Agda (Cont.)

- The notation $(+)$ means, that $+$ can be used **infix**.
- Now we have, if $A, B :: \mathrm{Set}$:
  - $\mathrm{inl}@(A + B) :: A \to (A + B)$
  - $\mathrm{inr}@(A + B) :: B \to (A + B)$
  - This can be checked using the menu "**agda-infer-type**"
  - Note that we cannot assign a type to $\mathrm{inr}@_$.
  - $(+)$ **cannot** be defined using the abbreviated data nota
    (which would be of the form
    $\mathrm{data}\,(+) = \cdots$).

(C) Anton Setzer 2003 (except for pictures)

Disjoint Union in Agda (Cont.)

- Elimination is again represented by case distinction.
  So if want to define for $A, B :: \mathrm{Set}$ for instance

$$\begin{array}{lll} f & (c :: A + B) \\ & :: & \mathrm{Bool} \\ & = & \{! \quad !\} \end{array}$$

  we can type into the goal c and choose menu "agda-case"

(C) Anton Setzer 2003 (except for pictures)

Disjoint Union in Agda (Cont.)

- We obtain

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

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

(C) Anton Setzer 2003 (except for pictures)

Use of Concrete Disjoint Sets

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

$$\text{data Plant} = \text{tree}(t :: \text{Tree}) \mid \text{flower}(f :: \text{Fl}$$

- Now one can define for instance

$$\begin{array}{lll} \text{isFlower} & (p :: \text{Plant}) & \\ & :: & \text{Bool} \\ & = & \text{case } p \text{ of} \\ & & \quad \{ \ (\text{tree } t) \quad \rightarrow \quad \text{ff}; \\ & & \quad \ \ \ (\text{flower } f) \quad \rightarrow \quad \text{tt}; \end{array}$$

(C) Anton Setzer 2003 (except for pictures)

(e) The Σ-Set

Formation Rule

$$\frac{A : \mathrm{Set} \qquad B : A \to \mathrm{Set}}{\Sigma\ A\ B : \mathrm{Set}}$$

Introduction Rule

$$\frac{\begin{array}{c} A : \mathrm{Set} \\ B : A \to \mathrm{Set} \\ a : A \\ b : B\ a \end{array}}{\mathrm{p}\ A\ B\ a\ b : \Sigma\ A\ B}$$

Elimination Rule

$$\frac{\begin{array}{c} A : \mathrm{Set} \\ B : A \to \mathrm{Set} \\ C : (\Sigma\ A\ B) \to \mathrm{Set} \\ s : (a : A) \to (b : B\ a) \to C\ (\mathrm{p}\ A\ B\ a\ b)) \\ d : \Sigma\ A\ B \end{array}}{\mathrm{Sigma_Split}\ A\ B\ C\ s\ d : C\ d}$$

(C) Anton Setzer 2003 (except for pictures)

The Σ-Set (Cont)

Equality Rule

$$\mathrm{Sigma_Split}\ A\ B\ C\ s\ (\mathrm{p}\ A\ B\ a\ b) = s\ a\ b : C\ (\mathrm{p}\ \ldots$$

(C) Anton Setzer 2003 (except for pictures)

The Σ-Set using the Logical Framewor

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

The Σ-Set using the Logical Framework (C

- Sigma_Split

$$
\begin{aligned}
&: (A : \mathrm{Set}) \\
&\quad \rightarrow (B : A \rightarrow \mathrm{Set}) \\
&\qquad \rightarrow (C : (\Sigma\ A\ B) \rightarrow \mathrm{Set}) \\
&\qquad\quad \rightarrow (s : (a : A, b : B\ a) \\
&\qquad\qquad\qquad \rightarrow C\ (\mathrm{p}\ A\ B\ a\ b)) \\
&\qquad\qquad \rightarrow (d : \Sigma\ A\ B) \\
&\qquad\qquad\quad \rightarrow C\ d\ \ .
\end{aligned}
$$

(C) Anton Setzer 2003 (except for pictures)

The Σ-Set and the Dependent Produc

- The **dependent product and the Σ-set are very simila**
  - Both have similar introduction rules (for the Σ-set, the additional arguments A, B necessary for bureaucratic r
  - One can define the projections π_0, π_1 using $\mathrm{Sigma_Spl}$

$$\begin{array}{lcl}\pi_0 & = & \mathrm{Sigma_Split}\ A\ B\ (\lambda x.A)\ (\lambda(x : A).\lambda(y :\\ \pi_1 & = & \mathrm{Sigma_Split}\ A\ B\ (\lambda x.B\ \pi_0(x))\ (\lambda(x : A)\end{array}$$

  - On the other hand, from π_0, π_1 we can define $\mathrm{Sigma_S}$

$$\lambda A, B, C, s, d.s\ \pi_0(d)\ \pi_1(d)\ .$$

The Σ-Set and the Dependent Produc

- However the dependent product has the η-**rule** (whic implemented in Agda).
- Because of the lack of η-rule, Σ works usually **better tha product** in Agda.
  - I personally **don't use the dependent product** of Agd

(C) Anton Setzer 2003 (except for pictures)

The Σ-Set in Agda

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

$$\begin{array}{lll} \text{Sigma} & (A :: \text{Set}) & \\ & (B :: A \rightarrow \text{Set}) & \\ & :: & \text{Set} \\ & = & \text{data}\ p\ (a :: A)\ (b :: B a) \end{array}$$

The Σ-Set in Agda (Cont.)

- Again one usually defines concrete Σ-sets more directly.
- **Example:** Assume we have defined
  - a set $\mathrm{Plant_Group}$ for **groups of plants** (e.g. "tree",
  - depending on $g :: \mathrm{Plant_Group}$, sets $\mathrm{Plants_in_group}$
    plants in that group.
- The **set of plants** can then be defined as

$$\mathrm{data\ Plant} = \mathrm{plant}\ (g :: \mathrm{Plant_Group})(pg :: \mathrm{Plants}$$

The Σ-Set in Agda (Cont.)

- Not surprisingly, for **elimination** we use **case distinction**

$$
\begin{array}{lll}
f & (p :: \mathrm{Plant}) & \\
 & :: & \mathrm{Plant_group} \\
 & = & \mathrm{case}\ p\ \mathrm{of} \\
 & & \quad \{\ (\mathrm{plant}\ g\ pg) \ \rightarrow\ g; \}
\end{array}
$$

Formulae in Dependent Type Theo

- We have seen how to represent **atomic decidable formu**
- Now treatment of complex formulae constructed using **log**

Conjunction

- $\mathbf{A} \wedge \mathbf{B}$ is true iff both A is true and B is true.
- Therefore a proof of $A \wedge B$ consists of a **proof of A and**
  - It is therefore a pair $\langle p, q\rangle$ consisting of a proof p of A a
- Therefore the set of proofs of $A \wedge B$ is the set of pairs of B, i.e. $\mathbf{A} \times \mathbf{B}$.
- We can **identify** $A \wedge B$ with $A \times B$.

