

Change of Plan for Stream B

- New plan, stream B:
 - B1. Introduction.
 - B2. [Data types](#).
 - B3. Propositions as types.
 - B4. Interactive programs in dependent type theory.
 - B5. Case studies.
- The logical framework is the type theory introduced in B1.

B2. Data Types

- (a) The set of Booleans.
- (b) The finite sets.
- (c) The disjoint union of sets.
- (d) The Σ -set.
 - Addendum: Atomic Formulas and the traffic light exam
- (e) The Π -set.
- (f) The set of natural numbers.
 - Addendum: Lists.
- (g) Universes.
- (h) The W -set (might be omitted).
- (i) Inductive data sets.

General Remarks

- In the following we will introduce the rules for several set constructions.
So we will always introduce new elements of `Set` (which therefore are elements of `Type` as well).
- For each of these sets we will introduce
 - formation rules,
 - introduction rules,
 - elimination rules,
 - equality rules.
- We will look then for each such set, how it can be defined in Agda. Agda has a much more general concept, and we will explore this generality in (i).

(a) The Set of Booleans

Formation Rule

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

Introduction Rules

$\text{true} : \text{Bool}$

$\text{false} : \text{Bool}$

Elimination Rule

$$\frac{\begin{array}{l} C : \text{Bool} \rightarrow \text{Set} \\ ic : C \text{ true} \\ ec : C \text{ false} \\ con : \text{Bool} \end{array}}{\text{then_else_if } C \text{ } ic \text{ } ec \text{ } con : C \text{ } con}$$

Equality Rules

$$\frac{\begin{array}{l} C : \text{Bool} \rightarrow \text{Set} \\ ic : C \text{ true} \\ ec : C \text{ false} \end{array}}{\text{then_else_if } C \text{ } ic \text{ } ec \text{ true} = ic : C \text{ true}}$$
$$\frac{\begin{array}{l} C : \text{Bool} \rightarrow \text{Set} \\ ic : C \text{ true} \\ ec : C \text{ false} \end{array}}{\text{then_else_if } C \text{ } ic \text{ } ec \text{ false} = ec : C \text{ false}}$$

Remarks

- In the above
 - *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:
 - $\text{then_else_if} : (C : \text{Bool} \rightarrow \text{Set}) \rightarrow (ic : C \text{ true}) \rightarrow (ec : C \text{ false}) \rightarrow (cond : \text{Bool}) \rightarrow C \text{ cond}$.
- true, false are called the constructors of Bool.

Remarks (Cont.)

- Notice that we then get for $C : \text{Bool} \rightarrow \text{Set}$,
 $ic : C \text{ true}$, $ec : C \text{ false}$
 - $f := \text{then_else_if } C \text{ } ic \text{ } ec$,
 $: (cond : \text{Bool}) \rightarrow C \text{ } cond$
 - $f \text{ true} = ic : C \text{ true}$,
 - $f \text{ false} = ec : C \text{ false}$.
- That's why the use of `then_else_if` instead of `an` at first sight more natural construction

`if_then_else C con ic ec`

is more convenient:

It's often useful to obtain an f as above.

Bool in Agda

- We introduce Bool by simply listing its constructors:

```
Bool :: Set
      = data true | false
```

- Then we have new constants

```
- true@Bool :: Bool
- false@Bool :: Bool
```

- The type Bool is used, since we can define other sets using the same constructors eg.

```
Test :: Set
      = data true | false | undefined
```

- Since often the machine can work out the type directly, one can replace @Bool by @_ ie. we have

```
- true@_ :: Bool
- false@_ :: Bool
```

