(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 $\Sigma$-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**

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

**Elimination Rule**

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



(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
  - $\mathrm{tt}$ stands for true, $\mathrm{ff}$ 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 less readable way:
  - $\mathrm{Case}_{\mathrm{Bool}} : (C : \mathrm{Bool} \to \mathrm{Set}) \to (ic : C\ \mathrm{tt}) \to (ec : C\ \mathrm{ff}) \to (cond : \mathrm{Bool}) \to C\ cond$ .
- tt, ff are the **constructors** of Bool.

(C) Anton Setzer 2003 (except for pictures)

## Remarks (Cont.)

- Notice that we then get for $C : \mathrm{Bool} \to \mathrm{Set}$, $ic : C\ \mathrm{tt}, ec : C\ \mathrm{ff}$
  - $f := \mathrm{Case}_{\mathrm{Bool}}\ C\ ic\ ec$ ,
    $: (cond : \mathrm{Bool}) \to C\ cond$
  - $f\ \mathrm{tt} = \mathrm{Case}_{\mathrm{Bool}}\ C\ ic\ ec\ \mathrm{tt} = ic : C\ \mathrm{tt}$,
  - $f\ \mathrm{ff} = \mathrm{Case}_{\mathrm{Bool}}\ C\ ic\ ec\ \mathrm{ff} = ec : C\ \mathrm{ff}$.
- So we obtain functions from Bool into other sets **without having to write** $\lambda(\mathbf{b} : \mathbf{Bool}).\cdots$.
- That's why we choose the argument to eliminate from as the **last one**.



(C) Anton Setzer 2003 (except for pictures)

## Remarks (Cont.)

- This is similar to the definition of for instance $(+)$ in **curried form** in Haskell
  - $(+) : \mathrm{int} \to \mathrm{int} \to \mathrm{int}$.
  - $(+)\ 3$ is the function which takes an integer and adds to it $3$.
    - **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** of $\mathrm{Case}_{\mathrm{Bool}}$:
  - First we have the **set into which we eliminate**.
  - Then follow the **cases**, one for each constructor.
  - Finally we put the **element which we are eliminating**.
- In some sense $\mathrm{Case}_{\mathrm{Bool}}$ is a "then _else _if " – the **condition** (if ...) **is the last one**.



(C) Anton Setzer 2003 (except for pictures)

## Example

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

- 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 $b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \lambda(b' : \mathbf{Bool}).\mathit{Bool} : \mathbf{Bool} \to \mathrm{Set}$:

$$
\begin{array}{c}
\mathit{Bool} : \mathrm{Set} \\ \hline
\mathbf{Bool} : \mathrm{Type} \\ \hline
b : \mathbf{Bool} \Rightarrow \mathrm{Context} \\ \hline
b : \mathbf{Bool} \Rightarrow \mathit{Bool} : \mathrm{Set} \\ \hline
b : \mathbf{Bool} \Rightarrow \mathbf{Bool} : \mathrm{Type} \\ \hline
b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{Context} \\ \hline
b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathit{Bool} : \mathrm{Set} \\ \hline
b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathbf{Bool} : \mathrm{Type} \\ \hline
b : \mathbf{Bool}, c : \mathbf{Bool}, b' : \mathbf{Bool} \Rightarrow \mathrm{Context} \\ \hline
b : \mathbf{Bool}, c : \mathbf{Bool}, b' : \mathbf{Bool} \Rightarrow \mathit{Bool} : \mathrm{Set} \\ \hline
b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \lambda(b' : \mathbf{Bool}).\mathit{Bool} : \mathbf{Bool} \to \mathrm{Set}
\end{array}
$$



(C) Anton Setzer 2003 (except for pictures)

## Example (Cont.)

- Similarly follows

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



(C) Anton Setzer 2003 (except for pictures)

## Example (Cont.)

- We derive

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

(using part of the derivation above):

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



(C) Anton Setzer 2003 (except for pictures)

## Example (Cont.)

- Using part of the proof above, we derive

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

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

- We derive

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

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

## Example (Cont.)

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

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





## Example (Cont.)

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

$$\frac{\dfrac{\dfrac{\begin{array}{c} b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \lambda(b' : \mathbf{Bool}).\mathit{Bool} : \mathbf{Bool} \to \mathrm{Set} \\ b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow c : (\lambda(b' : \mathbf{Bool}).\mathit{Bool})\ \mathrm{tt} \\ b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{ff} : (\lambda(b' : \mathbf{Bool}).\mathit{Bool})\ \mathrm{ff} \\ b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow b : \mathbf{Bool} \end{array}}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathrm{Case}_{\mathrm{Bool}}\ (\lambda(b' : \mathbf{Bool}).\mathit{Bool})\ c\ \mathrm{ff}\ b : \mathbf{Bool}}}{b : \mathbf{Bool} \Rightarrow \lambda(c : \mathbf{Bool}).\mathrm{Case}_{\mathrm{Bool}}\ (\lambda(b' : \mathbf{Bool}).\mathit{Bool})\ c\ \mathrm{ff}\ b : \mathbf{Bool} \to \mathbf{Boo}}}{\lambda(b, c : \mathbf{Bool}).\mathrm{Case}_{\mathrm{Bool}}\ (\lambda(b' : \mathbf{Bool}).\mathit{Bool})\ c\ \mathrm{ff}\ b : \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Boo}}$$





## Elimination into Type

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

**Elimination Rule into Type**

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

**Equality Rules into Type**

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

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





## Elimination into Type (Cont.)

We can extend this into an elimination rule **into Kind or other higher types**.