(C) Anton Setzer 2003 (except for pictures)

Conjunction (Cont.)

- With this identification, the **introduction rule** for $\wedge$ allo of $A \wedge B$ from a proof of A and a proof of B:

$$\frac{p : A \qquad q : B}{\langle p, q \rangle : A \wedge B}$$

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

- This is what is expressed by the **ordinary introduction ru**

$$\frac{A \qquad B}{A \wedge B}$$

Conjunction (Cont.)

- The **elimination rule** for $\wedge$ allows to project a proof of A
  A and a proof of B:

$$\frac{p : A \wedge B}{\pi_0(p) : A} \qquad \frac{p : A \wedge B}{\pi_1(p) : B}$$

- This means that we can **derive from A $\wedge$ B both A anc**

- This is what is expressed by the **ordinary elimination rul**

$$\frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B}$$

(C) Anton Setzer 2003 (except for pictures)

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** **plus the information which one.**
  - It is therefore an element $\mathrm{inl}\, p$ for a proof $p : A$ or an proof $q : B$.

- Therefore the set of proofs of $A \lor B$ is the **disjoint unio** $\mathbf{A + B}$.

- We can **identify** $A \lor B$ with $A + B$.

Disjunction (Cont.)

- With this identification, the **introduction rules** for $\vee$ allo of $A \vee B$ from a proof of A or from a proof of B.

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set} \qquad p : A}{\mathrm{inl}\ A\ B\ p : A + B} \qquad \frac{A : \mathrm{Set} \qquad B}{\mathrm{inr}\ A\ B\ p}$$

  – Omitting the premises $A, B : \mathrm{Set}$ and omitting them a and inr (which is needed only for bureaucratic reasons)

$$\frac{p : A}{\mathrm{inl}\ p : A + B} \qquad \frac{p : B}{\mathrm{inr}\ p : A + B}$$

(C) Anton Setzer 2003 (except for pictures)

Disjunction (Cont.)

- This means that we can **derive A ∨ B from A and from**
- This is what is expressed by the **ordinary introduction ru**

$$\frac{A}{A \vee B} \qquad \frac{B}{A \vee B}$$

Disjunction (Cont.)

- The **elimination rule** for $+$ allows to form from an ele[...] element of any set C provided we can compute such an el[...] from B:

$$\frac{\begin{array}{c} A : \mathrm{Set} \\ B : \mathrm{Set} \\ C : (A \vee B) \rightarrow \mathrm{Set} \\ sl : (a : A) \rightarrow C\ (\mathrm{inl}\ A\ B\ a) \\ sr : (b : B) \rightarrow C\ (\mathrm{inr}\ A\ B\ b) \\ d : A \vee B \end{array}}{\mathrm{Plus_Split}\ A\ B\ C\ sl\ sr\ d : C\ d}$$

- Omitting the dependency of C on $A \vee B$ and omittin[...] premises and arguments A, B and C we get:

$$\frac{d : A \vee B \qquad sl : A \rightarrow C \qquad sr : B \rightarrow [...]}{\mathrm{Plus_Split}\ sl\ sr\ d : C}$$

Disjunction (Cont.)

- This means that we can **derive from A $\vee$ B a formula C**
  C from A and from B.

- This is what is expressed by the **ordinary elimination rul**

$$\frac{A \vee B \quad \begin{matrix} A \\ \vdots \\ C \end{matrix} \quad \begin{matrix} B \\ \vdots \\ C \end{matrix}}{C}$$

- (Note that in the ordinary elimination rule, from the pre
  from A" we obtain "$A \to C$", similarly for "C derivabl
  $B \to C$.)

(C) Anton Setzer 2003 (except for pictures)

Implication

- We write temporarily $\supset$ for logical implication, in order to
  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 pr**
- Therefore a proof of $A \supset B$ is a **function, which takes**
  computes a proof of B.
- Therefore the set of proofs of $A \supset B$ is the **function typ**
- We can **identify** $A \supset B$ with $A \rightarrow B$.

Implication (Cont.)

- With this identification, the **introduction rule for $\supset$** allo 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 prov**
  - (i.e. we can make use of a context $p : A$ for proving q :

  then we can derive A $\supset$ B without assuming p:A.

Implication (Cont.)

- This is what is expressed by the **ordinary introduction ru**

$$\frac{\begin{array}{c}A\\ \vdots\\ B\end{array}}{A \supset B}$$

(C) Anton Setzer 2003 (except for pictures)

Implication (Cont.)

- The **elimination rule for $\supset$** allows to apply a proof p of
of q of A in order to obtain a proof of B:

$$\frac{p : A \supset B \qquad q : A}{p\,q : B}$$

- This means that we can **derive from A $\supset$ B and A that**

- This is what is expressed by the **ordinary elimination rul**

$$\frac{A \supset B \qquad A}{B}$$

(C) Anton Setzer 2003 (except for pictures)

Negation

- $\neg \mathbf{A}$ has the same meaning as $\mathbf{A} \supset \bot$ (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 abs
    cannot be a proof of A, A must be false.
- Therefore we can identify $\neg \mathbf{A}$ with $\mathbf{A} \to \mathbf{False}$.

(C) Anton Setzer 2003 (except for pictures)

Universal Quantification

- Since we have many types, we have to write when using q
the type, the bound variable is ranging over:
We write therefore $\mathbf{\forall x : A.B}$, $\mathbf{\exists x : A.B}$.

- $\forall x : A.B$ 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 t**
computes an element of B.

- Therefore the set of proofs of $\forall x : A.B$ is the **depende**
$\mathbf{(x : A) \to B}$.

- We can **identify** $\mathbf{\forall x : A.B}$ with $\mathbf{(x : A) \to B}$.

Universal Quantification (Cont.)

- With this identification, the **introduction rule** for $\forall$ allo
of $\forall x : A.B$ from a proof of B depending on an element

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

- This means that, if we, **from x:A can prove B**, then
$\forall x : A.B$ which doesn't depend on $x : A$.

Universal Quantification (Cont.)

- This is what is expressed by the **ordinary introduction ru**

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

where

– **x might not occur free in any assumption of the p**
  * This is guaranteed in type theory, since $x : A$ must b
    of the context, so any other assumptions must be loc
    can therefore **not depend on x:A**.

– The **conclusion will no longer depend on free varial**
  * This corresponds in type theory to the fact that **x:/**
    occur in the context of the conclusion.

