B3.1

(1) Algebraic data types.
(2) Unions.
(3) Lists.
(4) The set of natural numbers.
(5) The set of integers.
(6) The set of rational numbers.
(7) The set of real numbers.
(8) The set of complex numbers.
(9) The set of booleans.

B3.2

The elimination rule:
\[ \text{cond} : \text{Bool} \rightarrow \text{bool} : \text{bool} \]

The introduction rules:
\[ \text{boo}l : \text{bool} \]
\[ \text{set} : \text{set} \]

B3.3

The equality rules:
\[ \text{equal} : \text{bool} = \text{equal} : \text{bool} \]
\[ \text{not} : \text{bool} \rightarrow \text{bool} \]

B3.4

Remarks

In the above:
- \( \text{tt} \) stands for true,
- \( \text{ff} \) stands for false.
- \( \text{case} \) stands for "case".
- \( \text{cond} \) stands for "condition".

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

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

In the above:
- \( \text{tt} \) stands for true,
- \( \text{ff} \) stands for false.
- \( \text{case} \) stands for "case".
- \( \text{cond} \) stands for "condition".

B3.5

The Set of Booleans

Formation Rule

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

Introduction Rules

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

Elimination Rule

\[ \text{cond} : \text{Bool} \rightarrow \text{bool} : \text{bool} \]

Equality Rules

\[ \text{equal} : \text{bool} = \text{equal} : \text{bool} \]
\[ \text{not} : \text{bool} \rightarrow \text{bool} \]

The Set of Booleans (cont.)
Noticethatwethengetfor
$C : \text{Bool} \rightarrow \text{Set}$, $ic : C \texttt{tt}$; $ec : C \texttt{ff}$:

\[
\begin{align*}
(f) & \rightarrow \text{Case Bool } \{ \
\texttt{tt} & = \text{Case Bool } \{ \
\texttt{tt} & = C \texttt{tt}, \\
\texttt{ff} & = C \texttt{ff} \} \\
\texttt{ff} & = \text{Case Bool } \{ \
\texttt{tt} & = C \texttt{ff}, \\
\texttt{ff} & = C \texttt{ff} \} .
\end{align*}
\]

Soweobtainfunctionsfrom
$\text{Bool}$inothersets
withouthavingtowrite
$(b : \text{Bool})$.

That'swhywewanttoeliminatefromasthe
lastone.

Inthesesewhichweget
fortheconstructor:

\[
\begin{align*}
\text{AND} & \leftarrow \text{Bool} \Rightarrow \text{Bool} \Rightarrow \text{Bool} \\
\{ + \} & = \text{Case Bool } \{ \
\texttt{tt} & = \texttt{tt} + x, \\
\texttt{ff} & = \texttt{ff} + x \} .
\end{align*}
\]

Thisissimilartothe
definitionofforinstance$(+)$in
Haskell:

\[
\begin{align*}
(+) & : \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
(+) & = \text{case } \text{int} \{ \
\text{tt} & = \text{tt} + \text{tt}, \\
\text{ff} & = \text{ff} + \text{ff} \} .
\end{align*}
\]

Shorterthanwriting$\texttt{tt} + \texttt{tt}$.

Inthefollowingwewrite$\text{Bool}$if
$\text{isatype}$in
boldfacered
andifitisaterm,in
italicblue.

\[
\text{AND}(\texttt{tt}, \texttt{ff}) = \texttt{tt}.
\]

\[
\text{AND}(\texttt{tt}, \text{tt}) = \texttt{tt}.
\]

\[
\text{AND}(\texttt{ff}, \text{tt}) = \texttt{tt}.
\]

\[
\text{AND}(\texttt{ff}, \text{ff}) = \texttt{ff}.
\]
Example (Cont.)

Derivation of $\land$:

$\text{Bool} \land \text{Bool} = \text{Bool}$

We derive $b : \text{Bool} ; c : \text{Bool} \vdash b \land c : \text{Bool}$ using part of the derivation above:

\[ \text{Example (Cont.)} \]

Similarly follows

\[ \text{Example (Cont.)} \]
We can extend elimination and equality rules, having as result Type Elimination Rule into Type

<table>
<thead>
<tr>
<th>CaseBool</th>
<th>Bool → Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaseBool</td>
<td>C tt e.C</td>
</tr>
<tr>
<td>CaseBool</td>
<td>C ff e.C</td>
</tr>
</tbody>
</table>

Equality Rules into Type

<table>
<thead>
<tr>
<th>Case</th>
<th>Bool → Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case</td>
<td>C tt e.C</td>
</tr>
<tr>
<td>Case</td>
<td>C ff e.C</td>
</tr>
</tbody>
</table>

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

\[ b: \text{Bool}, c: \text{Bool} \Rightarrow \lambda(x': \text{Bool}) : \text{Bool}, c: \text{Bool} \Rightarrow \text{Bool} \]
We introduce Bool by simply listing its constructors (similarly to Haskell):

data Bool = tt
          | ff

Critical Systems, CS 411, Lent term 2003, Sec. B3
B3-16

(C) Anton Setzer 2003 (except for pictures)

Internally, it will always be represented as tt | ff. Similarly for ll.

