

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.

(a) The Set of Booleans

Formation Rule

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

Introduction Rules

$\text{tt} : \text{Bool}$

$\text{ff} : \text{Bool}$

Elimination Rule

$$\frac{C : \text{Bool} \rightarrow \text{Set} \quad \text{ic} : C \quad \text{tc} : C \quad \text{ec} : C \quad \text{fc} : C \quad \text{cc} : C \quad \text{cond} : C \rightarrow \text{Bool}}{\text{Case}_{\text{Bool}} C \text{ ic ec cond} : C \rightarrow \text{Bool}}$$

Equality Rules

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

The Set of Booleans (Cont.)

Remarks

- In the above

- \tilde{t} stands for true, \tilde{f} stands for false.
- \tilde{ic} stands for "if-case", \tilde{ec} for "else-case".
- \tilde{con} for "condition".

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

$$- \text{Case}_{\text{Bool}} : (C : \text{Bool} \rightarrow \text{Set})$$

$$\rightarrow (ic : C \rightarrow \tilde{t})$$

$$\rightarrow (ec : C \rightarrow \tilde{f})$$

$$\rightarrow (cond : \text{Bool})$$

$$\rightarrow C \rightarrow \text{cond}$$

- \tilde{t} , \tilde{f} are the **constructors** of `Bool`.

Remarks (Cont.)

- Notice that we then get for $C : \text{Bool} \rightarrow \text{Set}$, $ic : C \text{ tt}, ec : C \text{ ff}$

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

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

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

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

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

Remarks (Cont.)

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

Remarks (Cont.)

- Note that we have the following **order of the arguments** of $\text{Case}_{\text{Bool}}$:
 - 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 $\text{Case}_{\text{Bool}}$ is a “then_else_if” – the **condition** (if ...) **is the last one**.

Example

$\text{AND} := \lambda(b, c : \text{Bool}). \text{Case}_{\text{Bool}} (\lambda(b' : \text{Bool}). \text{Bool}) c \text{ ff } b$
 $: \text{ Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

• AND is the **conjunction**:

– AND tt $c = c$.

– Correct since tt $\wedge c = c$.

– AND ff $c = \text{ff}$.

– Correct since ff $\wedge c = \text{ff}$.

• In the following we write Bool, if it

– is a type in **boldface red**,

– and if it is a term, in *italic blue*.

Example (Cont.)

- Derivation of $\text{AND} : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$:

– First we derive $b : \text{Bool}, c : \text{Bool} \Rightarrow \lambda(b' : \text{Bool}). \text{Bool} : \text{Bool} \rightarrow \text{Set}$:

$$\begin{array}{c}
 \text{Bool} : \text{Type} \\
 \hline
 b : \text{Bool} \Rightarrow \text{Context} \\
 \hline
 b : \text{Bool} \Rightarrow \text{Bool} : \text{Type} \\
 \hline
 b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context} \\
 \hline
 b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Bool} : \text{Set} \\
 \hline
 b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Bool} : \text{Type} \\
 \hline
 b : \text{Bool}, c : \text{Bool}, b' : \text{Bool} \Rightarrow \text{Context} \\
 \hline
 b : \text{Bool}, c : \text{Bool}, b' : \text{Bool} \Rightarrow \text{Bool} : \text{Set} \\
 \hline
 b : \text{Bool}, c : \text{Bool} \Rightarrow \lambda(b' : \text{Bool}). \text{Bool} : \text{Bool} \rightarrow \text{Set}
 \end{array}$$

Example (Cont.)

- Similarly follows

$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathbf{Bool} = (\lambda (b' : \mathbf{Bool}). \mathbf{Bool}) . \mathbf{Bool} \quad \# : \text{Type}$$

Example (Cont.)

– Using part of the proof above, we derive

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

$$\frac{\dots \quad b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \text{Context} \quad b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow c : \mathbf{Bool}}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow c : (\lambda(b' : \mathbf{Bool}). \mathbf{Bool}) \text{ tt}}$$

– We derive

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

$$\frac{\dots \quad b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \text{Context} \quad b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathbf{ff} : \mathbf{Bool}}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathbf{ff} : (\lambda(b' : \mathbf{Bool}). \mathbf{Bool}) \text{ ff}}$$

Example (Cont.)

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

$$\frac{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathbf{Context} \quad \dots}{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow b : \mathbf{Bool}}$$

Example (Cont.)

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

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

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

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

$$b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow b : \mathbf{Bool}$$

$$\frac{b : \mathbf{Bool}, c : \mathbf{Bool} \Rightarrow \mathbf{Bool} \Rightarrow \text{Case}_{\mathbf{Bool}}(\lambda(b' : \mathbf{Bool}). \mathbf{Bool}) \text{ c ff } b : \mathbf{Bool}}{b : \mathbf{Bool} \Rightarrow \lambda(c : \mathbf{Bool}). \text{Case}_{\mathbf{Bool}}(\lambda(b' : \mathbf{Bool}). \mathbf{Bool}) \text{ c ff } b : \mathbf{Bool}}$$

$$\frac{b : \mathbf{Bool} \Rightarrow \lambda(c : \mathbf{Bool}). \text{Case}_{\mathbf{Bool}}(\lambda(b' : \mathbf{Bool}). \mathbf{Bool}) \text{ c ff } b : \mathbf{Bool} \rightarrow \mathbf{Bool}}{\lambda(b, c : \mathbf{Bool}). \text{Case}_{\mathbf{Bool}}(\lambda(b' : \mathbf{Bool}). \mathbf{Bool}) \text{ c ff } b : \mathbf{Bool} \rightarrow \mathbf{Bool}}$$

Elimination into Type

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

Elimination Rule into Type

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

Equality Rules into Type

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

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

Elimination into Type (Cont.)