Case Distinction

- The elimination rule in Agda is based on case distinction.
 - Assume we want to define
 - * $f : \text{Bool} \rightarrow \text{Bool}$, s.t.
 - * $f \text{ true} = \text{false}$,
 - * $f \text{ false} = \text{true}$.
 - So we have the goal:
$$\begin{aligned} & f \text{ (x :: Bool)} \\ & \quad :: \text{Bool} \\ & \quad = \{! \quad !\} \end{aligned}$$
 - We can then type into the goal x and choose the menu item “agda-case”.
 - * This allows now case distinction by what x can be:
it can be true or false.
 - The goal expands to:
$$\begin{aligned} & f \text{ (x :: Bool)} \\ & \quad :: \text{Bool} \\ & \quad = \text{case } x \text{ of } \{ \\ & \quad \quad (\text{true}) \rightarrow \{! \quad !\}; \\ & \quad \quad (\text{false}) \rightarrow \{! \quad !\}; \} \end{aligned}$$

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” \approx result of reducing the term.
and type into the mini-buffer x .
 - One gets the answer
`true@Bool.`
- Alternatively, check, the cursor being in that goal, the context
 - (use goal-menu “agda-context”):
It contains

`x :: Bool = true@_.`
- Similarly one finds that in the second goal x is
`false@_.`

Case Distinction (Cont.)

- Now we can solve the new goals by inserting
 - `false@_` (or `false@Bool`) into the first one,
 - `true@_` into the second one.
- We obtain our desired function:

```
f (x :: Bool)
  :: Bool
  = case x of {
      (true) -> false@_;
      (false) -> true@_;}
```

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 `true` or `false`.
 - Not every place in the buffer is a good place.
 - Good places for context are goals, and that’s the only places where Agda allows us to compute the WHNF of expressions.
- So we
 - type in a dummy goal:
`test :: Set`
 `= {! !}`
 - move to the new goal
 - choose “agda-compute-WHNF”,
 - and type into the mini-buffer `f true@_`
- The result shown is `false@_`.

(b) The Finite Sets

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

Formation Rule

$$N_n : \text{Set}$$

Introduction Rules

$$A_k^n : N_n$$

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

Elimination Rule

$$\frac{\begin{array}{l} C : N_n \rightarrow \text{Set} \\ s_0 : C A_0^n \\ s_1 : C A_1^n \\ \vdots \\ s_{n-1} : C A_{n-1}^n \\ a : N_n \end{array}}{\text{Case}_n C s_0 \dots s_{n-1} a : C a}$$

The Finite Sets (Cont)

Equality Rules

$$\frac{\begin{array}{l} C : \mathbb{N}_n \rightarrow \text{Set} \\ s_0 : C A_0^n \\ s_1 : C A_1^n \\ \vdots \\ s_{n-1} : C A_{n-1}^n \end{array}}{\text{Case}_n C s_0 \dots s_{n-1} A_k^n = s_k : C A_k^n}$$

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:

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

We sometimes even omit the type:

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

More compact elimination rules

- $\text{Case}_n : (C : \mathbb{N}_n \rightarrow \text{Set}) \quad .$
 - $\rightarrow (s_0 : C \ A_0^n)$
 - $\rightarrow \dots$
 - $\rightarrow (s_{n-1} : A_{n-1}^n)$
 - $\rightarrow (a : \mathbb{N}_n)$
 - $\rightarrow C \ a$

Finite Sets in Agda

- Finite sets can be introduced by giving one constructor for each element. Eg.

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

– Then we have for instance

```
red@_ :: Colour
```

– And we can define for instance

```
is_red (c :: Colour)
  :: Bool
  = case c of {
      (red) -> true@_;
      (green) -> false@_;
      (blue) -> false@_;}
```

Special Case $n=0$

In the case $n = 0$ we get the following rules:

Formation Rule

$N_0 : \text{Set}$

There is no introduction rule.

Elimination Rule

$$\frac{C : N_0 \rightarrow \text{Set} \quad e : N_0}{\text{Case}_0 C e : C e}$$

There is no equality rule.

Special Case $n=0$ (Cont)

- N_0 is the empty set.
 - The empty set has no elements.
 - There is a function from the empty set into every set.
 - * It chooses an element of the empty set (there is none) and creates an element of the range.