Universal Quantification (Cont.)

- The **elimination rule** for the dependent function type allo
  p of $\forall x : A.B$ to an element $a : A$ in order to obtain a pro

$$\frac{p : \forall x : A.B \qquad a : A}{p\, a : B[x := a]}$$

- This means that we can **derive from ∀x:A.B and an ele**
  B[x:=a] holds.

(C) Anton Setzer 2003 (except for pictures)

Universal Quantification (Cont.)

- This is what is expressed by the **ordinary elimination rul**
  - For the simple languages used in ordinary logic, there i[s] that $a : A$; in more **complex type theories we have** **derivation**.

$$\frac{\forall x : A.B \qquad a : A}{B[x := a]}$$

Existential Quantification

- $\exists x : A.B$ is true iff there exists an $a : A$ such that $B[x :=$
- Therefore a proof of $\exists x : A.B$ is a **pair** $\langle \mathbf{a}, \mathbf{p} \rangle$ **consisti** $\mathbf{a} : \mathbf{A}$ **and a proof p of** $\mathbf{B}[\mathbf{x} := \mathbf{a}]$.
- Therefore the set of proofs of $\exists x : A.B$ is the **dependent** $(\mathbf{x} : \mathbf{A}) \times \mathbf{B}$.
- We can **identify** $\exists \mathbf{x} : \mathbf{A}.\mathbf{B}$ with $(\mathbf{x} : \mathbf{A}) \times \mathbf{B}$.

(C) Anton Setzer 2003 (except for pictures)

Existential Quantification (Cont.)

- With this identification, the **introduction rule** for $\exists$ allo
of $\exists x : A.B$ from an element $a : A$ and a proof $p : B[x :=$

$$\frac{a : A \qquad p : B[x := a]}{\langle a, p \rangle : \exists x : A.B}$$

- This is what is expressed by the **ordinary introduction ru**

$$\frac{a : A \qquad B[x := a]}{\exists x : A.B}$$

(C) Anton Setzer 2003 (except for pictures)

Existential Quantification (Cont.)

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

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

- From this we can derive a rule which is essentially th deduction (in which one doesn't have explicit proofs):
  - Assume:
    * $C : \mathrm{Set}$, which does not depend on $x : A$,
    * $p : \exists x : A.B$ and
    * $x : A, y : B \Rightarrow c : C$.
  - Then we have $\mathbf{c[x := \pi_0(p), y := \pi_1(p)] : C}$, **not depending on x:A or y:B**.

(C) Anton Setzer 2003 (except for pictures)

Existential Quantification (Cont.)

- Therefore the **rule in natural deduction** follows from rules:

$$\dfrac{\exists x : A.B \qquad \begin{matrix} x : A \\ B \\ \vdots \\ C \end{matrix}}{C}$$

where the conclusion does not depend on $x : A$ and B.

Constructive (or Intuitionistic) Logic

- From type theoretic proofs we can **directly extract progr**
- 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 $\pi_1(p\ x) : C(x,$
  - Therefore $f := \lambda x : A.\pi_0(p\ x)$ is a **function** $\mathbf{A \to B}$, a

$$\lambda(\mathbf{x} : \mathbf{A}).\pi_1(\mathbf{p}\ \mathbf{x}) : \forall \mathbf{x} : \mathbf{A}.\mathbf{C}(\mathbf{x}, \mathbf{f}\ \mathbf{x})$$

  i.e. we have a proof that $\forall \mathbf{x} : \mathbf{A}.\mathbf{C}(\mathbf{x}, \mathbf{f}\ \mathbf{x})$ **holds**.
  - Therefore, from a proof of $\forall x : A.\exists y : B.C(x, y)$, **function**, which computes the y from the x.

(C) Anton Setzer 2003 (except for pictures)

Constructive Logic (Cont.)

- We can derive as well a function which **depending on p**
whether p= inl(a) or p = inr(b).

- Therefore we can decide, from a proof of a disjunctio
disjuncts holds.

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

$$\forall \mathbf{x} : \mathbf{Turing_Machine}.(\mathbf{x\ halts} \vee \neg(\mathbf{x\ h}$$

Constructive Logic (Cont.)

- In classical logic we **can prove the above**, since we ca
(tertium non datur) for any formula A.

- In type theory, this law **cannot hold**, unless we don't wan
can be evaluated.

  – The logic of type theory is **intuitionistic (constructiv**
  $A \vee \neg A$ and $\neg\neg A \rightarrow A$ don't hold for all formulae A.

(C) Anton Setzer 2003 (except for pictures)

Constructive Logic (Cont.)

- In **classical logic**,
  - **$\exists x : A.B$ is equivalent to $\neg\forall x : A.\neg B$**,
  - **$A \vee B$ is equivalent to $\neg(\neg A \wedge \neg B)$**.

- If we take decidable atomic formulae only and replace $\exists x$ by the above formulae, then **all formulas provable in c derivable**.
  - This requires $(\neg\neg A) \to A$, which can be shown for all f decidable atomic formulae using $\neg$, $\to$, $\wedge$, $\forall$.
  - The formula $A \vee \neg A$ translates into $\neg(\neg A \wedge \neg\neg A)$, w 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**

(C) Anton Setzer 2003 (except for pictures)

Constructive Logic (Cont.)

- Proof (using classical logic) of

$$\exists \mathbf{x} : \mathbf{A}.\mathbf{B} \leftrightarrow \neg \forall \mathbf{x} : \mathbf{A}.\neg \mathbf{B} \quad :$$

  - We have classically:

$$\neg\neg \mathbf{A} \rightarrow \mathbf{A} \quad :$$

    * If A is true, then $\neg\neg A \rightarrow A$ holds.
    * If A is false, then $\neg\neg A$ is false, therefore $\neg\neg A \rightarrow A$

(C) Anton Setzer 2003 (except for pictures)

Constructive Logic (Cont.)

- We show intuitionistically $\neg(\exists \mathbf{x} : \mathbf{A}.\mathbf{B}) \leftrightarrow \forall \mathbf{x} : \mathbf{A}.\neg\mathbf{B}$:
  - Assume $\neg(\exists x : A.B)$, $x : A$ and show $\neg B$.
    If we had B, then we had $\exists x : A.B$, contradicting $\neg(\exists x$
    $\neg B$.
  - 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):

$$(\exists \mathbf{x} : \mathbf{A}.\mathbf{B}) \leftrightarrow \neg\neg(\exists \mathbf{x} : \mathbf{A}.\mathbf{B}) \leftrightarrow \neg\forall \mathbf{x} : \mathbf{A}.$$

Constructive Logic (Cont.)