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

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

– $\text{tt} :: \text{Bool}$
– $\text{ff} :: \text{Bool}$

- 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):

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

Bool in Agda (Cont.)

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

```
Bool :: Set
      = data tt | ff
      tt  : Bool
      ff  : Bool
      tt@Bool =
      ff@Bool =
```

Bool in Agda (Cont.)

- The definition of Bool as

```

Bool :: Set
      = data tt | ff

```

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.

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.

Case Distinction

- Elimination in Agda is based on case distinction.

- Assume we want to define
 - * $f : \text{Bool} \rightarrow \text{Bool}$, s.t.
 - * $f \text{ tt} = \text{ff}$,
 - * $f \text{ ff} = \text{tt}$.
- So we have the goal:

$$f \quad (x :: \text{Bool}) :: \text{Bool} = \{ | i \}$$

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:

$$\begin{aligned}
 f \quad (x :: \text{Bool}) &:: \text{Bool} \\
 = \text{case } x \text{ of} & \\
 \{ \text{tt} \} &\leftarrow \{ i \}; \\
 (\text{ff}) &\leftarrow \{ i \};
 \end{aligned}$$

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 `tt@-.`

Case Distinction (Cont.)

- Alternatively, **check**, the cursor being in that goal, **the context** – (use goal-menu “**agda-context**”):
It contains
$$x :: \text{Bool} = \text{tt} @ _.$$
- Similarly one finds that in the second goal x is $\text{ff} @ _.$

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:

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

{	(tt)	←	ff;
	(ff)	←	tt; }

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

Testing the Defined Function

- We can test our function by using “**agda-compute-WHNF**”.

- We have to create a goal for this.

- The reduction machinery is **context dependent**.
- The context depends on where in the buffer we are.
- * See the above example where x was depending on the goal tt or ff .
- Not every place in the buffer is a good place.
- **Good places for context are goals**, and that's **the only place** where Agda allows us to **compute the weak head normal form of expressions**.

Testing the Defined Function

- So we

– type in a dummy goal:

```
test :: Set  
      = { i }
```

– move to the new goal

– choose “**agda-compute-WHNF**”,
– and type into the mini-buffer *f* tt.

- The result shown is $f@-$.

(b) The Finite Sets

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

Formation Rule

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

Introduction Rules

$$A_n^k : \text{Fin}_n$$

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

Elimination Rule

$$\begin{array}{c}
 C : \text{Fin}_n \rightarrow \text{Set} \\
 s_0 : C A_n^0 \\
 s_1 : C A_n^1 \\
 \dots \\
 s_{n-1} : C A_n^{n-1} \\
 \hline
 \text{Case}_n C s_0 \dots s_{n-1} a : C a
 \end{array}$$

Equality Rules

The Finite Sets (Cont)

$$\frac{C : \text{Fin}_n \rightarrow \text{Set} \quad s_0 : C A_n^0 \quad s_1 : C A_n^1 \quad \dots \quad s_{n-1} : C A_n^{n-1}}{\text{Case}_n C s_0 \dots s_{n-1} A_n^k = s_k : C A_n^k}$$

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

Omitting Premises in Equality Rules

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

$$\text{Case}_n \ C \ s_0 \ \dots \ s_{n-1} \ A_n^k = s_k : C \ A_n^k$$

We sometimes even **omit the type**:

$$\text{Case}_n \ C \ s_0 \ \dots \ s_{n-1} \ A_n^k = s_k$$

More compact elimination rules

- $\text{Case}_n : (C : \text{Fin } n \rightarrow \text{Set})$
 $\rightarrow (s_0 : C \text{A}_n^0)$
 $\dots \rightarrow$
 $\rightarrow (s_{n-1} : C \text{A}_{n-1}^{n-1})$
 $\rightarrow (a : \text{Fin } n)$
 $\rightarrow C \ a$

Elimination into Type

- Similarly as for Bool we can write down **elimination rules**, where $C : \text{Fin}_n \rightarrow \text{Type}$ (instead of $C : \text{Fin}_n \rightarrow \text{Set}$).
- This can be done for all sets defined later as well.

Rules for True

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

Formation Rule

True : Set

Introduction Rules

true : True

Elimination Rule

$$\frac{C : \text{True} \rightarrow \text{Set} \quad e : C \quad \text{true} : \text{True}}{\text{CaseTrue } e \ t : C \ t}$$

Equality Rule

$$\frac{C : \text{True} \rightarrow \text{Set} \quad e : C \quad \text{true} : \text{True}}{\text{CaseTrue } e \ \text{true} = e : C \ \text{true}}$$

Rules for True (Cont.)

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

$$\lambda(t : \text{True}).c : (t : \text{True}) \rightarrow C\ t$$

- From a **logic point of view**, it expresses:
 - From an element of C true we obtain an element of $C\ t$ **for every $t : \text{True}$** .
 - So there is no $C : \text{True} \rightarrow \text{Set}$ s.t. C true is inhabited, but $C\ x$ is not inhabited for some other $x : \text{True}$.
 - 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**.

Rules for False

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

Formation Rule

$\text{False} : \text{Set}$

There is no Introduction Rule

Elimination Rule

$$\frac{C : \text{False} \rightarrow \text{Set} \quad f : \text{False}}{\text{Case}_{\text{False}} f : C}$$

There is no Equality Rule

False

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

False (Cont.)

- $\text{Case}_{\text{False}}$ 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 $f : \text{False}$ by $f = f$.

However that f doesn't reduce to canonical form.