Change of Plan for Stream B

(The distribution of this slide was forgotten when distributing the first part of B2).

- New plan, stream B:

B1. Introduction.

B2. [Data types](#).

B3. Propositions as types.

B4. Interactive programs in dependent type theory.

B5. Case studies.

- The logical framework is the type theory introduced in B1.

The Empty Set in Agda

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

```
Empty :: Set
      = data
```

- If we want to solve

```
g (x :: Empty)
  :: Bool
  = {!  !}
```

we can type `x` into the goal and choose menu-item “agda-case”.

- The result is

```
g (x :: Empty)
  :: Bool
  = case x of { }
```

- The goal is solved.

(c) 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

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

The Disjoint Union of Sets (Cont.)

Equality Rules

$$\text{Plus_Split } A B C \text{ } sl \text{ } sr \text{ } (\text{inl } A B a)$$
$$= sl \ a : C \text{ } (\text{inl } A B a)$$
$$\text{Plus_Split } A B C \text{ } sl \text{ } sr \text{ } (\text{inr } A B b)$$
$$= sr \ b : C \text{ } (\text{inr } A B b)$$

Disjoint Union using the Logical Framework

- The more compact notation is:
 - $+ : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$, written infix.
 - $\text{inl} : (A, B : \text{Set}) \rightarrow A \rightarrow A + B$.
 - $\text{inr} : (A, B : \text{Set}) \rightarrow B \rightarrow A + B$.
 - 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` and `inr`:

```
(+) (A :: Set)
    (B :: Set)
    :: Set
    = data inl (a :: A) | inr (b :: B)
```

- Because of the use of `(+)`, we can use `+` infix.
- Now we have, if `A, B :: Set`
 - * `inl@(A+B) :: A -> (A + B)`
 - * `inr@(A+B) :: B -> (A + B)`
 - * This can be checked using the menu “agda-infer-type” in a dummy goal.
 - * Note that the type of `inr@_` cannot be inferred.

Disjoint Union in Agda (Cont.)

- Elimination works via case distinction.
So if want to define for $A, B :: \text{Set}$

```
f (c :: A + B)
  :: Bool
= {!  !}
```

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

```
f (c :: A+B)
  :: Bool
= case c of {
      (inl a) -> {!  !};
      (inr b) -> {!  !};}
```

and insert into the first goal eg. true and the second one false

Use of Concrete Disjoint Sets

- It is often more convenient to define concrete disjoint unions directly using more intuitive names for constructors, eg.

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

- If we translate it back into rules we get the following:
 - $\text{Plant} : \text{Set}$.
 - $\text{tree} : \text{Tree} \rightarrow \text{Plant}$.
 - $\text{flower} : \text{Flower} \rightarrow \text{Plant}$.
- and if we use concrete elimination rules we get for instance:
 - $\text{isflower} : \text{Plant} \rightarrow \text{Bool}$,
 - $\text{isflower}(\text{tree } t) = \text{false}$,
 - $\text{isflower}(\text{flower } f) = \text{true}$.

(d) The Σ -Set

Formation Rule

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

Introduction Rule

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

Elimination Rule

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

The Σ -Set (Cont)

Equality Rule

$$\begin{aligned} \text{Sigma_Split } A B C s (p A B a b) \\ = s a b : C (p A B a b) \end{aligned}$$

Disjoint Union using the Logical Framework

- The more compact notation is:

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

The Σ -Set and the Dependent Product

- The dependent product and the Σ -set are very similar.
 - One can define the projections π_0 , π_1 using `Sigma_Split`.
 - On the other hand, from π_0 , π_1 we can define `Sigma_Split` as:

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

- However the dependent product has the η -rule (not implemented in Agda).
- Because of the lack of η -rule, Σ works usually better than the dependent record set in Agda.
 - * I personally don't use the dependent record set much.