(C) Anton Setzer 2003 (except for pictures)

## Bool in Agda

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

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

- This introduces as well constants
  - tt :: Bool
  - ff :: Bool
- With this syntax, each constructor **can occur at most once in a data type**,
  - i.e. we cannot define a second type having constructor tt, e.g. for defining True (which is used later):

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



(C) Anton Setzer 2003 (except for pictures)

## Bool in Agda (Cont.)

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

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



(C) Anton Setzer 2003 (except for pictures)

## Bool in Agda (Cont.)

- The definition of Bool as

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

introduces Bool as a set **having constructors tt@Bool and ff@Bool**.

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



(C) Anton Setzer 2003 (except for pictures)

## Bool in Agda (Cont.)

- **Internally**, tt will always be represented as tt@Bool, similarly for ff.
  - So Agda **evaluates tt to tt@Bool**.
  - This can be seen when using for instance **"agda-compute-WHNF"**, compute weak head normal form.

(C) Anton Setzer 2003 (except for pictures)

## Case Distinction

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

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



(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-) menu item **"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 it starts with a constructor (or is a variable).
  - Then type into the mini-buffer $x$.
  - One gets the answer

    $\mathrm{tt}@_$.



(C) Anton Setzer 2003 (except for pictures)

## Case Distinction (Cont.)

- 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:

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



(C) Anton Setzer 2003 (except for pictures)

## Case Distinction (Cont.)

- Alternatively, **check**, the cursor being in that goal, **the context**
  - (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:

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

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



(C) Anton Setzer 2003 (except for pictures)

## Testing the Defined Function

- So we
  - type in a dummy goal:

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

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



(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 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** where Agda allows us to **compute the weak head normal form of expressions**.



(C) Anton Setzer 2003 (except for pictures)

## (b) The Finite Sets

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

**Formation Rule**

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

**Introduction Rules**

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

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

**Elimination Rule**

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

(C) Anton Setzer 2003 (except for pictures)

## The Finite Sets (Cont)

**Equality Rules**

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

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



(C) Anton Setzer 2003 (except for pictures)

## More compact elimination rules

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



(C) Anton Setzer 2003 (except for pictures)

## Omitting Premises in Equality Rules

Since the premises of the equality rule can in most cases be determined from the introduction and elimination rules, we will **usually omit them**, and write 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)

## Elimination into Type

- Similarly as for Bool we can write down **elimination rules**, where $\mathbf{C : Fin_n \rightarrow Type}$ (instead of $C : \mathrm{Fin}_n \rightarrow \mathrm{Set}$).
- 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 $\mathrm{Fin}_n$ 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 $C$ true we obtain an element of $C\ t$ **for every t : True**.
  - So there is no $C : \mathrm{True} \to \mathrm{Set}$ s.t. $C$ true is inhabited, but $C\ x$ is not inhabited for some other $x : \mathrm{True}$.
  - This means that all elements of $x$ of type True are **indistinguishable from 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 $\mathrm{Fin}_n$ for $n = 0$:

**Formation Rule**

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

**There is no Introduction Rule**

**Elimination Rule**

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

**There is no Equality Rule**



(C) Anton Setzer 2003 (except for pictures)

## 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 obtain an element of any set** (which might depend on $f$).
  - From a logic point of view this is "Ex falsum quodlibet" (from the absurdity follows anything).
    E.g. A **false formula** like "$0 = 1$" or "Swansea lies in Germany" **implies everything**.

(C) Anton Setzer 2003 (except for pictures)

## False (Cont.)

- $\mathrm{Case}_{\mathrm{False}}$ has **no computational meaning**, since there is no element it can be applied to.
  - Applies of course only if we are working in a **terminating type theory**.
  - If we had **full recursion**, we could define $f : \mathrm{False}$ 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 $\mathrm{False}$



(C) Anton Setzer 2003 (except for pictures)

## False in Agda

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

$$\mathrm{data\ False} =$$

- If we want to solve

$$\begin{array}{ll} g & (x :: \mathrm{False}) \\ :: & \mathrm{Bool} \\ = & \{!\ \ !\} \end{array}$$

  we can insert into the goal $x$ and choose menu-item **"agda-case"**.



(C) Anton Setzer 2003 (except for pictures)

## Finite Sets in Agda

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

$$\mathrm{data\ Colour} = \mathrm{blue} \mid \mathrm{red} \mid \mathrm{green}$$

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

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



(C) Anton Setzer 2003 (except for pictures)

## False in Agda (Cont.)

- The result is

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

- If we make case distinction on $x$ there is **no case to choose from**, so we 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.

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



(C) Anton Setzer 2003 (except for pictures)

## Example for the Use of False (Cont.)

- 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":

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



(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 expresses the fact that we know this.
- Note that we **don't have to invent a result** of $f$ in case $t$ is a pine tree.



(C) Anton Setzer 2003 (except for pictures)

## Example 2 for the Use of False

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

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

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 empty**.