- Proof of $\mathbf{A} \vee \mathbf{B} \leftrightarrow \neg(\neg\mathbf{A} \wedge \neg\mathbf{B})$:
  - We show intuitionistically $\neg(\mathbf{A} \vee \mathbf{B}) \leftrightarrow (\neg\mathbf{A} \wedge \neg\mathbf{B})$:
    - Assume $\neg(A \vee B)$. If A then $A \vee B$, a contradiction
      Similarly we get $\neg B$, therefore $\neg A \wedge \neg B$.
    - Assume $\neg A \wedge \neg B$, show $\neg(A \vee B)$.
      Assume $A \vee B$. If A then a contradiction with $\neg A$, s
  - Now it follows (classically):

$$(\mathbf{A} \vee \mathbf{B}) \leftrightarrow \neg\neg(\mathbf{A} \vee \mathbf{B}) \leftrightarrow \neg(\neg\mathbf{A} \wedge \neg$$

Constructive Logic (Cont.)

- **Weak disjunction and existential quantification** is
formulae $\neg(\neg A \land \neg B)$ and $\neg\forall x : A.\neg B$.
  - When using only weak disjunction, existential quantifica
atomic formulae, we obtain classical logic.

- **Strong disjunction and existential quantificaton** is
original type theoretic formulae.

(C) Anton Setzer 2003 (except for pictures)

(f) The Set of Natural Numbers

- The set N is the type theoretic representation of the set $\mathbb{N}$
- $\mathbb{N}$ 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.

(C) Anton Setzer 2003 (except for pictures)

The Set of Natural Numbers (Cont.)

- Let S be a type theoretic notation for the operation $x \mapsto$
- Then the type theoretic rules are

$$\mathrm{N} : \mathrm{Set}$$

$$0 : \mathrm{N}$$

$$\frac{n : \mathrm{N}}{\mathrm{S}\ n : \mathrm{N}}$$

(C) Anton Setzer 2003 (except for pictures)

Primitive Recursion

- **Primitive Recursion expresses:**
  Assume we have
  - $a : \mathrm{N}$.
  - and, if $n : \mathrm{N}$, $x : \mathrm{N}$ then $g\ n\ x : \mathrm{N}$.

  Then we can define $\mathrm{f} : \mathbf{N} \rightarrow \mathbf{N}$, s.t.
  - $f\ 0 = a$,
  - $f\ (\mathrm{S}\ n) = g\ n\ (f\ n)$.

(C) Anton Setzer 2003 (except for pictures)

Primitive Recursion (Cont.)

- The **computation of $\mathbf{f}$ $\mathbf{n}$** proceeds now as follows:
  - Compute n.
  - If $n = 0$, then the result is a.
  - Otherwise $n = \mathrm{S}(n')$.
    - ∗ We assume that we have determined already how to
    - ∗ Now $f\ n$ reduces to $g\ n'\ (f\ n')$.
    - ∗ $g\ n'\ (f\ n')$ can be computed, since we know how to
      - · g
      - · $f\ n'$.

Example

- The function $f : \mathrm{N} \to \mathrm{N}$ with $f(x) = 2 \cdot x$ can be **recursively** by:
  - $f(0) = 0$.
  - $f(\mathrm{S}\ n) = \mathrm{S}\ (\mathrm{S}\ (f\ n))$.
- Therefore take in the definition above:
  - $a = 0$,
  - $g\ n\ x = \mathrm{S}\ (\mathrm{S}\ x)$.

Generalized Primitive Recursion

- We can **generalize primitive recursion** as follows:
  - First we can **replace the range of f by an arbitrary s**
    - i.e. we allow for any set C

$$f : \mathrm{N} \to C$$

  - Further, C can now **depend on N**.
- We obtain the following set of rules:

(C) Anton Setzer 2003 (except for pictures)

Rules for the Natural Numbers

Formation Rule

$$\mathrm{N} : \mathrm{Set}$$

Introduction Rules

$$0 : \mathrm{N}$$

$$\frac{n : \mathrm{N}}{\mathrm{S}\ n : \mathrm{N}}$$

Elimination Rule

$$\frac{\begin{array}{c} C : \mathrm{N} \to \mathrm{Set} \\ a : C\ 0 \\ f : (x : \mathrm{N}) \to C\ x \to C\ (\mathrm{S}\ x) \\ n : \mathrm{N} \end{array}}{\mathrm{P}\ C\ a\ f\ n : C\ n}$$

Equality Rules

$$\mathrm{P}\ C\ a\ f\ 0 = a$$

$$\mathrm{P}\ C\ a\ f\ (\mathrm{S}\ n) = f\ n\ (\mathrm{P}\ C\ a\ f\ n)$$

Rules for the Natural Numbers (Cont.)

- Note that if we define in the elimination rule $g := \mathrm{P}\ C\ f$
  - The conclusion of the elimination rule reads:

$$g\ n : C\ n$$

which means that

$$\lambda(n : \mathrm{N}).g\ n : (n : \mathrm{N}) \to C\ n\ \ .$$

  - The equality rules read:

$$\begin{array}{rcl} g\ 0 & = & a \\ g\ (\mathrm{S}\ n) & = & f\ n\ (g\ n) \end{array}$$

Rules for N using the Logical Framework

- The more compact notation is:
  - $\mathrm{N} : \mathrm{Set}$,
  - $0 : \mathrm{N}$,
  - $S : \mathrm{N} \to \mathrm{N}$,
  - $\mathrm{P} \ : (C : \mathrm{N} \to \mathrm{Set}) \to C\ 0 \to ((x : \mathrm{N}) \to C\ x \to C\ (S\ x)) \to (n : \mathrm{N}) \to C\ n$.

Natural Numbers in Agda

- N is defined using **data**:

$$\mathrm{data\ N} = \mathrm{Z} \mid \mathrm{S}(n :: N)$$

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

- Therefore we have

$$\begin{array}{lll} \mathrm{Z} & :: & \mathrm{N} \\ \mathrm{S} & :: & \mathrm{N} \rightarrow \mathrm{N} \end{array}$$

Elimination Rules for N in Agda

- Elimination works via case distinction in Agda.
  - If we want to introduce

$$\begin{array}{ll} f & (n :: \mathrm{N}) \\ & :: \quad A \\ & = \quad \{! \quad !\} \end{array}$$

    * A possibly depending on n,

    we can type into the goal n and use the menu agda-cas
    We get

$$\begin{array}{ll} f & (n :: \mathrm{N}) \\ :: & A \\ = & \text{case } n \text{ of} \\ & \quad \begin{array}{llll} \{ & (\mathrm{Z}) & \rightarrow & \{! \quad !\}; \\ & (\mathrm{S}\ n') & \rightarrow & \{! \quad !\}; \} \end{array} \end{array}$$

Elimination Rules for N in Agda (Cont

- 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-**

- If we

  – **do not make use of f in the case n=Z** and
  – **only use of $\mathrm{f}\ \mathrm{n}'$ in case n $= \mathrm{S}\ \mathrm{n}'$**.

  then **agda-check-termination succeeds**.