The Σ -Set in Agda

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

```
Sigma (A :: Set)
      (B :: A -> Set)
      :: Set
      = data p (a :: A)(b :: B a)
```

- Again one usually defines concrete sets more directly eg.

```
Plant :: Set
      = data form (g :: Plant_Group)
                 (n :: Group_plants g)
```

- Not surprisingly, for elimination we use case distinction:

```
f (plant :: Plant)
  :: Plant_group
  = case plant of {
      (form g n) -> g;}
```

Addendum: Atomic Formulas and the Traffic Light Example

Atomic Formulas

- Atomic Formulas are in this lecture sets corresponding to Boolean values.

We have:

– True is the one element set.

* In Agda:

```
True :: Set
      = data true
```

* In general type theory: $\text{True} = N_1$.

* True is inhabited means True is provable.
true is a proof of True.

– False is the empty set.

* In Agda:

```
False :: Set
       = data
```

* In general type theory: $\text{False} = N_0$.

* False is not inhabited means
False is not provable.

atom

- We can define now:

$$\begin{aligned} \text{atom} & : \text{Bool} \rightarrow \text{Set} \\ \text{atom true} & = \text{True} \\ \text{atom false} & = \text{False} \end{aligned}$$

- In Agda:

```
atom (b :: Bool)
  :: Set
  = case b of
      { (true) -> True;
        (false) -> False; }
```

.

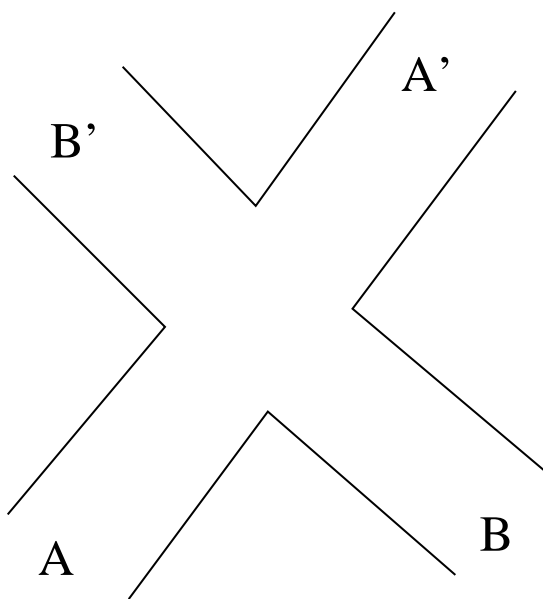
- in general type theory we have to add new rules,
 - $\text{atom} : \text{Bool} \rightarrow \text{Set}$,
 - $\text{atom true} = \text{True}$,
 - $\text{atom false} = \text{False}$,or we can make use of a universe (see later).

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.
- 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 crossing of two lights, controlled by a traffic light:



- Assume from each direction A, A', B, B' there is one traffic light,
 - but A and A', B and B' resp. coincide.

The Set of Physical States

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

```
Colour :: Set
      = data red | green
```

- The set of physical states of the systems determines the state of the pair (A,A') and of (B,B') of traffic lights:

```
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.
- So we have

```
Control_State :: Set
               = data allred
                   | Agreen
                   | Bgreen
```

Mapping Control States to Physical States

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

```
sigA (s :: Control_State)
  :: Colour
  = case s of
      { (allred) -> red@_;
        (Agreen) -> green@_;
        (Bgreen) -> red@_; }
```

```
sigB (s :: Control_State)
  :: Colour
  = case s of
      { (allred) -> red@_;
        (Agreen) -> red@_;
        (Bgreen) -> green@_; }
```

- Now we can define

```
phys_state(s :: Control_State)
  :: Phys_State
  = struct{SigA = sigA s;
           SigB = sigB s;}
```

Correctness Predicate

- We define now the correctness of a physical state. It is true iff not both signals are green. We define it directly, without defining first a Boolean function:
- We first define a Predicate on two signals:

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

- Now we define