– That's why it's important to carry out the **termination check in Agda**,

otherwise one obtains for instance elements of False

Finite Sets in Agda

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

```
data Colour = blue | red | green
```

- With this we obtain `red :: Colour`
- And we can define for instance

```
is_red (c :: Colour)
  :: Bool
```

```
= case c of
```

```
{ red } → tt;
  green → ff;
  blue  → ff;
```

False in Agda

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

data False =

- If we want to solve

$g \quad (x :: \text{False})$
:: Bool
= { ! ! }

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

False in Agda (Cont.)

- The result is

g ($x :: \text{False}$)
 :: Bool

= case x of { }

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

Example for the Use of False

- Assume the **type of trees**:

data Tree = pine | oak

- Below we will show, how to introduce a function

ISOak :: Tree → Set

s.t.

ISOak pine = False

ISOak oak = True

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":

$$\begin{aligned} f & (t :: \text{Tree}) \\ & (p :: \text{IsOak } t) \\ & :: A \\ & = \text{case } t \text{ of} \\ & \{ \text{pine } \rightarrow \text{case } p \text{ of } \{ \}; \\ & \text{oak } \rightarrow \dots \}; \end{aligned}$$

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 d 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.

Example 2 for the Use of False

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

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

s.t.

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

if s is the empty stack.

- Now we can define

$\text{pop } (s :: \text{Stack})$
 $(p :: \text{NotEmpty } s)$
 $:: \text{Stack}$
 $= \dots$

- 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**:

data False = true

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

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

(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**.

atom

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

atom : Bool \rightarrow Set

atom tt = True

atom ff = False

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

atom (Cont.)

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

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

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

$$\text{atom } \text{ff} = \text{False}$$

atom in Agda

```
atom (b :: Bool)
  :: Set
  = case b of
    { (tt) → True;
      (ff) → False; }
```

Decidable Predicates

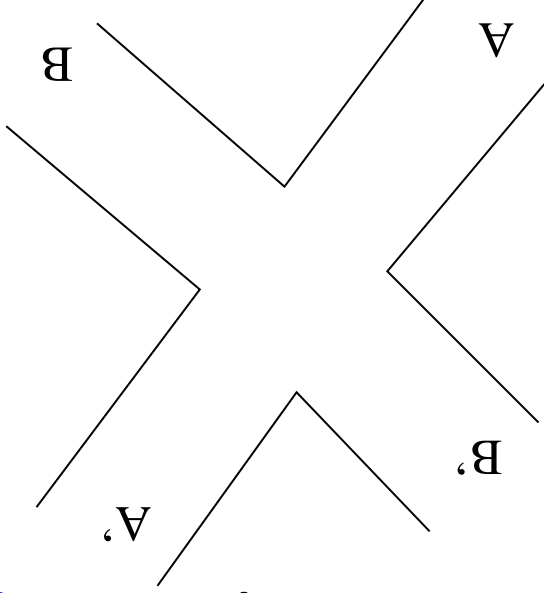
- Using atom we can now define **decidable predicates** on sets.
- Assume we have a **set of states** of a system A .
 - E.g. the set of states a railway controller can choose.
- Assume we have a function $f : A \rightarrow \text{Bool}$.
 - E.g. $f a$ means: **state a is safe**.

Decidable Predicates (Cont.)

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

data Colour = red | 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')

Phys-State :: Set
 = sig { sigA :: Colour,
 sigB :: Colour; }

The Set of Control States

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

- Each of these states **should be safe**.
- In our example, **all safe states will be captured** (this can usually be only achieved in small examples).

- A **complete set of control states** consists of:

- Allred – all signals are red.
- Agreen – signal A (and A') is green, signal B is red.
- Bgreen – signal B is green, signal A is red.

The Set of Control States (Cont.)

- We therefore define

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

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; }
```

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; }
```