(C) Anton Setzer 2003 (except for pictures)

Elimination Rules for N in Agda (Cont

- If **agda-check-termination succeeds**, the definition shou
  - (The lecturer hasn't checked the algorithm).
- However, **if agda-check-termination fails**, the **definiti correct**.

(C) Anton Setzer 2003 (except for pictures)

Example of the Power of Termination Ch

- The following definition of the **Fibonacci numbers** can't
directly using the rules of type theory, but it **can be de**
follows and **agda-check-termination accepts it**:
(one := S Z):

$$
\begin{array}{lll}
\mathrm{fib} & (n :: N) & \\
 & :: & \mathrm{N} \\
 & = & \mathrm{case}\ n\ \mathrm{of} \\
 & & \quad \{\ (\mathrm{Z}) \rightarrow \mathrm{one}; \\
 & & \quad\ \ (\mathrm{S}\ n') \rightarrow \mathrm{case}\ n'\ \mathrm{of} \\
 & & \qquad \{\ (\mathrm{Z}) \rightarrow \mathrm{one}; \\
 & & \qquad\ \ (\mathrm{S}\ n'') \rightarrow \mathrm{fib}\ n'
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

Example for Limitations of Termination C

- Assume we define the **predecessor function**

$$\begin{array}{lll} \mathrm{pred} & (n :: \mathrm{N}) & \\ & :: & \mathrm{N} \\ & = & \mathrm{case}\ n\ \mathrm{of} \\ & & \quad \{\ (\mathrm{Z}) \quad \rightarrow \quad \mathrm{Z}; \\ & & \quad\ \ (\mathrm{S}\ n') \quad \rightarrow \quad n';\} \end{array}$$

i.e

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

(C) Anton Setzer 2003 (except for pictures)

Example for Limitations of Termination Check

- Then the function

$$\begin{array}{lll} f & (n :: \mathrm{N}) & \\ & :: & \mathrm{N} \\ & = & \mathrm{case}\ n\ \mathrm{of} \\ & & \quad \{\ (\mathrm{Z}) \quad \rightarrow \quad \mathrm{Z}; \\ & & \quad \quad (\mathrm{S}\ n') \quad \rightarrow \quad f\ (\mathrm{pred}\ n); \} \end{array}$$

  terminates always

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

- However, **agda-check-termination fails**.

Limitations of the Termination Check (Co

- Because of the **undecidability of the Turing halting pro**
  - it is undecidable whether a recursively defined function
- there is no **extension of agda-check-termination, whic all in agda definable functions, which terminate for a**

Example: Addition

- Definition of $+$ in Agda:

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

- The definition expresses:

$$\begin{array}{rcl} n + 0 & = & n \\ n + (m' + 1) & = & (n + m') + 1 \end{array}$$

Example: Addition

- Note that $(+)$ is used **infix**, i.e. we write $n+m$ for $(+)\ n$
- If $m = Sm'$, the definition of $(+)\ n\ m$ refers to $(+)\ n\ m'$
  - $(+)\ n\ m'$ is **defined before** $(+)\ \mathbf{n}\ \mathbf{m}$ since $\mathbf{m}'$ **is intro**

(C) Anton Setzer 2003 (except for pictures)

Example: Multiplication

- Definition

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

- The definition expresses:

$$
\begin{array}{rcl}
n \cdot 0 & = & 0 \\
n \cdot (m' + 1) & = & (n \cdot m') + n
\end{array}
$$

Example: Multiplication (Cont.)

- Again $*$ is **treated infix**.
- Agda has built in that $*$ **binds more than** $+$.
  - $n * m' + n$ is treated as $(n * m') + n$.
- Note that the definition of $*$ requires, that $+$ **is already**

(C) Anton Setzer 2003 (except for pictures)

Equality on N

- The **equality (n == m) ::Set** for $n, m :: \mathrm{N}$ can be equations:
  - $(\mathrm{Z} == \mathrm{Z}) = \mathrm{True}.$
  - $(\mathrm{Z} == \mathrm{S}\ n) = (\mathrm{S}\ n == \mathrm{Z}) = \mathrm{False}.$
  - $(\mathrm{S}\ n == \mathrm{S}\ m) = (n == m).$

(C) Anton Setzer 2003 (except for pictures)

Equality on N (Cont.)

- From this one can now derive a definition in Agda:

$$
\begin{array}{lll}
(==) & (n, m :: \mathrm{N}) & \\
 & :: \quad \mathrm{Set} & \\
 & = \quad \mathrm{case}\ n\ \mathrm{of} & \\
 & \qquad \{ \ (\mathrm{Z}) \quad \rightarrow \quad \mathrm{case}\ m\ \mathrm{of} & \\
 & \qquad\qquad\qquad \{ \ (\mathrm{Z}) \quad \rightarrow & \\
 & \qquad\qquad\qquad\quad (\mathrm{S}\ m') \quad \rightarrow & \\
 & \qquad\quad (\mathrm{S}\ n') \quad \rightarrow \quad \mathrm{case}\ m\ \mathrm{of} & \\
 & \qquad\qquad\qquad \{ \ (\mathrm{Z}) \quad \rightarrow & \\
 & \qquad\qquad\qquad\quad (\mathrm{S}\ m') \quad \rightarrow &
\end{array}
$$

- Task of coursework 3, Question 1 to fill in those goals.

Reflexivity of ==

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

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

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

$$\begin{array}{lll} \mathrm{refl} & (n : \mathrm{N}) & \\ & :: & n == n \\ & = & \{! \quad !\} \end{array}$$

(C) Anton Setzer 2003 (except for pictures)

Reflexivity of == (Cont.)

- This can now be shown using **case distinction**:

$$
\begin{array}{lll}
\mathrm{refl} & (n : \mathrm{N}) & \\
 & :: & n == n \\
 & = & \mathrm{case}\ n\ \mathrm{of} \\
 & & \quad \{\ (\mathrm{Z}) \quad \rightarrow \quad \{!\ \ !\}; \\
 & & \quad\ \ \ (\mathrm{S}\ n') \quad \rightarrow \quad \{!\ \ !\};\}
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

Reflexivity of == (Cont.)

- Case $n = \mathrm{Z}$ is trivial.
- Case $n = \mathrm{S}\ n'$ can be solved using $\mathrm{refl}\ n'$ (which is define
- Task of Coursework 3, Question 1 (e) to solve this goal.