## 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**:

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





# (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 provable, i.e.

**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 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**).





## Atomic Formulae (Cont.)

- There are formulae in type theory, which are **neither type-theoretically true nor type-theoretically false**.
  - This means that we can neither prove them, nor derive from a proof a contradiction.
  - Truth in type theory means that we **know that it is true**.
  - Falsity in type theory means that we **know that it cannot be true**.
  - There are formulae in type theory for which **neither of these two holds**.
- True and False as above are formulae corresponding to the **truth values true and false**.

(C) Anton Setzer 2003 (except for pictures)

## atom

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

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

- 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 needed)

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

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



(C) Anton Setzer 2003 (except for pictures)

## atom in Agda

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



(C) Anton Setzer 2003 (except for pictures)

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

## Decidable Predicates (Cont.)

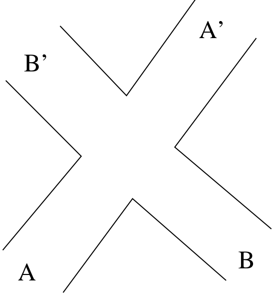
- Let now $g : A \rightarrow \text{Set}$, $g\ a = \text{atom}(f\ a)$.
  - If $f\ a$ is **true** (e.g. $a$ is safe), $g\ a$ is **inhabited**.
  - If $f\ a$ is **false** (e.g. $a$ is unsafe), $g\ a$ is **not inhabited**.
- Now, the existence of a $h : (a : A) \rightarrow g\ a$ means:
  - For all $a : A$ 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'.





## 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')

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





## 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.

(C) Anton Setzer 2003 (except for pictures)

## The Set of Control States (Cont.)

- We therefore define

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



(C) Anton Setzer 2003 (except for pictures)

## Mapping Control States to Physical States

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

```
toSigA  (s :: Control_State)
        ::  Colour
        =   case s of
              { (Allred)  →  red;
                (Agreen)  →  green;
                (Bgreen)  →  red; }

toSigB  (s :: Control_State)
        ::  Colour
        =   case s of
              { (Allred)  →  red;
                (Agreen)  →  red;
                (Bgreen)  →  green; }
```

(C) Anton Setzer 2003 (except for pictures)

## Mapping Control States to Physical States

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

```
phys_state  (s :: Control_State)
            ::  Phys_State
            =   struct{ sigA  =  toSigA s;
                        sigB  =  toSigB s; }
```



(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**, without defining first a Boolean function.
  - We first define a predicate depending on two signals:

```
CorAux  (a :: Colour)
        (b :: Colour)
        ::  Set
        =   case a of
              { (red)    →  True;
                (green)  →  case b of
                              { (red)    →  True;
                                (green)  →  False; }; }
```

(C) Anton Setzer 2003 (except for pictures)

## Safety Predicate (Cont.)

– Now we define

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

- **Remark:** In some cases in order to define a function from 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).



(C) Anton Setzer 2003 (except for pictures)

## Safety of the System (Cont.)

- The first element $\mathrm{true}$ was an element of $\mathbf{Cor(phys_state\ Allred)}$, which reduces to $\mathbf{True}$.
- 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 $\mathbf{False}$, which we can't solve.



(C) Anton Setzer 2003 (except for pictures)

## Safety of the System

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

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



(C) Anton Setzer 2003 (except for pictures)

## Safety of the System (Cont.)

- If one makes a **mistake** which results in an unsafe situation (e.g. sets $\mathrm{toSigB\ Agreen} = \mathrm{green}$, then in the last step we obtain one goal of type $\mathrm{False}$.
  – Then we can't solve this goal directly and **cannot prove the correctness**.
  – (In fact we could type-theoretically solve this goal by using **full recursion**, (e.g. solve this goal as $\mathbf{cor_proof\ Agreen}$), but this would be rejected 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**

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



(C) Anton Setzer 2003 (except for pictures)

## Disjoint Union using the Logical Framework

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



(C) Anton Setzer 2003 (except for pictures)

## Disjoint Union in Agda

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

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

(C) Anton Setzer 2003 (except for pictures)

## 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**" in a dummy goal.
  - Note that we cannot assign a type to $\mathrm{inr}@_$.
  - $(+)$ **cannot** be defined using the abbreviated $\mathrm{data}$ notation (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

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

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

$$\begin{array}{lllll} f & (c :: A+B) & & & \\ :: & \mathrm{Bool} & & & \\ = & \mathrm{case}\ c\ \mathrm{of} & & & \\ & \{ & (\mathrm{inl}\ a) & \to & \{!\ \ !\}; \\ & & (\mathrm{inr}\ b) & \to & \{!\ \ !\}; \} \end{array}$$

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



(C) Anton Setzer 2003 (except for pictures)

## Use of Concrete Disjoint Sets

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

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

- Now one can define for instance

$$\begin{array}{lllll} \mathrm{isFlower} & (p :: \mathrm{Plant}) & & & \\ & :: & \mathrm{Bool} & & \\ & = & \mathrm{case}\ p\ \mathrm{of} & & \\ & & \{\ (\mathrm{tree}\ t) & \to & \mathrm{ff}; \\ & & \ \ (\mathrm{flower}\ f) & \to & \mathrm{tt}; \} \end{array}$$

(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**

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



(C) Anton Setzer 2003 (except for pictures)

## The Σ-Set using the Logical Framework

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



(C) Anton Setzer 2003 (except for pictures)

## The Σ-Set using the Logical Framework (Cont.)

- $\mathrm{Sigma_Split} : (A : \mathrm{Set}) \to (B : A \to \mathrm{Set}) \to (C : (\Sigma\ A\ B) \to \mathrm{Set}) \to (s : (a : A, b : B\ a) \to C\ (\mathrm{p}\ A\ B\ a\ b)) \to (d : \Sigma\ A\ B) \to C\ d$ .