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

```

Safety Predicate (Cont.)

– Now we define

$$\text{Cor } (s :: \text{Phys_State})$$

$$:: \text{Set}$$

$$= \text{CorAux } s.\text{sigA } s.\text{sigB}$$

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

Safety of the System

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

$\text{cor_proof } (s :: \text{Control_State})$
 $:: \text{Cor(phys_state } s)$
 $= \text{case } s \text{ of}$
 $\{ \text{Allred} \} \leftarrow \text{true};$
 $\text{Agreen} \leftarrow \text{true};$
 $\text{Bgreen} \leftarrow \text{true}; \}$

Safety of the System (Cont.)

- The first element true was an element of $\text{Cor}(\text{phys_state Allred})$, which reduces to **True**.

- Similarly for the other two elements.

- This works only because **each control state corresponds to a correct physical state**.

– If this hadn't been the case, we would have gotten instances where the goal to solve is **False**, which we can't solve.

Safety of the System (Cont.)

- If one makes a **mistake** which results in an unsafe situation (e.g. sets to $\text{SigB Agree} = \text{green}$, then in the last step we obtain one goal of type 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 **cor-proof Agree**), but this would be rejected by the termination check.)

(d) The Disjoint Union of Sets

Formation Rule

$$\frac{A : \text{Set} \quad B : \text{Set}}{A + B : \text{Set}}$$

Introduction Rules

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

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

Elimination Rule

$$\frac{A : \text{Set} \quad B : \text{Set} \quad C : (A + B) \rightarrow \text{Set} \quad \text{sl} : (a : A) \rightarrow C \ (\text{inl } A \ B \ a) \quad \text{sr} : (b : B) \rightarrow C \ (\text{inr } A \ B \ b)}{d : A + B}$$

$$\frac{\text{Plus-Split } A \ B \ C \ \text{sl} \ \text{sr} \ d : C \ d}{d}$$

Equality Rules

The Disjoint Union of Sets (Cont.)

$$\begin{aligned} \text{Plus-Split } A B C \text{ sl } sr \text{ (inl } A B a) &= \text{sl } a : C \text{ (inl } A B a) \\ \text{Plus-Split } A B C \text{ sl } sr \text{ (inr } A B b) &= \text{sr } b : C \text{ (inr } A B b) \end{aligned}$$

**Disjoint Union using
the Logical Framework**

- A **more compact notation** is:

- $(+)$: $\text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$, written infix.
- $\text{inl} : (A, B : \text{Set}) \rightarrow A \rightarrow (A + B)$.
- $\text{inr} : (A, B : \text{Set}) \rightarrow B \rightarrow (A + B)$.
- $\text{Plus_Split} : (A, B : \text{Set})$

- $\rightarrow (C : (A + B) \rightarrow \text{Set})$
- $\rightarrow (sl : (a : A) \rightarrow C \text{ (inl } A B a))$
- $\rightarrow (sr : (b : B) \rightarrow C \text{ (inr } A B b))$
- $\rightarrow (d : A + B)$
- $\rightarrow C d$.

Disjoint Union in Agda

- The disjoint union can be defined as a “data”-set having **two constructors** `inl` (in-left) and `inr` (in-right):

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

Disjoint Union in Agda (Cont.)

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

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

– $\text{inl}@ (A + B) :: A \rightarrow (A + B)$

– $\text{inr}@ (A + B) :: B \rightarrow (A + B)$

- This can be checked using the menu “**agda-infer-type**” in a dummy goal.
- Note that we cannot assign a type to $\text{inr}@-$.

- $(+)$ **cannot** be defined using the abbreviated data notation (which would be of the form

$\text{data } (+) = \dots$).

Disjoint Union in Agda (Cont.)

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

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

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

Disjoint Union in Agda (Cont.)

- We obtain

$$f \quad (c :: A + B) \quad :: \quad \text{Bool}$$

= case c of

$$\begin{aligned} & \{ (inl a) \} \quad \leftarrow \quad \{ ! i \}; \\ & (inr b) \quad \leftarrow \quad \{ ! i \}; \end{aligned}$$

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

Use of Concrete Disjoint Sets

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

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

- Now one can define for instance

```
isFlower (p :: Plant) :: Bool
  = case p of
    { tree t  → ff;
      flower f → tt; }
```

(e) The Σ -Set

Formation Rule

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

Introduction Rule

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

Elimination Rule

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

The Σ -Set (Cont)

Equality Rule

Sigma-Split $A B C s (p A B a b) = s a b : C (p A B a b)$

The Σ -Set using the Logical Framework

- The more compact notation is:

$$\begin{aligned} & - \Sigma : (A : \text{Set}) \\ & \quad \rightarrow (B : A \rightarrow \text{Set}) \\ & \quad \rightarrow \text{Set} . \\ & - p : (A : \text{Set}) \\ & \quad \rightarrow (B : A \rightarrow \text{Set}) \\ & \quad \rightarrow (a : A) \\ & \quad \rightarrow (b : B a) \\ & \quad \rightarrow \Sigma A B . \end{aligned}$$

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

– Sigma-Split
: (A : Set)
→ (B : A → Set)
→ (C : (Σ A B) → Set)
→ (s : (a : A, b : B a)
→ C (p A B a b))
→ (d : Σ A B)
→ C d .

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 π_0, π_1 using Sigma-Split:

$$\begin{aligned} \pi_0 &= \text{Sigma-Split } A B (\lambda x.A) (\lambda(x : A).\lambda(y : B x).x) \\ \pi_1 &= \text{Sigma-Split } A B (\lambda x.B \pi_0(x)) (\lambda(x : A).\lambda(y : B x).y) \end{aligned}$$

- On the other hand, from π_0, π_1 we can define Sigma-Split as follows:

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

The Σ -Set and the Dependent Product

- However the dependent product has the **η -rule** (which is however not implemented in Agda).

- Because of the lack of η -rule, Σ works usually **better than the dependent product** in Agda.

– I personally **don't use the dependent product** of Agda much.

The Σ -Set in Agda

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

Sigma (A :: Set)
(B :: A \rightarrow Set)
:: Set
= data p (a :: A) (b :: Ba)

The Σ -Set in Agda (Cont.)

- Again one usually defines concrete Σ -sets more directly.

- **Example:** Assume we have defined

- a set `Plant-Group` for **groups of plants** (e.g. "tree", "flower"),
- depending on $g :: \text{Plant-Group}$, sets `Plants-in-group g` for

plants in that group.

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