(C) Anton Setzer 2003 (except for pictures)

Symmetry of ==

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

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

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

$$\begin{array}{ll} \mathrm{sym} & (n, m : \mathrm{N}) \\ & (p :: n == m) \\ & :: \quad m == n \\ & = \quad \{! \quad !\} \end{array}$$

(C) Anton Setzer 2003 (except for pictures)

Symmetry of == (Cont.)

- This can now be shown using **case distinction**:

$$
\begin{array}{lll}
\text{sym} & (n, m : \mathrm{N}) & \\
 & (p :: n == m) & \\
 & :: \quad m == n & \\
 & = \quad \text{case } n \text{ of} & \\
 & \quad \{ \ (\mathrm{Z}) \quad \rightarrow \quad \text{case } m \text{ of} & \\
 & \qquad\qquad \{ \ (\mathrm{Z}) \quad \rightarrow \quad \{ & \\
 & \qquad\qquad \ \ (\mathrm{S}\ m') \quad \rightarrow \quad \{ & \\
 & \quad\ \ (\mathrm{S}\ n') \quad \rightarrow \quad \text{case } m \text{ of} & \\
 & \qquad\qquad \{ \ (\mathrm{Z}) \quad \rightarrow \quad \{ & \\
 & \qquad\qquad \ \ (\mathrm{S}\ m') \quad \rightarrow \quad \{ &
\end{array}
$$

Symmetry of == (Cont.)

- The **first goal** can be solved by using true (since $(\mathrm{Z} ==$
- For the **second goal** we know p is an element of $\mathrm{Z} == \mathrm{S}$
  - Therefore if we make **case distinction on p** we get

$$\mathrm{case}\ p\ \ \mathrm{of}\ \{\ \ \}$$

    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 $\mathbf{S\ m' == S\ n}$
  to $\mathbf{m' == n'}$.
  - The type of p is $\mathrm{S}\ n' == \mathrm{S}\ m'$ which is identical to n'
- The goal can be solved by using $\mathrm{sym}\ n'\ m'\ p$.
  - Note that we can **use here p** since it is of **type $\mathbf{n' ==}$**
    * It is correct to use it since **$\mathbf{n'}$ is introduced before**
      · Therefore **$\mathbf{sym\ n'}$ can be defined before $\mathbf{sym\ n}$**.
    * This definition will be **accepted by agda-check-ter**

(C) Anton Setzer 2003 (except for pictures)

Example: Tuples (or Vectors) of Length

- Define first

$$\begin{array}{rcl} \text{data Nil} & = & \text{nil} \\ \\ \text{Cons} & (A, B :: \text{Set}) \\ & :: & \text{Set} \\ & = & \text{data cons}(a :: A)(b :: B) \end{array}$$

(C) Anton Setzer 2003 (except for pictures)

Example: Tuples (or Vectors) of Length

- Now we can define (we use Vec for vector)

$$
\begin{array}{lll}
\mathrm{Vec} & (A :: \mathrm{Set}) & \\
 & (n :: N) & \\
 & :: \quad \mathrm{Set} & \\
 & = \quad \mathrm{case}\ n\ \mathrm{of} & \\
 & \qquad \{\ (\mathrm{Z}) & \rightarrow \quad \mathrm{Nil}; \\
 & \qquad\ \ (\mathrm{S}\ m') & \rightarrow \quad \mathrm{Cons}\ A\ (\mathrm{Vec}\ A
\end{array}
$$

Tuples of Length n

- Therefore (with the obvious definition of two),

$$\mathrm{Vec}\ A\ n = \underbrace{\mathrm{Cons}\ A\ (\mathrm{Cons}\ A\ \cdots (\mathrm{Cons}\ A}_{n\ \text{times}}\,\mathrm{Nil})$$

- The elements of $\mathrm{Vec}\ A\ n$ are

$$\mathrm{cons}\ a_1\ (\mathrm{cons}\ a_2\ \cdots (\mathrm{cons}\ a_n\ \mathrm{nil}) \cdots)$$

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

- In ordinary mathematical notation, we would write $\langle a_1, .$
  element.

(C) Anton Setzer 2003 (except for pictures)

Remarks on Tuples of Length n

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

$$
\begin{aligned}
\mathrm{Vec}(A,0) &:= \{\langle\rangle\} \ , \\
\mathrm{Vec}(A,n+1) &:= \{\langle a_1,\ldots,a_{n+1}\rangle \quad | \ a_1,\ldots,a_n
\end{aligned}
$$

(C) Anton Setzer 2003 (except for pictures)

Remarks on Tuples of Length n

- If we define

$$\begin{aligned} \mathrm{nil} &:= \langle\rangle \ , \\ \mathrm{cons}(a_1, \langle a_2, \ldots, a_{n+1}\rangle) &:= \langle a_1, \ldots, a_{n+ \end{aligned}$$

then this reads:

$$\begin{aligned} \mathrm{Vec}(A, 0) &:= \{\mathrm{nil}\} \ , \\ \mathrm{Vec}(A, n+1) &:= \{\mathrm{cons}(a, b) \quad \mid a \in A \wedge b \in \mathrm{Ve} \end{aligned}$$

(C) Anton Setzer 2003 (except for pictures)

Remarks on Tuples of Length n (Cont.

- In the type theoretic definition we have **constructors**
  - $\mathrm{nil} :: \mathrm{Vec}\ A\ \mathrm{Z}$
  - $\mathrm{cons@(Vec}\ A\ (\mathrm{S}\ n)) :: A \to \mathrm{Vec}\ A\ n \to \mathrm{Vec}\ A\ (\mathrm{S}\ n)$.
- This is the **type theoretic analogue** of the previous defin

(C) Anton Setzer 2003 (except for pictures)

Example: Sum of Tuples of Length n

- Define

$$\begin{array}{lll} \mathrm{NVec} & (n :: N) & \\ & :: & \mathrm{Set} \\ & = & \mathrm{Vec}\ \mathrm{N}\ n \end{array}$$

- $\mathrm{Nvec}\ n$ are tuples of natural numbers of length n.