(C) Anton Setzer 2003 (except for pictures)

## The Σ-Set and the Dependent Product

- The **dependent product and the Σ-set are very similar**.
  - Both have similar introduction rules (for the Σ-set, the constructors have additional arguments $A$, $B$ necessary for bureaucratic reasons only).
  - One can define the projections $\pi_0$, $\pi_1$ using $\mathrm{Sigma_Split}$:

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

  - On the other hand, from $\pi_0$, $\pi_1$ we can define $\mathrm{Sigma_Split}$ as follows:

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



(C) Anton Setzer 2003 (except for pictures)

## 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, Σ works usually **better than the dependent product** in Agda.
  - I personally **don't use the dependent product** of Agda much.



(C) Anton Setzer 2003 (except for pictures)

## The Σ-Set in Agda

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

$$\begin{aligned}\mathrm{Sigma}\quad & (A :: \mathrm{Set})\\ & (B :: A \to \mathrm{Set})\\ & ::\ \ \mathrm{Set}\\ & =\ \ \mathrm{data}\ p\ (a :: A)\ (b :: B a)\end{aligned}$$



(C) Anton Setzer 2003 (except for pictures)

## 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", "flower"),
  - depending on $g :: \mathrm{Plant_Group}$, sets $\mathrm{Plants_in_group}\ g)$ for **plants in that group**.
- The **set of plants** can then be defined as

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

(C) Anton Setzer 2003 (except for pictures)

## The Σ-Set in Agda (Cont.)

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

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



(C) Anton Setzer 2003 (except for pictures)

## Formulae in Dependent Type Theory

- We have seen how to represent **atomic decidable formulae**.
- Now treatment of complex formulae constructed using **logical connectives**.



(C) Anton Setzer 2003 (except for pictures)

## Conjunction

- $\mathbf{A \land B}$ 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 a 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 elements of $A$ and $B$, i.e. $\mathbf{A \times B}$.
- We can **identify** $A \land B$ with $A \times B$.



(C) Anton Setzer 2003 (except for pictures)

## 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 $B$:

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

- This means that we can **derive A ∧ B from A and B**.
- This is what is expressed by the **ordinary introduction rule for** $\land$:

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

(C) Anton Setzer 2003 (except for pictures)

## Conjunction (Cont.)

- The **elimination rule** for $\wedge$ allows to project a proof of $A \wedge B$ to a proof of $A$ 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 and B**.
- This is what is expressed by the **ordinary elimination rule for $\wedge$**:

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



(C) Anton Setzer 2003 (except for pictures)

## Disjunction

- $A \vee B$ is true iff $A$ is true or $B$ is true.
- Therefore a **proof of A $\vee$ B consists of a proof of A or a proof of B, plus the information which one.**
  - It is therefore an element $\mathrm{inl}\ p$ for a proof $p : A$ or an element $\mathrm{inr}\ q$ for a proof $q : B$.
- Therefore the set of proofs of $A \vee B$ is the **disjoint union of A and B**, i.e. $\mathbf{A + B}$.
- We can **identify** $A \vee B$ with $A + B$.



(C) Anton Setzer 2003 (except for pictures)

## Disjunction (Cont.)

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

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

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

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



(C) Anton Setzer 2003 (except for pictures)

## Disjunction (Cont.)

- This means that we can **derive A $\vee$ B from A and from B**.
- This is what is expressed by the **ordinary introduction rules for $\vee$**:

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

(C) Anton Setzer 2003 (except for pictures)

## Disjunction (Cont.)

- The **elimination rule** for $+$ allows to form from an element of $A+B$ an element of any set $C$ provided we can compute such an element from $A$ and from $B$:

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

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

$$\frac{d : A \vee B \qquad sl : A \to C \qquad sr : B \to C}{\mathrm{Plus_Split}\ sl\ sr\ d : C}$$



(C) Anton Setzer 2003 (except for pictures)

## Disjunction (Cont.)

- This means that we can **derive from A $\vee$ B a formula C, if we can derive C from A and from B**.
- This is what is expressed by the **ordinary elimination rules for** $\vee$:

$$\frac{A \vee B \qquad \begin{array}{c}A\\ \vdots\\ C\end{array} \qquad \begin{array}{c}B\\ \vdots\\ C\end{array}}{C}$$

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



(C) Anton Setzer 2003 (except for pictures)

## Implication

- We write temporarily $\supset$ for logical implication, in order to distinguish it from the function type $\to$.
  - Below we see that $\supset$ can be identified with $\to$.
- $A \supset B$ 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 $\to$ B**.
- We can **identify** $A \supset B$ with $A \to B$.



(C) Anton Setzer 2003 (except for pictures)

## Implication (Cont.)

- 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**.