```
Cor (s :: Phys_State)
    :: Set
    = CorAux s.SigA s.SigB
```

Correctness of the System

- Now we show that all control states are safe:

```
cor_proof(s :: Control_State)
  :: Cor (phys_state s)
= case s of
    {(allred) -> true@_;
     (Agreen) -> true@_;
     (Bgreen) -> true@_;}
```

(d) The Π -Set

This part, which is of minor importance, was omitted.

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

$$\mathbb{N} : \text{Set}$$
$$0 : \mathbb{N}$$
$$\frac{n : \mathbb{N}}{S\ n : \mathbb{N}}$$

Primitive Recursion

- Primitive Recursion expresses:

Assume we have

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

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

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

- In order to compute $f\ n$, we can now proceed as follows:

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

Generalized Primitive Recursion

- We can generalize primitive recursion as follows:
 - First we can replace the range of f by an arbitrary set C
 - * ie. allow
$$f : \mathbb{N} \rightarrow C$$
 - Further, C can now depend on \mathbb{N} .
- We obtain the following set of rules:

Rules for the Natural Numbers

Formation Rule

$\mathbb{N} : \text{Set}$

Introduction Rules

$0 : \mathbb{N}$

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

Elimination Rule

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

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 $g := \text{P } C \text{ } f$ then
 - The conclusion of the elimination rule reads:

$$g \ n : C \ n$$

which means that g is the function introduced by extended primitive recursion.

- The equality rules read:

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

Rules for N using the Logical Framework

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

Natural Numbers in Agda

- Again we we can define \mathbb{N} using data:

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

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

- So we have
 - $Z@_ :: \mathbb{N}$
 - $S@N :: \mathbb{N} \rightarrow \mathbb{N}$

- It's useful to define

```
Z :: N
  = Z@_
```

```
S :: N -> N
  = S@N
```

S@N in Alfa

- Unfortunately, Alfa is not fully compatible with Agda:
 - Alfa doesn't understand S@N as having type $N \rightarrow N$.
 - Alfa understands only $S@_n :: N$ for $n :: N$.
 - So if the Agda-correct code above is submitted to Alfa, it returns an error message.
 - A work-around is to define instead
$$S (x :: N) :: N = S@_x$$
or equivalently
$$S N \rightarrow N = \lambda (x :: N) \rightarrow S@_x$$

Elimination Rules for N in Agda

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

```
f (n :: N)
  :: A
  = {!  !}
```

* (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) -> {!  !}};
      (S n') -> {!  !}};
```

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 make
 - no use of f in the case $n=Z$ and
 - only use of $f\ n'$ in the case $n = S\ n'$then agda-check-termination succeeds.
- 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 and `agda-check-termination` accepts it:

```
fib(n::N)
  :: N
  = case n of
    {(Z) -> one;
     (S n')
    -> case n' of
      {(Z) -> one;
       (S n'')
      -> fib n' + fib n''};};}
```

Example for Limitations of Termination Check

- If we define the predecessor function

```
pred (n :: N)
  :: N
  = case n of
      { (Z) -> Z;
        (S n) -> n; }
```

i.e

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

then the function

```
f (n :: N)
  -> N
  = case n of
      { (Z) -> Z;
        (S n') -> f (pred n); }
```

terminates always

– (it returns for all $n :: N$ the value Z),

but 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

(+) (n :: N)
 (m :: N)
 :: N
= case m of
 {(Z) -> n;
 (S m') -> S (n+m');}

- The definition expresses:

$$n + 0 = n$$
$$n + (m + 1) = (n + m) + 1$$

- Note that (+) is used infix (n+m for (+) n m).
- If $m = S m'$, the definition of (+) n m refers to (+) n m', which is defined before (+) n m.