Example: Componentwise Sum of Tuples of L

- We define **component-wise sum of tuples of length n**.
  - Using mathematical notation, this sum for instance as t

$$\langle 2, 3, 4\rangle + \langle 5, 6, 7\rangle = \langle 7, 9, 11\rangle \ .$$

(C) Anton Setzer 2003 (except for pictures)

Example: Componentwise Sum of Tuples of Leng

$$
\begin{array}{lll}
\text{SumNVec} & (n :: \text{N}) & \\
 & (avec, bvec :: \text{NVec}\ n) & \\
 & :: \quad \text{Nvec}\ n & \\
 & = \quad \text{case}\ n\ \text{of} & \\
 & \qquad \{\ (\text{Z}) \quad \rightarrow \quad \text{nil}; & \\
 & \qquad\ \ \ (\text{S}\ n') \quad \rightarrow & \\
 & \qquad\quad \text{case}\ avec\ \text{of} & \\
 & \qquad\qquad \{\ (\text{cons}\ a\ avec') \quad \rightarrow & \\
 & \qquad\qquad\quad \text{case}\ bvec\ \text{of} & \\
 & \qquad\qquad\qquad \{\ (\text{cons}\ b\ bvec') \quad - & \\
 & \qquad\qquad\qquad\quad \text{cons@_} & \\
 & \qquad\qquad\qquad\quad (a + b) & \\
 & \qquad\qquad\qquad\quad (\text{SumNVec}\ n'\ a &
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

(g) Lists

- We define the set of lists of elements of type `A` in Agda.
- We have two constructors:
  - nil, generating the empty list.
  - cons, adding an element of `A` in front of a list
- So we define lists as:

$$\begin{array}{lll} \mathrm{list} & (A :: \mathrm{Set}) & \\ & :: & \mathrm{Set} \\ & = & \mathrm{data}\ \mathrm{nil} \\ & & \qquad \mid \mathrm{cons}(a :: A)\ (l :: \mathrm{list}\ A) \end{array}$$

Elimination Rule for Lists

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

  Then we can define

$$\begin{array}{llllll} f & (l :: \mathrm{list}\ A) & & & & \\ & :: & C & & & \\ & = & \mathrm{case}\ l\ \mathrm{of} & & & \\ & & & \{\ (\mathrm{nil}) & \rightarrow & \{!\ \ !\}; \\ & & & \ \ (\mathrm{cons}\ a\ l') & \rightarrow & \{!\ \ !\};\} \end{array}$$

  and in the second goal we can make use of $f\ l'$.

(C) Anton Setzer 2003 (except for pictures)

Example: Length of a List

$$
\begin{array}{lll}
\text{length} & (l :: \text{list N}) & \\
 & :: & \text{N} \\
 & = & \text{case } l \text{ of} \\
 & & \quad \{ \; (\text{nil}) \quad \rightarrow \quad \text{Z}; \\
 & & \quad\;\;\; (\text{cons } a\ l') \quad \rightarrow \quad \text{S (length } \ldots
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

Example: Sum of the Elements of a Li

$$
\begin{array}{lll}
\text{sumlist} & (l :: \text{list N}) & \\
 & :: & \text{N} \\
 & = & \text{case } l \text{ of} \\
 & & \quad \{ \ (\text{nil}) \quad \longrightarrow \quad \text{Z}; \\
 & & \quad\ \ \ (\text{cons } n\ l') \quad \longrightarrow \quad n + \text{sumlis}
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

Interesting Exercise

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

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

(C) Anton Setzer 2003 (except for pictures)

(h) Universes.

- A universe U is a set, the elements of which are codes for

- So we have
  - $\mathrm{U} : \mathrm{Set}$,
  - $\mathrm{T} : \mathrm{U} \rightarrow \mathrm{Set}$ (the decoding function).

- We consider in the following a universe closed under
  - Fin_0, Fin_1, Bool,
  - N,
  - $+$,
  - Σ,
  - the dependent function type.

(C) Anton Setzer 2003 (except for pictures)

Rules for the Universe

Formation Rule

$$\mathrm{U} : \mathrm{Set}$$

$$\frac{a : \mathrm{U}}{\mathrm{T}\ a : \mathrm{Set}}$$

Introduction and Equality Rules

$$\widehat{\mathrm{Fin}_0} : \mathrm{U} \qquad \mathrm{T}(\widehat{\mathrm{Fin}_0}) = \mathrm{Fin}_0 : \mathrm{Set}$$

$$\widehat{\mathrm{Fin}_1} : \mathrm{U} \qquad \mathrm{T}(\widehat{\mathrm{Fin}_1}) = \mathrm{Fin}_1 : \mathrm{Set}$$

$$\widehat{\mathrm{Bool}} : \mathrm{U} \qquad \mathrm{T}(\widehat{\mathrm{Bool}}) = \mathrm{Bool} : \mathrm{Set}$$

(C) Anton Setzer 2003 (except for pictures)

Introduction/Equality Rules for the Universe (Cont.)

$$\frac{a : \mathrm{U} \qquad b : \mathrm{U}}{a \mathbin{\widehat{+}} b : \mathrm{U}}$$

$$\mathrm{T}(a \mathbin{\widehat{+}} b) = \mathrm{T}(a) + \mathrm{T}(b) : \mathrm{Set}$$

$$\frac{a : \mathrm{U} \qquad b : \mathrm{T}(a) \rightarrow \mathrm{U}}{\widehat{\Sigma}(a, b) : \mathrm{U}}$$

$$\mathrm{T}(\widehat{\Sigma}(a, b)) = \Sigma\ \mathrm{T}(a)\ (\lambda x.\mathrm{T}\ (b\ x)) : \mathrm{Set}$$

(C) Anton Setzer 2003 (except for pictures)

Introduction/Equality Rules for the Universe (Cont.)

$$\frac{a : \mathrm{U} \qquad b : \mathrm{T}(a) \to \mathrm{U}}{\widehat{\Pi}(a, b) : \mathrm{U}}$$

$$\mathrm{T}(\widehat{\Pi}(a, b)) = (x : \mathrm{T}(a)) \to \mathrm{T}\ (b\ x) : \mathrm{Set}$$

Elimination and Equality Rules for the Universe (Cont.)

- There exist as well elimination rules and corresponding ec
universe.
- They are very long (one step for each of constructor of U
much used.
- They follow the principles present in previous rules.

(C) Anton Setzer 2003 (except for pictures)

Applications of the Universe

- Ordinary elimination rules don't allow to eliminate into Se
- However often, one can verify, that all sets needed a universe",
  - i.e. there are codes in the universe representing them.
- Then one can eliminate into the universe instead of Set a the required function.

(C) Anton Setzer 2003 (except for pictures)

Applications of the Universe

- Example: Define

$$
\begin{array}{lcl}
\widehat{\text{atom}} & : & \text{Bool} \to \text{U} \quad , \\
\widehat{\text{atom}} & := & \text{Case}_{\text{Bool}}\ (\lambda(x : \text{Bool}).\text{U})\ \widehat{\text{Fin}_1}\ \widehat{\text{F}
\end{array}
$$

$$
\begin{array}{lcl}
\text{atom} & : & \text{Bool} \to \text{Set} \quad , \\
\text{atom} & : & \lambda(x : \text{Bool}).\text{T}\ (\widehat{\text{atom}}\ x) \quad ,
\end{array}
$$

Then

- $\text{atom tt} = \text{Fin}_1$,
- $\text{atom ff} = \text{Fin}_0$.