(C) Anton Setzer 2003 (except for pictures)

## Implication (Cont.)

- This is what is expressed by the **ordinary introduction rule for $\supset$**:

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



(C) Anton Setzer 2003 (except for pictures)

## Implication (Cont.)

- The **elimination rule for $\supset$** allows to apply a proof $p$ of $A \supset B$ to a proof 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 B holds**.
- This is what is expressed by the **ordinary elimination rule for $\supset$**:

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



(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 $\mathrm{False}$):
  - 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 $\neg\mathbf{A}$ 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 quantifiers explicitly the type, the bound variable is ranging over:
  We write therefore $\forall\mathbf{x} : \mathbf{A}.\mathbf{B}$, $\exists\mathbf{x} : \mathbf{A}.\mathbf{B}$.
- $\forall x : A.B$ is true iff, for all $x : A$ there exists a proof of $B$ (with that $x$).
- Therefore a proof of $\forall x : A.B$ is a **function, which takes an x:A and computes an element of B**.
- Therefore the set of proofs of $\forall x : A.B$ is the **dependent function type** $(\mathbf{x} : \mathbf{A}) \to \mathbf{B}$.
- We can **identify** $\forall\mathbf{x} : \mathbf{A}.\mathbf{B}$ with $(\mathbf{x} : \mathbf{A}) \to \mathbf{B}$.

(C) Anton Setzer 2003 (except for pictures)

## Universal Quantification (Cont.)

- 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 $x : A$:

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

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



(C) Anton Setzer 2003 (except for pictures)

## 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 x**.
    - This corresponds in type theory to the fact that **x:A does no longer occur in the context of the conclusion**.



(C) Anton Setzer 2003 (except for pictures)

## 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 of $B[x := a]$:

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

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



(C) Anton Setzer 2003 (except for pictures)

## Universal Quantification (Cont.)

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

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

(C) Anton Setzer 2003 (except for pictures)

## Existential Quantification

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



(C) Anton Setzer 2003 (except for pictures)

## Existential Quantification (Cont.)

- The **elimination rule** for the dependent product allows to project a proof $p$ of $\exists x : A.B$ to an element $\pi_0(p) : A$ and proof $\pi_1(p) : B[x := \pi_0(p)]$.
- This kind of rule works only if we have **explicit proofs**.
- From this we can derive a rule which is essentially that used in natural deduction (in which one doesn't have explicit proofs):
  - Assume:
    - $C : \mathrm{Set}$, which does not depend on $x : A$,
    - $p : \exists x : A.B$ and
    - $x : A, y : B \Rightarrow c : C$.
  - Then we have $c[x := \pi_0(p), y := \pi_1(p)] : C$, **not depending on x:A or y:B**.



(C) Anton Setzer 2003 (except for pictures)

## Existential Quantification (Cont.)

- 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 $p : B[x := a]$:

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

- This is what is expressed by the **ordinary introduction rule for $\exists$**:

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



(C) Anton Setzer 2003 (except for pictures)

## Existential Quantification (Cont.)

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

$$\frac{\exists x : A.B \qquad \begin{array}{c} x : A \\ B \\ \vdots \\ C \end{array}}{C}$$

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

(C) Anton Setzer 2003 (except for pictures)

## Constructive (or Intuitionistic) Logic

- From type theoretic proofs we can **directly extract programs**.
- For instance, if $p : \forall x : A.\exists y : B.C(x, y)$, then we have
  - for $x : A$ it follows $b := \pi_0(p\ x) : B$ and $\pi_1(p\ x) : C(x, b)$.
  - Therefore $f := \lambda x : A.\pi_0(p\ x)$ is a **function** $\mathbf{A} \to \mathbf{B}$, and we have

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

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



(C) Anton Setzer 2003 (except for pictures)

## Constructive Logic (Cont.)

- We can derive as well a function which **depending on** $\mathbf{p} : \mathbf{A} + \mathbf{B}$ **decides whether p= inl(a) or p = 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**

    $$\forall \mathbf{x} : \mathbf{Turing_Machine}.(\mathbf{x\ halts} \lor \neg(\mathbf{x\ halts}))$$



(C) Anton Setzer 2003 (except for pictures)

## Constructive Logic (Cont.)

- In classical logic we **can prove the above**, since we can derive $\mathbf{A} \lor \neg\mathbf{A}$ (tertium non datur) for any formula $A$.
- In type theory, this law **cannot hold**, unless we don't want that all programs can be evaluated.
  - The logic of type theory is **intuitionistic (constructive) logic**, in which $A \lor \neg A$ and $\neg\neg A \to A$ don't hold for all formulae $A$.



(C) Anton Setzer 2003 (except for pictures)

## Constructive Logic (Cont.)

- In **classical logic**,
  - $\exists \mathbf{x} : \mathbf{A}.\mathbf{B}$ **is equivalent to** $\neg\forall \mathbf{x} : \mathbf{A}.\neg\mathbf{B}$,
  - $\mathbf{A} \lor \mathbf{B}$ **is equivalent to** $\neg(\neg\mathbf{A} \land \neg\mathbf{B})$.
- If we take decidable atomic formulae only and replace $\exists x : A.B$ and $A \lor B$ by the above formulae, then **all formulas provable in classical logic are derivable**.
  - This requires $(\neg\neg A) \to A$, which can be shown for all formulae built from decidable atomic formulae using $\neg$, $\to$, $\land$, $\forall$.
  - The formula $A \lor \neg A$ translates into $\neg(\neg A \land \neg\neg A)$, which trivially holds, since $\neg A$ and $\neg\neg A$ implies $\bot$.
- In this sense, **type theory contains classical logic**, but is **richer**, since it has as well so called **strong disjunction and existential quantification**.