Example: Multiplication

- Definition

```
(*) (n :: N)
     (m :: N)
     :: N
     = case m of
         { (Z) -> Z;
           (S m') -> n * m' + n; }
```

- The definition expresses:

$$n \cdot 0 = 0$$
$$n \cdot (m + 1) = (n \cdot m) + n$$

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

Example: Vectors of Length n

- Define first

```
Nil  :: Set
     = data nil
```

```
nil  :: Nil
     = nil@_
```

```
Cons (A :: Set)
     (B :: Set)
     :: Set
     = data cons (a :: A) (b :: B)
```

- Now

```
Vec(A :: Set)
   (n :: N)
   :: Set
   = case n of
       {(Z) -> Nil;
        (S m') -> Cons A (Vec A m')};}
```

Remarks on Vectors of Length n

- In ordinary mathematics, we would define

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

- 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:

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

Remarks on Vectors of Length n (Cont)

- In our 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.

Example: Sum of Vectors of Length n

- Define

```
NVec(n::N)
  :: Set
  = Vec N n
```

- Then

```
SumNVec
  (n::N)
  (avec :: NVec n)
  (bvec :: NVec n)
  :: NVec n
= case n of
  {(Z) -> nil@_;
   (S n')}
  -> case avec of
    {(cons a avec')}
      -> case bvec of
        {(cons b bvec')}
          -> cons@_
            (a + b)
            (SumNVec n' avec' bvec')
          ;};};}
```

Equality on N

- An equality $\text{Eqnat}(n,m) :: \text{Set}$ for $n,m :: \mathbb{N}$ can be defined using the equations:
 - $\text{Eqnat } Z \ Z = \text{True}$.
 - $\text{Eqnat } Z \ (S \ n) = \text{Eqnat } (S \ n) \ Z = \text{False}$.
 - $\text{Eqnat } (S \ n) \ (S \ m) = \text{Eqnat } n \ m$.
 - Use case distinction on the first and second argument of Eqnat .
(See `labsession3`).
- Now we can express properties like:
Equality is reflexive, ie. for all n we have n is equal to itself:

```
refl(n::N)
  :: Eqnat n n
  = case n of
    {(Z) -> true@_;
     (S n') -> refl n';}
```

Proof of Symmetry

- Equality is symmetric, ie. $n = m$ implies $m = n$:

```
sym (n::N)
    (m::N)
    (p:: Eqnat n m)
    :: Eqnat m n
= case n of
    {(Z)
     -> case m of
         {(Z) -> true@_;
          (S m') -> case p of { };};
     (S n')
     -> case m of {
         {(Z) -> case p of { };
          (S m')
          -> sym n' m' p;};};}
```

Addendum: 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:

```
list (A :: Set)
  :: Set
= data nil
      | cons (a :: A)
              (l :: list A)
```

Elimination Rule for Lists

- Elimination rule uses list-recursion:

Assume

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

Then we can define

```
f (l :: list A)
  :: C
= case l of
  {(nil) -> {! !}};
  (cons a l') -> {! !}};
```

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

Examples for Functions on Lists

- **Example 1.**

length of a list:

```
length (l :: list N)
  :: N
= case l of
  {(nil) -> zero;
   (cons n l') -> S (length l')};
```

- **Example 2.**

Sum of elements of a list:

```
sum (l :: list N)
  :: N
= case l of
  {(nil) -> zero;
   (cons n l') -> n + (sum l')};
```

(g) 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
 - N_0, N_1, 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

$$\widehat{N}_0 : U$$
$$T(\widehat{N}_0) = N_0 : \text{Set}$$
$$\widehat{N}_1 : U$$
$$T(\widehat{N}_1) = N_1 : \text{Set}$$
$$\widehat{\text{Bool}} : U$$
$$T(\widehat{\text{Bool}}) = \text{Bool} : \text{Set}$$

Introduction/Equality Rules for the Universe (Cont.)