Universes in Agda

- U and T need to be defined simultaneously.
  - Usually Agda type checks definitions in sequence, so no
    definitions possible.
  - Special construct `mutual`.
    * Everything in the scope of it is type checked simultar
    * Scope determined by indentation.

(C) Anton Setzer 2003 (except for pictures)

Universes in Agda (Cont.)

$$
\begin{array}{lll}
\text{mutual} & & \\
\quad \text{U} & :: & \text{Set} \\
 & = & \text{data Nhat} \\
 & & \qquad \mid \text{Finzerohat} \\
 & & \qquad \mid \text{Finonehat} \\
 & & \qquad \mid \text{Boolhat} \\
 & & \qquad \mid \text{Sigmahat } (a :: \text{U})(b :: \text{T } a \rightarrow \\
 & & \qquad \mid \text{Pihat } (a :: \text{U})(b :: \text{T } a \rightarrow \text{U})
\end{array}
$$

Universes in Agda (Cont.)

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

$$
\begin{array}{lll}
\mathrm{T} & (u :: \mathrm{U}) & \\
& :: & \mathrm{Set} \\
& = & \text{case } u \text{ of} \\
& & \quad \{ \begin{array}{lll}
(\mathrm{Nhat}) & \rightarrow & N; \\
(\mathrm{Finzerohat}) & \rightarrow & \mathrm{Finzero}; \\
(\mathrm{Finonehat}) & \rightarrow & \mathrm{Finone}; \\
(\mathrm{Boolhat}) & \rightarrow & \mathrm{Bool}; \\
(\mathrm{Sigmahat}\ a\ b) & \rightarrow & \mathrm{Sigma}\ (\mathrm{T}\ a)\ (\lambda(x :: \\
(\mathrm{Pihat}\ a\ b) & \rightarrow & (x :: \mathrm{T}\ a) \rightarrow \mathrm{T}\ (b\ x)
\end{array}
\end{array}
$$

(i) Algebraic Data Types.

- The construct "data" in Agda is much more powerful th
by type theoretic rules.

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

$$
\begin{array}{lll}
\mathrm{A} & :: & \mathrm{Set} \\
 & = & \mathrm{data}\ \mathrm{C}_1(a_{11} :: \mathrm{A}_{11}) \cdots (\mathrm{a}_{1\mathrm{n}_1} :: \mathrm{A}_{1\mathrm{n}_1}) \\
 & & \quad \mid \mathrm{C}_2(a_{21} :: \mathrm{A}_{21}) \cdots (a_{2n_2} :: \mathrm{A}_{2\mathrm{n}_2} \\
 & & \quad \cdots \\
 & & \quad \mid \mathrm{C}_\mathrm{m}(a_{m1} :: \mathrm{A}_{\mathrm{m}1}) \cdots (a_{mn_m} :: \mathrm{A}
\end{array}
$$

(C) Anton Setzer 2003 (except for pictures)

Meaning of "data"

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

```
Ci@A ::  (ai1 :: Ai1)
         -> ...
         -> (aini :: Aini)
         -> A
```

where a constructor always constructs new elements.

- In other words the elements of A are exactly those co
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 **ne**
    of A.
  - Example:
    ```
    A ::  Set
      = data C (f ::  A -> A)
    ```
  - What is the least set A having a constructor
    ```
    C@A ::  (f ::  A -> A)
            -> A  ?
    ```

(C) Anton Setzer 2003 (except for pictures)

Strictly Positive Algebraic Data Types (Co

– If we
  * have constructed some part of A already,
  * find a function f:: A -> A, and
  * add C@_ f to A,

  then f might no longer be a function A -> A.
  (f applied to the new element C@_ f might not be defi
– In fact, "agda-check-termination" issues a warning, if we

– We shouldn't make use of such definitions.

Strictly Positive Algebraic Data Types (Co

- A "good" definition is the set of lists of natural numbers,

```
Nlist ::  Set
      = data nil
           | cons (a::  N)
                  (l ::  Nlist)
```

- The constructor cons@_ of N-lists refers to Nlist, but in
  We have: if a :: N and l :: Nlist, then we hav
  Nlist.
  - If we add cons@_ a l to Nlist, the reason for addi
    Nlist) is not destroyed by this addition.
  - So we can "construct" the set Nlist by
    - starting with the emptyset,
    - adding nil@_ and
    - closing it under cons@_ whenever possible.

- Because we can "construct" Nlist, the above is an accep

Strictly Positive Algebraic Data Types (Co

- In general:

```
A :: Set
  = data C_1 (a_11:: A_11) ··· (a_1n_1:: A_1n_1)
         | C_2 (a_21:: A_21) ··· (a_2n_2:: A_2n_2)
         ···
         | C_m (a_m1:: A_m1) ··· (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 i
- The definitions of finite sets, $\Sigma\ A\ B$, $A + B$ and N we algebraic data types.

One further Example

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

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

- This is a strictly positive data type.

Strictly Positive Algebraic Data Types Extensions

- An often used extension is to define several sets simultane
- Example: the even and odd numbers:

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

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

- In such examples the constructors refer strictly positive to to be defined simultaneously.

Strictly Positive Algebraic Data Types, Extensions

- We can even allow $\mathtt{A_{ij}} = \mathtt{B_1} -> \mathtt{A}$ or even $\mathtt{A_{ij}} = \mathtt{B_1} ->$
  where A is one of the types introduced simultaneously.

- Example (called "Kleene's O":

```
O::  Set
 = data leaf
       | succ (o::O)
       | lim (f ::  N -> O)
```

- The last definition is unproblematic, since, if we have f
  construct lim@_ f out of it, adding this new element to
  the reason for adding it to O.

- So again O can be "constructed".

(C) Anton Setzer 2003 (except for pictures)

Elimination Rules for data

- Functions from strictly positive data types can now b distinction as before.
- For termination we need only that in the definition of f, w `f (C@_ a`$_1$ `···` `a`$_n$`)`, we can refer only to f applied to ele `a`$_1$ `···` `a`$_n$.

- For instance
  - in the Bintree example, when defining

    $$\texttt{f} :: \texttt{Bintree}- > \texttt{A}$$

    by case-distinction, then the definition of

    $$\texttt{f}\ (\texttt{branch@_ left right})$$

    can make use of `f left` and `f right`.
  - In the example of `O`, when defining

    $$\texttt{g} :: \texttt{O}- > \texttt{A}$$

    by case-distinction, then the definition of

    $$\texttt{g}\ (\texttt{lim@_ f})$$

    can make use of `g (f n)` for all `n::  N`.