(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 $\neg\neg A \rightarrow A$ holds.



(C) Anton Setzer 2003 (except for pictures)

## Constructive Logic (Cont.)

- We show intuitionistically $\neg(\exists \mathbf{x} : \mathbf{A}.\mathbf{B}) \leftrightarrow \forall \mathbf{x} : \mathbf{A}.\neg \mathbf{B} \quad :$
  - Assume $\neg(\exists x : A.B)$, $x : A$ and show $\neg B$.
    If we had $B$, then we had $\exists x : A.B$, contradicting $\neg(\exists x : A.B)$. Therefore $\neg B$.
  - Assume $\forall x : A.\neg B$. Show $\neg(\exists x : A.B)$:
    Assume $\exists x : A.B$. Assume $x$ s.t. $B$ holds.
    By $\forall x : A.\neg B$ we get $\neg B$, therefore a contradiction.
- Now it follows (classically):

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



(C) Anton Setzer 2003 (except for pictures)

## Constructive Logic (Cont.)

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

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



(C) Anton Setzer 2003 (except for pictures)

## Constructive Logic (Cont.)

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

(C) Anton Setzer 2003 (except for pictures)

## (f) The Set of Natural Numbers

- The set $\mathrm{N}$ 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.



(C) Anton Setzer 2003 (except for pictures)

## The Set of Natural Numbers (Cont.)

- Let $\mathrm{S}$ be a type theoretic notation for the operation $x \mapsto x + 1$.
- 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 $\mathbf{f} : \mathbf{N} \to \mathbf{N}$**, 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 $\mathbf{f\ n}$** 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 compute $f\ n'$.
    - Now $f\ n$ reduces to $g\ n'\ (f\ n')$.
    - $g\ n'\ (f\ n')$ can be computed, since we know how to compute
      - $g$
      - $f\ n'$.

(C) Anton Setzer 2003 (except for pictures)

## Example

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



(C) Anton Setzer 2003 (except for pictures)

## Generalized Primitive Recursion

- 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 $C$

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

  - 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**

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

**Equality Rules**

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

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



(C) Anton Setzer 2003 (except for pictures)

## Rules for the Natural Numbers (Cont.)

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

$$g\ n : C\ n$$

which means that

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

  - The equality rules read:

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

(C) Anton Setzer 2003 (except for pictures)

## Rules for N using the Logical Framework

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



(C) Anton Setzer 2003 (except for pictures)

## Elimination Rules for N in Agda

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

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

    * $A$ possibly depending on $n$,

    we can type into the goal $n$ and use the menu agda-case.
    We get

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



(C) Anton Setzer 2003 (except for pictures)

## Natural Numbers in Agda

- N is defined using **data**:

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

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

- Therefore we have

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



(C) Anton Setzer 2003 (except for pictures)

## 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-message**.
- If we
  - **do not make use of f in the case n=Z** and
  - **only use of f n′ in case n = S n′**.

  then **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 should be **correct.**
  - (The lecturer hasn't checked the algorithm).
- However, **if agda-check-termination fails**, the **definition might still be correct**.



(C) Anton Setzer 2003 (except for pictures)

## Example of the Power of Termination Check

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

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



(C) Anton Setzer 2003 (except for pictures)

## Example for Limitations of Termination Check

- Assume we define the **predecessor function**

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

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 (Cont.)

- Then the function

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

  terminates always
  - (it returns for all $n : \mathrm{N}$ the value Z).
- However, **agda-check-termination fails**.

(C) Anton Setzer 2003 (except for pictures)

## Limitations of the Termination Check (Cont.)

- Because of the **undecidability of the Turing halting problem**
  - it is undecidable whether a recursively defined function terminates or not
- there is no **extension of agda-check-termination, which accepts exactly all in agda definable functions, which terminate for all inputs**.



(C) Anton Setzer 2003 (except for pictures)

## Example: Addition

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



(C) Anton Setzer 2003 (except for pictures)

## Example: Addition

- Definition of $+$ in Agda:

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

- The definition expresses:

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



(C) Anton Setzer 2003 (except for pictures)

## Example: Multiplication

- Definition

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

- The definition expresses:

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

(C) Anton Setzer 2003 (except for pictures)

## 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 defined**.



(C) Anton Setzer 2003 (except for pictures)

## Equality on N

- The **equality (n == m) ::Set** for $n, m :: \mathrm{N}$ can be defined using the 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:

```
(==)  (n, m :: N)
      ::  Set
      =   case n of
            { (Z)     →  case m of
                           { (Z)    →  {! !};
                             (S m') →  {! !};};
              (S n')  →  case m of
                           { (Z)    →  {! !};
                             (S m') →  {! !};};}
```

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



(C) Anton Setzer 2003 (except for pictures)

## Reflexivity of ==

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

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

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

```
refl  (n : N)
      ::  n == n
      =   {! !}
```

(C) Anton Setzer 2003 (except for pictures)

## Reflexivity of == (Cont.)

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

```
refl  (n : N)
      ::  n == n
      =   case n of
            { (Z)     →  {! !};
              (S n')  →  {! !}; }
```



(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 function sym:

```
sym  (n, m : N)
     (p :: n == m)
     ::  m == n
     =   {! !}
```



(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 defined before $\mathrm{refl}\ n'$).
- Task of Coursework 3, Question 1 (e) to solve this goal.