$$\frac{a : U \quad b : U}{a \hat{+} b : U}$$

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

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

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

$$\frac{a : U \quad b : \mathbb{T}(a) \rightarrow U}{\hat{\Pi}(a, b) : U}$$

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

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.
 - Example: Define

$$\begin{aligned}\widehat{\text{atom}} & : \text{Bool} \rightarrow U , \\ \widehat{\text{atom}} & := \text{then_else_if } U \widehat{N}_1 \widehat{N}_0 ,\end{aligned}$$

$$\begin{aligned}\text{atom} & : \text{Bool} \rightarrow \text{Set} , \\ \text{atom} & : \lambda x. T (\widehat{\text{atom}} x) ,\end{aligned}$$

Then

$$* \text{ atom true} = N_1,$$

$$* \text{ atom false} = N_0.$$

Universes in Agda

- U and 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.

Universes in Agda (Cont.)

mutual

```
U :: Set
= data Nhat
  | Nzerohat
  | Nonehat
  | Boolhat
  | Sigmahat (a :: U) (b :: T a -> U)
  | Pihat    (a :: U) (b :: T a -> U)
```

```
T(u :: U)
:: Set
= case u of
  { (Nhat)      -> N;
    (Nzerohat) -> Nzero;
    (Nonehat)   -> None;
    (Boolhat)   -> Bool;
    (Sigmahat a b)
      -> Sigma (T a) (\(x :: T a) -> T (b x));
    (Pihat a b)
      -> (x :: T a) -> T (b x); }
```

(h) The W -set.

Omitted.

(i) Inductive Data Sets.

- The construct “data” in Agda is much more powerful than what is covered by type theoretic rules.
- In general we can define now sets having arbitrarily many constructors with arbitrarily many arguments of arbitrary types.

```
A :: Set
  = data C1 (a11:: A11) ... (a1n1:: A1n1)
      | C2 (a21:: A21) ... (a2n2:: A2n2)
      ...
      | Cm (am1:: Am1) ... (amnm:: Amnm)
```

Meaning of “data”

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

$$\begin{aligned} C_i @ A &:: (a_{i1} :: A_{i1}) \\ &\rightarrow \dots \\ &\rightarrow (a_{in_i} :: A_{in_i}) \\ &\rightarrow A \end{aligned}$$

where a constructor always constructs new elements.

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

Strictly Positive Inductive Data Types

- In the types A_{ij} we can make use of A .
 - However, it is difficult to understand A , if we have **negative** occurrences of A .
 - Example:
 $A :: \text{Set}$
 $= \text{data } C (f :: A \rightarrow A)$
 - What is the least set A having a constructor
 $C @ A :: (f :: A \rightarrow A)$
 $\rightarrow A \quad ?$

Strictly Positive Inductive Data Types (Cont.)

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

Strictly Positive Inductive 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.

Strictly Positive Inductive Data Types (Cont.)

- In general:

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

is a strictly positive inductive 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 inductive data type, then A is acceptable.
 - The definitions of finite sets, $\Sigma A B$, $A + B$ and \mathbb{N} were strictly positive inductive 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 Inductive 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.

Strictly Positive Inductive Data Types, Extensions

- We can even allow $A_{ij} = B_1 \rightarrow A$ or even $A_{ij} = B_1 \rightarrow \dots \rightarrow B_1 \rightarrow A$, 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 $f :: N \rightarrow O$ and construct $\text{lim@}_\infty f$ out of it, adding this new element to O doesn’t destroy the reason for adding it to O .
- So again O can be “constructed”.

Elimination Rules for data

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

Examples

- For instance
 - in the Bintree example, when defining

$$f :: \text{Bintree} \rightarrow A$$

by case-distinction, then the definition of

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

can make use of $f \text{ left}$ and $f \text{ right}$.

- In the example of \mathbb{O} , when defining

$$g :: \mathbb{O} \rightarrow A$$

by case-distinction, then the definition of

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

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