```
data Plant = plant (g :: Plant-Group) (pg :: Plants-in-group g)
```

The Σ -Set in Agda (Cont.)

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

$$\begin{aligned} f \quad (p :: \text{Plant}) &:: \text{Plant_group} \\ &= \text{case } p \text{ of} \\ &\{ \text{plant } g \, dg \} \rightarrow g; \{ \end{aligned}$$

Formulae in Dependent Type Theory

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

Conjunction

- $A \wedge B$ 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 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 $A \wedge B$ is the set of pairs of elements of A and B , i.e. $A \times B$.
- We can **identify** $A \wedge B$ with $A \times B$.

Conjunction (Cont.)

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

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

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

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

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

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 :

$$\frac{p : A \wedge B}{p : A} \quad \frac{p : A \wedge B}{p : B}$$

- 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** :

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

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 in p for a proof $p : A$ or an element in 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. **$A + B$.**
- We can **identify** $A \vee B$ with $A + B$.

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 .

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

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

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

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** :

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

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

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 :

$$\frac{\begin{array}{l} A : \text{Set} \\ B : \text{Set} \\ C : (A \vee B) \rightarrow \text{Set} \\ sl : (a : A) \rightarrow C \text{ (inl } A B a) \\ sr : (b : B) \rightarrow C \text{ (inr } A B b) \end{array}}{d : A \vee B} \text{Plus-Split } A B C \text{ sl sr } d : C d$$

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

$$\frac{d : A \vee B \quad sl : A \rightarrow C \quad sr : B \rightarrow C}{\text{Plus-Split } sl sr d : C}$$

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** :

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

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

Implication

- We write temporarily \supset for logical implication, in order to distinguish it from the function type \rightarrow .
 - Below we see that \supset can be identified with \rightarrow .
- $A \supset B$ is true iff, whenever A is true then B is true.
- Therefore **if there is a proof of A , there must be a proof of B** .
- Therefore a proof of $A \supset B$ is a **function, which takes a proof of A and computes a proof of B** .
- Therefore the set of proofs of $A \supset B$ is the **function type $A \rightarrow B$** .
- We can **identify** $A \supset B$ with $A \rightarrow B$.

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 $p:A$.

Implication (Cont.)

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

$$\frac{A \quad \dots \quad B}{A \supset B}$$

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 :

$$\frac{p : A \supset B \quad q : A}{p q : B}$$

- 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** :

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

Negation

- $\neg A$ has the same meaning as $A \supset \perp$ (where \perp is **absurdity** or the set False):
 - If there is no proof of A , then we can prove $A \supset \perp$.
 - If from any proof of A we can create a proof of absurdity, then there cannot be a proof of A , A must be false.
- Therefore we can identify $\neg A$ with $A \rightarrow$ **False**.

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 $\forall x : A.B$, $\exists x : A.B$.

- $\forall x : A.B$ 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** $(x : A) \rightarrow B$.

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

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 $x : A$.

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.**

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 $B[x := a]$:

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

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

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.**

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

Existential Quantification

- $\exists x : A.B$ is true iff there exists an $a : A$ such that $B[x := a]$ is true.
- Therefore a proof of $\exists x : A.B$ is a pair $\langle a, p \rangle$ consisting of an 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 $(x : A) \times B$.

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

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

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

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

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 : 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 $y : B$.

Existential Quantification (Cont.)

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

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

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

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** $A \rightarrow B$, and we have

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

!i.e. we have a proof that $\forall x : A. C(x, f\ x)$ **holds**.

- Therefore, from a proof of $\forall x : A. \exists y : B. C(x, y)$, we can **extract a function**, which computes the y from the x .

Constructive Logic (Cont.)

- We can derive as well a function which **depending on $p : A + B$ decides whether $p = \text{inl}(a)$ or $p = \text{inr}(b)$** .
- Therefore we can decide, from a proof of a disjunction, **which of the disjuncts holds**.

• Now:

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

$\forall x : \text{Turing_Machine.}(x \text{ halts} \vee \neg(x \text{ halts}))$

Constructive Logic (Cont.)

- In classical logic we **can prove the above**, since we can derive $A \vee \neg A$ (tertium non datur) for any formula A .

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

- The logic of type theory is **intuitionistic (constructive) logic**, in which $A \vee \neg A$ and $\neg\neg A \rightarrow A$ don't hold for all formulae A .

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 $\exists x : A.B$ and $A \vee B$ by the above formulae, then **all formulas provable in classical logic are derivable**.
 - This requires $(\neg\neg A) \rightarrow A$, which can be shown for all formulae built from decidable atomic formulae using $\neg, \rightarrow, \vee, \wedge$.
 - The formula $A \vee \neg A$ translates into $\neg(\neg A \wedge \neg\neg A)$, which trivially holds, since $\neg A$ and $\neg\neg A$ implies \perp .
- In this sense, **type theory contains classical logic**, but is **richer**, since it has as well so called **strong disjunction and existential quantification**.

Constructive Logic (Cont.)

- Proof (using classical logic) of

$$\exists x : A.B \leftrightarrow \neg \forall x : A.\neg B :$$

– We have classically:

$$\neg \neg A \rightarrow A :$$

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

Constructive Logic (Cont.)

- We show intuitionistically $\neg(\exists x : A.B) \leftrightarrow \forall x : A.\neg B$:

– Assume $\neg(\exists x : A.B)$, $x : A$ and show $\neg B$.

If we had B , then we had $\exists x : A.B$, contradicting $\neg(\exists x : A.B)$. 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):

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

Constructive Logic (Cont.)

- Proof of $A \vee B \leftrightarrow \neg(\neg A \wedge \neg B)$:
 - We show intuitionistically $\neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$:
 - * 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):
- $$(A \vee B) \leftrightarrow \neg(\neg(A \vee B)) \leftrightarrow \neg(\neg A \wedge \neg B)$$

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 quantification** is expressed by the original type theoretic formulae.

(f) The Set of Natural Numbers

- The set \mathbb{N} is the type theoretic representation of the set $\mathbb{N} := \{0, 1, 2, \dots\}$.
- \mathbb{N} can be generated by
 - starting with the empty set,
 - adding 0 to it, and
 - adding, whenever we have x in it $x + 1$ to it.

The Set of Natural Numbers (Cont.)

- Let S be a type theoretic notation for the operation $x \mapsto x + 1$.
- Then the type theoretic rules are

$$N : \text{Set}$$

$$0 : N$$

$$\frac{N : n \quad S}{N : n + 1}$$

Primitive Recursion

- **Primitive Recursion expresses:**

Assume we have

- $a : \mathbb{N}$.
- and, if $n : \mathbb{N}$, $x : \mathbb{N}$ then $g\ n\ x : \mathbb{N}$.

Then we can define $f : \mathbb{N} \leftarrow \mathbb{N}$, s.t.

- $f\ 0 = a$,
- $f\ (S\ n) = g\ n\ (f\ n)$.

Primitive Recursion (Cont.)

- The **computation of f on n** proceeds now as follows:

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

- * We assume that we have determined already how to compute f on n' .
- * Now f on n reduces to g on n' (f on n').
- * g on n' (f on n') can be computed, since we know how to compute g on n' .
- f on n' .

Example

- The function $f : \mathbb{N} \rightarrow \mathbb{N}$ with $f(x) = 2 \cdot x$ can be defined **recursively** by:

$$\begin{aligned} - & f(0) = 0. \\ - & f(Sn) = S(fn). \end{aligned}$$

- Therefore take in the definition above:

$$\begin{aligned} - & a = 0, \\ - & g\ n\ x = S(S\ x). \end{aligned}$$

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

$$f : \mathbb{N} \leftarrow C$$

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

Rules for the Natural Numbers

Formation Rule

$$N : \text{Set}$$

Introduction Rules

$$0 : N$$

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

Elimination Rule

$$\frac{C : N \rightarrow \text{Set} \quad a : C \quad 0 : C \quad f : (x : N) \rightarrow C \rightarrow C \rightarrow C(S x)}{P C a f n : C n}$$

Equality Rules

$$P C a f 0 = a$$

$$P C a f (S n) = f n (P C a f n)$$

Rules for the Natural Numbers (Cont.)

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

$$g n : C n$$

which means that

$$\lambda(n : N).g n : n \leftarrow C n .$$

- The equality rules read:

$$g 0 = a$$

$$g (S n) = f n (g n)$$

Rules for N
using the Logical Framework

- The more compact notation is:

$$\begin{array}{l}
- N : \text{Set}, \\
- 0 : N, \\
- S : N \rightarrow N, \\
- P : (C : N \rightarrow \text{Set}) \rightarrow C 0 \\
\leftarrow C 0 \\
\leftarrow ((x : N) \rightarrow C x \rightarrow C (S x)) \\
\leftarrow (N : N) \rightarrow \\
\leftarrow C n .
\end{array}$$

Natural Numbers in Agda

- N is defined using **data**:

```
data N = Z | S(n :: N)
```

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

- Therefore we have

```
Z :: N  
S :: N → N
```

Elimination Rules for N in Agda

- Elimination works via case distinction in Agda.

– If we want to introduce

$$f \ (n :: N) \ :: \ A \ = \ \{ i \ i \}$$

* A possibly depending on n ,

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

We get

$$f \ (n :: N) \ :: \ A$$

= case n of

$$\{ (Z) \} \ \leftarrow \ \{ i \ i \} \ \leftarrow \ (S \ n') \ \{ i \ i \} \ \leftarrow \ \{ i \ i \}$$

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 in case $n = S n'$** .then **agda-check-termination succeeds**.

Elimination Rules for N in Agda (Cont.)

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

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:

$$\text{one} := S Z):$$

$$\text{fb } (n :: N) :: N$$

= case n of

$$\{ Z \}$$

← one;

$$(S n')$$

← case n' of

$$\{ Z \}$$

← one;

$$(S n'')$$

← fb n' + fb n'';

$$\{ S n'' \}$$

Example for Limitations of Termination Check

- Assume we define the **predecessor function**

$$\text{pred} (n :: N) :: N = \text{case } n \text{ of } \{ Z \} \leftarrow Z; \{ S n' \} \leftarrow n'; \{ \}$$

i.e.

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

Example for Limitations of Termination Check (Cont.)

- Then the function

$$\begin{aligned}
 f &:: N \rightarrow Z \\
 f &= \text{case } n \text{ of} \\
 &\quad \{ Z \rightarrow Z, \\
 &\quad S\ n' \rightarrow f(\text{pred } n) \}
 \end{aligned}$$

terminates always

- (it returns for all $n : N$ the value Z).

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

Limitations of the Termination Check (Cont.)

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

Example: Addition

- Definition of $+$ in Agda:

$$(+) (n, m :: N) :: N$$

= case m of

{ (Z) \leftarrow

$n;$

(S m') \leftarrow S ($n + m'$); }

- The definition expresses:

$$n + 0 = n$$

$$n + (m + 1) = (n + m) + 1$$

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** $(+) n m$ since m' is introduced before m .

Example: Multiplication

- Definition

$$(*) \quad (n, m :: N) :: N$$

= case m of

$$\{ (Z) \} \leftarrow Z;$$

$$(S \ m') \leftarrow n * m' + n;$$

- The definition expresses:

$$n \cdot 0 = 0$$

$$n \cdot (m + 1) = (n \cdot m) + n$$

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**.

Equality on \mathbb{N}

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

- $(Z == Z) = \text{True}$.
- $(Z == S n) = (S n == Z) = \text{False}$.
- $(S n == S m) = (n == m)$.

Equality on N (Cont.)

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

$$(==) :: \text{Set} \rightarrow \text{Set}$$

$$= \text{case } n \text{ of}$$

$$\{ Z \} \rightarrow$$

$$\text{case } m \text{ of}$$

$$\{ Z \} \rightarrow$$

$$\{ i \} \rightarrow$$

$$\{ i \} \rightarrow$$

$$(S \ m')$$

$$\rightarrow$$

$$(S \ n')$$

$$\rightarrow$$

$$\text{case } m \text{ of}$$

$$\{ Z \} \rightarrow$$

$$\{ i \} \rightarrow$$

$$\{ i \} \rightarrow$$

$$(S \ m')$$

$$\rightarrow$$

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

Reflexivity of ==

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

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

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

$$\text{refl} (n : \mathbb{N}) :: n == n$$

$$\{ i \} =$$

- Task of Coursework 3, Question 1 (e) to solve this goal.

Reflexivity of == (Cont.)

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

$$\text{refl } (n : \mathbb{N}) \quad :: \quad n == n$$

$$= \text{case } n \text{ of}$$

$$\{ (Z) \leftarrow \{ i ; i \} \quad \leftarrow \{ i ; i \} \}$$

Reflexivity of $=$ (Cont.)

- Case $n = Z$ is trivial.
- Case $n = S n'$ can be solved using $\text{refl } n'$ (which is defined before $\text{refl } n$).

Symmetry of ==

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

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

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

$$\text{sym} (n, m : \mathbb{N}) (d :: n == m) \\ = \{ i \} i :: m == n$$

Symmetry of == (Cont.)

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

$$\begin{array}{l}
 \text{sym } (n, m : \mathbb{N}) \\
 d :: n == m \\
 :: m == n \\
 = \text{ case } n \text{ of} \\
 \{ (Z) \leftarrow \text{ case } m \text{ of} \\
 \{ (Z) \leftarrow \{ i \}; \\
 (S\ m') \leftarrow \{ i \}; \} \\
 (S\ n') \leftarrow \text{ case } m \text{ of} \\
 \{ (Z) \leftarrow \{ i \}; \\
 (S\ m') \leftarrow \{ i \}; \} \\
 \{ i \}; \}
 \end{array}$$

Symmetry of == (Cont.)

- The **first goal** can be solved by using true (since $Z == Z$) = True).