(C) Anton Setzer 2003 (except for pictures)

## Symmetry of == (Cont.)

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

```
sym  (n, m : N)
     (p :: n == m)
     ::  m == n
     =   case n of
           { (Z)     →  case m of
                          { (Z)     →  {! !};
                            (S m')  →  {! !}; };
             (S n')  →  case m of
                          { (Z)     →  {! !};
                            (S m')  →  {! !}; }; }
```

(C) Anton Setzer 2003 (except for pictures)

## Symmetry of == (Cont.)

- The **first goal** can be solved by using true (since $(\mathrm{Z} == \mathrm{Z}) = \mathrm{True}$).
- For the **second goal** we know $p$ is an element of $\mathrm{Z} == \mathrm{S}\ m'$ which is False.
  - 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**.



(C) Anton Setzer 2003 (except for pictures)

## Symmetry of == (Cont.)

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



(C) Anton Setzer 2003 (except for pictures)

## Example: Tuples (or Vectors) of Length n

- Define first

$$\mathrm{data\ Nil} \;=\; \mathrm{nil}$$

$$\begin{array}{ll} \mathrm{Cons} & (A, B :: \mathrm{Set}) \\ & :: \ \mathrm{Set} \\ & = \ \mathrm{data\ cons}(a :: A)(b :: B) \end{array}$$



(C) Anton Setzer 2003 (except for pictures)

## Example: Tuples (or Vectors) of Length n

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

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

(C) Anton Setzer 2003 (except for pictures)

## Tuples of Length n

- Therefore (with the obvious definition of two),

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

- 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 $\langle a_1, \ldots, a_n \rangle$ for such an element.



(C) Anton Setzer 2003 (except for pictures)

## Remarks on Tuples of Length n

- If we define

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

  then this reads:

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



(C) Anton Setzer 2003 (except for pictures)

## Remarks on Tuples of Length n

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

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



(C) Anton Setzer 2003 (except for pictures)

## Remarks on Tuples of Length n (Cont.)

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

(C) Anton Setzer 2003 (except for pictures)

## Example: Sum of Tuples of Length n

- Define

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

- Nvec $n$ are tuples of natural numbers of length $n$.



(C) Anton Setzer 2003 (except for pictures)

## 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:

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



(C) Anton Setzer 2003 (except for pictures)

## Example: Componentwise Sum of Tuples of Length n (Cont.)

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



(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:

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

(C) Anton Setzer 2003 (except for pictures)

## 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

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

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



(C) Anton Setzer 2003 (except for pictures)

## Example: Sum of the Elements of a List

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



(C) Anton Setzer 2003 (except for pictures)

## Example: Length of a List

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



(C) Anton Setzer 2003 (except for pictures)

## Interesting Exercise

- Define
$$\mathrm{append} : (A : \mathrm{Set}) \rightarrow (\mathrm{list}\ A) \rightarrow (\mathrm{list}\ A) \rightarrow \mathrm{list}\ A\ ,$$
  s.t. $\mathrm{append}\ A\ l\ l'$ is the result of appending the list $l'$ at the end of list $l$.
- E.g., if $\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}$ are elements of $A$,
  and if we define $\mathrm{cons} := \mathrm{cons}@(\mathrm{list}\ A)$, $\mathrm{nil} := \mathrm{nil}@(\mathrm{list}\ A)$, then:

$$\begin{array}{l} \mathrm{append}\ A\ (\mathrm{cons\ a\ (cons\ b\ nil)})\ (\mathrm{cons\ c\ (cons\ d\ nil)}) \\ = \mathrm{cons\ a\ (cons\ b\ (cons\ c\ (cons\ d\ nil)))} \end{array}$$

(C) Anton Setzer 2003 (except for pictures)

## (h) Universes.

- A universe $\mathrm{U}$ is a set, the elements of which are codes for sets.
- So we have
  - $\mathrm{U} : \mathrm{Set}$,
  - $\mathrm{T} : \mathrm{U} \to \mathrm{Set}$ (the decoding function).
- We consider in the following a universe closed under
  - $\mathrm{Fin}_0$, $\mathrm{Fin}_1$, $\mathrm{Bool}$,
  - $\mathrm{N}$,
  - $+$,
  - $\Sigma$,
  - 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.)

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

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

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

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



(C) Anton Setzer 2003 (except for pictures)

## Introduction/Equality Rules for the Universe (Cont.)

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

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

(C) Anton Setzer 2003 (except for pictures)

## Elimination and Equality Rules for the Universe (Cont.)

- There exist as well elimination rules and corresponding equality rules for the universe.
- They are very long (one step for each of constructor of $\mathrm{U}$) and are not very 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 $\mathrm{Set}$.
- 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 $\mathrm{Set}$ and use $\mathrm{T}$ to obtain the required function.



(C) Anton Setzer 2003 (except for pictures)

## Applications of the Universe

- Example: Define

$$
\begin{array}{lcl}
\widehat{\mathrm{atom}} & : & \mathrm{Bool} \to \mathrm{U} \ , \\
\widehat{\mathrm{atom}} & := & \mathrm{Case}_{\mathrm{Bool}}\ (\lambda(x : \mathrm{Bool}).\mathrm{U})\ \widehat{\mathrm{Fin}_1}\ \widehat{\mathrm{Fin}_0} \ , \\
 & & \\
\mathrm{atom} & : & \mathrm{Bool} \to \mathrm{Set} \ , \\
\mathrm{atom} & : & \lambda(x : \mathrm{Bool}).\mathrm{T}\ (\widehat{\mathrm{atom}}\ x) \ ,
\end{array}
$$

Then

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



(C) Anton Setzer 2003 (except for pictures)

## Universes in Agda

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

(C) Anton Setzer 2003 (except for pictures)

## Universes in Agda (Cont.)

$$\begin{array}{l}\text{mutual}\\ \quad \mathrm{U} \;::\; \text{Set}\\ \quad\quad = \text{data Nhat}\\ \quad\quad\quad\quad \mid \text{Finzerohat}\\ \quad\quad\quad\quad \mid \text{Finonehat}\\ \quad\quad\quad\quad \mid \text{Boolhat}\\ \quad\quad\quad\quad \mid \text{Sigmahat } (a :: \mathrm{U})(b :: \mathrm{T}\ a \to \mathrm{U})\\ \quad\quad\quad\quad \mid \text{Pihat } (a :: \mathrm{U})(b :: \mathrm{T}\ a \to \mathrm{U})\end{array}$$



(C) Anton Setzer 2003 (except for pictures)

## Universes in Agda (Cont.)

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

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



(C) Anton Setzer 2003 (except for pictures)

## (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.

$$\begin{array}{lll}\mathrm{A} & :: & \text{Set}\\ & = & \text{data } \mathrm{C}_1(a_{11} :: \mathrm{A}_{11})\cdots(a_{1n_1} :: \mathrm{A}_{1n_1})\\ & & \quad\mid \mathrm{C}_2(a_{21} :: \mathrm{A}_{21})\cdots(a_{2n_2} :: \mathrm{A}_{2n_2})\\ & & \quad\cdots\\ & & \quad\mid \mathrm{C}_m(a_{m1} :: \mathrm{A}_{m1})\cdots(a_{mn_m} :: \mathrm{A}_{mn_m})\end{array}$$



(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 constructors:

$$\begin{array}{ll}\mathtt{C_i@A} :: & (\mathtt{a}_{i1} :: \mathtt{A}_{i1})\\ & \texttt{->}\ \cdots\\ & \texttt{->}\ (\mathtt{a}_{in_i} :: \mathtt{A}_{in_i})\\ & \texttt{->}\ \mathtt{A}\end{array}$$

  where a constructor always constructs new elements.

- In other words the elements of A are exactly those constructed by those constructors.

(C) Anton Setzer 2003 (except for pictures)

## Strictly Positive Algebraic Data Types

- In the types `Aij` we can make use of `A`.
  - However, it is difficult to understand `A`, if we have **negative** occurrences 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 (Cont.)

- 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 defined).
- In fact, "agda-check-termination" issues a warning, if we define `A` as above.
- We shouldn't make use of such definitions.



(C) Anton Setzer 2003 (except for pictures)

## Strictly Positive Algebraic Data Types (Cont.)

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

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

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



(C) Anton Setzer 2003 (except for pictures)

## Strictly Positive Algebraic Data Types (Cont.)

- In general:

$$
\begin{array}{l}
\mathtt{A} :: \mathtt{Set} \\
\quad = \mathtt{data}\ C_1\ (a_{11}::\ A_{11}) \cdots (a_{1n_1}::\ A_{1n_1}) \\
\qquad |\ C_2\ (a_{21}::\ A_{21}) \cdots (a_{2n_2}::\ A_{2n_2}) \\
\qquad \cdots \\
\qquad |\ C_m\ (a_{m1}::\ A_{m1}) \cdots (a_{mn_m}::\ A_{mn_m})
\end{array}
$$

  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 $\mathbb{N}$ were strictly positive algebraic data types.

(C) Anton Setzer 2003 (except for pictures)

## 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.



(C) Anton Setzer 2003 (except for pictures)

## Strictly Positive Algebraic Data Types, Extensions

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

- Example: the even and odd numbers:

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

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

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



(C) Anton Setzer 2003 (except for pictures)

## Strictly Positive Algebraic Data Types, Extensions

- We can even allow $\texttt{A}_{\texttt{ij}} = \texttt{B}_1 -> \texttt{A}$ or even $\texttt{A}_{\texttt{ij}} = \texttt{B}_1 -> \cdots -> \texttt{B}_1 -> \texttt{A}$, 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 ::  N -> O and construct lim@_ f out of it, adding this new element to O doesn't destroy the reason for adding it to O.

- So again O can be "constructed".



(C) Anton Setzer 2003 (except for pictures)

## Elimination Rules for data

- Functions from strictly positive data types can now be defined by case distinction as before.

- For termination we need only that in the definition of $f$, when have to define f (C@_ $\texttt{a}_1 \cdots \texttt{a}_n$), we can refer only to $f$ applied to elements used in C@_ $\texttt{a}_1 \cdots \texttt{a}_n$.

## Examples

- For instance
  - in the Bintree example, when defining

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

    by case-distinction, then the definition of

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

    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

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

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