- For the **second goal** we know p is an element of $Z == S m'$ which is False.
 - Therefore if we make **case distinction on p** we get
case p of { }
- Similarly the **third goal can be solved.**
and have solved the second goal.

Symmetry of == (Cont.)

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

- Define first

data Nil = nil

Cons (A, B :: Set)

:: Set

= data cons(a :: A)(b :: B)

Example: Tuples of Length n

Example: Tuples of Length n

- Now we can define

$$\begin{aligned} \text{Vec } (A :: \text{Set}) & (n :: N) &:: \text{Set} &= \text{case } n \text{ of} \\ & \{ (Z) &\leftarrow \text{Nil}, \\ & (S m') &\leftarrow \text{Cons } A \text{ (Vec } A \text{ m')} \} \end{aligned}$$

Remarks on Tuples of Length n

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

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

Remarks on Tuples of Length n

- If we define

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

then this reads:

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

Remarks on Tuples of Length n (Cont.)

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

- Define

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

Example: Sum of Tuples of Length n

Example: Componentwise Sum of Tuples of Length n

- We define **component-wise sum of tuples of length n**.

– Using mathematical notation, this sum for instance as follows:

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

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

$\text{SumNVec } (n :: N)$
 $\text{:: NVec } n$
 $(\text{vec}, \text{vec} :: \text{NVec } n)$
 $=$ case n of
 $\{ (Z) \rightarrow \text{nil};$
 $(S n') \rightarrow$
case vec of
 $\{ (\text{cons } a \text{ vec}') \rightarrow$
case vec of
 $\{ (\text{cons } b \text{ vec}') \rightarrow$
 cons@-
 $(a + b)$
 $(\text{SumNVec } n' \text{ vec}' \text{ vec}') :: \} \}$

(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:

$$\begin{aligned} \text{list} & (A :: \text{Set}) \\ & :: \text{Set} \\ & = \text{data nil} \\ & | \text{cons}(a :: A) (l :: \text{list } A) \end{aligned}$$

Elimination Rule for Lists

- Elimination rule uses list-recursion:
Assume

– $A : \text{Set}$

– $C :: \text{Set}, \text{depending on } l :: \text{list } A.$

Then we can define

$$\begin{aligned}
 f \quad (l :: \text{list } A) &:: C \\
 &= \text{case } l \text{ of} \\
 \{ \text{nil} \} &\leftarrow \{ i; \} \\
 (\text{cons } a \ l') &\leftarrow \{ i; \} \{ i; \}
 \end{aligned}$$

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

Example: Length of a List

length (l :: list N) =
:: N
= case l of
{ (nil) → Z;
 (cons a l') → S (length l') }

Example: Sum of the Elements of a List

$$\begin{aligned} \text{sumlist } (l :: \text{list } N) &:: N \\ &= \text{case } l \text{ of} \\ &\quad \{ \text{nil} \} \leftarrow Z; \\ &\quad (\text{cons } a \ l') \leftarrow S \ (n + \text{sumlist } l'); \end{aligned}$$

Interesting Exercise

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

(h) Universes.

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

Rules for the Universe

Formation Rule

$$U : \text{Set}$$

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

Introduction and Equality Rules

$$\begin{array}{l} \widehat{\text{Fin}}_0 : U \\ \widehat{\text{Fin}}_0(T(\widehat{\text{Fin}}_0)) = \text{Fin}_0 : \text{Set} \end{array}$$

$$\begin{array}{l} \widehat{\text{Fin}}_1 : U \\ \widehat{\text{Fin}}_1(T(\widehat{\text{Fin}}_1)) = \text{Fin}_1 : \text{Set} \end{array}$$

$$\begin{array}{l} \widehat{\text{Bool}} : U \\ \widehat{\text{Bool}}(T(\widehat{\text{Bool}})) = \text{Bool} : \text{Set} \end{array}$$

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

$$\frac{a : U \quad b : U}{a \doteq b : U}$$

$$\mathbb{T}(a \doteq b) = \mathbb{T}(a) + \mathbb{T}(b) : \text{Set}$$

$$\frac{a : U \quad b : \mathbb{T}(a) \leftarrow U \quad \Sigma(a, b) : U}{\Sigma(a, b) : U}$$

$$\mathbb{T}(\Sigma(a, b)) = \Sigma(\mathbb{T}(a), \mathbb{T}(b)) : \text{Set}$$

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

$$\frac{a : U \quad \overline{b : T(a)} \quad \overline{\Pi(a, b)} : U}{a : U \quad \overline{b : T(a)} \quad \overline{\Pi(a, b)} : U}$$

$$T(\overline{\Pi}(a, b)) = (x : T(a)) \rightarrow T(b \ x) : \text{Set}$$

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 U) and are not very much used.
- They follow the principles present in previous rules.

Applications of the Universe

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

Applications of the Universe

- Example: Define

$$\begin{aligned} \underbrace{\text{atom}} &: \text{Bool} \rightarrow \mathbb{U}, \\ \underbrace{\text{atom}} &:= \text{Case}_{\text{Bool}} (\lambda(x : \text{Bool}). \mathbb{U}) \underbrace{\text{Fin}_1} \underbrace{\text{Fin}_0}, \\ \text{atom} &: \text{Bool} \rightarrow \text{Set}, \\ \text{atom} &: \lambda(x : \text{Bool}). \mathbb{T} \underbrace{(\text{atom } x)}, \end{aligned}$$

Then

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

Universes in Agda

- \mathbb{U} and \mathbb{I} 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.

Universes in Agda (Cont.)

mutual
U :: Set
= data Nhat
| Finzerohat
| Finonehat
| Boolhat
| Sigmahat (a :: U)(b :: T a → U)
| Pihat (a :: U)(b :: T a → U)

Universes in Agda (Cont.)

\mathbb{T} in the following is to be intended the same as \mathbb{U} :

$\mathbb{T} (u :: \mathbb{U}) :: \text{Set}$

= case u of

{ (Nhat)

←

$N;$

←

FInzero;

←

FInone;

←

Bool;

←

Sigma ($\mathbb{T} a$) ($\lambda(x :: \mathbb{T} a) \rightarrow \mathbb{T} (b x)$);

←

($x :: \mathbb{T} a$) $\rightarrow \mathbb{T} (b x)$; }

(Pihat $a b$)

(SigmaHat $a b$)

(BoolHat)

(FInoneHat)

(FInzeroHat)

(!) 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.

$$\begin{aligned}
 A &:: \text{Set} \\
 &= \text{data } C_1(a_{11} :: A_{11}) \dots (a_{1n_1} :: A_{1n_1}) \\
 &\quad | C_2(a_{21} :: A_{21}) \dots (a_{2n_2} :: A_{2n_2}) \\
 &\quad \dots \\
 &\quad | C_m(a_{m1} :: A_{m1}) \dots (a_{mn_m} :: A_{mn_m})
 \end{aligned}$$

Meaning of "data"

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

$$\begin{aligned}
 C_1 @ A &:: (a_{11} :: A_{11}) \\
 &\dots \\
 &\rightarrow (a_{in_1} :: A_{in_1}) \\
 &\rightarrow A
 \end{aligned}$$

where a constructor always constructs new elements.

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

Strictly Positive Algebraic Data Types

- In the types $A_{i,j}$ we can make use of A .

- However, it is difficult to understand A , if we have **negative** occurrences of A .

- Example:

$A :: \text{Set}$

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

- What is the least set A having a constructor

$C @ A :: (f :: A \rightarrow A)$

$\rightarrow A \quad ?$

Strictly Positive Algebraic Data Types (Cont.)

- If we
 - * have constructed some part of A already,
 - * find a function $f :: A \rightarrow A$, and
 - * add $\text{C@}_- f$ to A ,then f might no longer be a function $A \rightarrow A$.
(f applied to the new element $\text{C@}_- f$ 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.

Strictly Positive Algebraic Data Types (Cont.)

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

$\text{Nlist} :: \text{Set}$

$= \text{data nil}$

$| \text{cons } (a :: \text{N})$

$(l :: \text{Nlist})$

- The constructor cons_- of N-lists refers to Nlist , but in a positive way:

We have: if $a :: \text{N}$ and $l :: \text{Nlist}$, then we have $\text{cons}_- a l :: \text{Nlist}$.

– If we add $\text{cons}_- a l$ to Nlist , the reason for adding it (namely $l :: \text{Nlist}$) 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.

Strictly Positive Algebraic Data Types (Cont.)

- In general:

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

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 N were strictly positive 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 simultaneously inductively.
- Example: the even and odd numbers:

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

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

Strictly Positive Algebraic Data Types, Extensions

- We can even allow $A_{1j} = B_1 - > A$ or even $A_{1j} = B_1 - > \dots - > B_1 - > A$, where A is one of the types introduced simultaneously.

- Example (called "Kleene's 0"):

```

0 :: Set
  = data leaf
    | succ (o :: 0)
    | lim (f :: N -> 0)

```

- The last definition is unproblematic, since, if we have $f :: N -> 0$ and construct $\text{lim } f$ out of it, adding this new element to 0 doesn't destroy the reason for adding it to 0.

- So again 0 can be "constructed".

Elimination Rules for data

- Functions from strictly positive data types can now be defined by case distinction as before.
- For termination we need only that in the definition of f , when have to define $f (C@_ a_1 \dots a_n)$, we can refer only to f applied to elements used in $C@_ a_1 \dots a_n$.

• For instance

– in the Bintree example, when defining

$f :: Bintree \rightarrow A$

by case-distinction, then the definition of

f (branch @_left right)

can make use of f left and f right.
– In the example of 0, when defining

$g :: 0 \rightarrow A$

by case-distinction, then the definition of

g (lim @_f)

can make use of g (f n) for all $n :: \mathbb{N}$.