"Bool in Agda (cont.)"

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

\[ \text{Bool} :: \text{Set} = \text{dat tt | ff} \]
\[ \text{tt} :: \text{Bool} = \text{tt} @ \text{Bool} \]
\[ \text{ff} :: \text{Bool} = \text{ff} @ \text{Bool} \]

"Bool in Agda (cont.)"

So tt, and ff have to be defined separately.

"Bool in Agda (cont.)"

With this syntax, each constructor can occur at most once in a data type.

\[ \text{data True} = \text{ll} \]
\[ \text{ll} :: \text{Bool} \]
\[ \text{ll} :: \text{Bool} \]

"Bool in Agda (cont.)"

This introduces as well constructors

\[ \text{data Bool} = \text{ll | ff} \]
\[ \text{Set} :: \text{Bool} \]

"Bool in Agda (cont.)"

We introduce Bool by simply listing its constructors (similarly to Haskell).
Case Distinction

Let's assume we want to define:

\[ f : \text{Bool} \rightarrow \text{Bool} \]

such that:

\[ f \text{ tt}= f \]

This goal expands to:

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

\[ f (\text{tt}) \]

\[ f (\text{ff}) \]

\[ g \]

The value of \( x \) in the first goal can be tested as follows:

Alternatively, check: the cursor being in that goal, the context

Case Distinction (cont.)
Now we can solve the new goals by inserting \( \{ tt \} \) into the first one and \( \{ ! \} \) into the second one.

We obtain a function:

\[
\begin{align*}
\text{test} & : \text{Set} \\
\{ i \} & = \text{test} \quad \text{if } i \in \text{a dummy goal} \\
\end{align*}
\]

So we have another \( \text{tt} \) or \( \text{!} \) goal.

Testing the Defined Function

Testing the Defined Function (cont.)

(b) The Finite Sets

\( \text{Fin} \)

\[ \text{Introduction Rules} \]

\[ \text{Formation Rule} \]

\( \text{Elimination Rule} \)

\( \text{Test} \) is the negation of \( x \).
This can be done for all sets defined later as well.

Similarly as for Bool we can write down elimination rules, where...

Cases: $s_0 = s_1, \ldots, s_n = 1$.

Similarly as for Bool we can write down elimination rules, where...

Cases: $s_0 = s_1, \ldots, s_n = 1$.

Similarly as for Bool we can write down elimination rules, where...

Cases: $s_0 = s_1, \ldots, s_n = 1$.

Similarly as for Bool we can write down elimination rules, where...

Cases: $s_0 = s_1, \ldots, s_n = 1$.
Rules for True

- Formation Rule True: Set true

- Introduction Rules True: true : True

- Elimination Rule C : True ! Set

- Equality Rule C : True ! Set
t

Case True is the special case Fin for n = 0.

Rules for False

- Formation Rule False: Set false

- Introduction Rules False: false

- Elimination Rule C : False ! Set
t

Case False is the special case Fin for n = 0.

Rules for True (cont.)

- There is no Equality Rule

- There is no Introduction Rule

- There is no Elimination Rule

Critical Systems, CS411, Lent term 2003, Sec.B3
False has no computational meaning, since there is no element it can be applied to.  

\{ \} \cdot \text{False in Agda (cont.)} 

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

\begin{cases} 
\text{red} & \text{False} \\
\text{green} & \text{False} \\
\text{blue} & \text{False} 
\end{cases} \cdot \text{Finite sets in Agda} 

\text{If we want to solve} \quad \text{if we had full recursion, we could define } f : \text{False} \cdot \text{True of course only if we are working in a terminating type theory.} 

\text{Otherwise, one obtains for instance elements of False,} 

\text{there is no element of False.} 

\text{But that } f \text{ is well defined, i.e. } f : \text{False} \cdot \text{False in Agda (cont.)} 

\text{Agda} we can define the empty set as a „data-set with no constructors:} 

\text{False in Agda}
Example for the Use of False

Assume the type of trees:

```
dataTree = pine :: oak
```

Below we will show how to introduce a function `IsOak :: Tree -> Bool` s.t.

- `IsOak pine = False`
- `IsOak oak = True`

If we want to define a function from trees which are oak trees into another set, we can do so by requiring an additional argument `IsOak`:

```
f :: Tree -> f (IfOak :: Tree -> Bool) p :: (p :: IsOak) :: (t :: Tree) f
```

Similarly we can introduce a stack, together with a predicate `NotEmpty :: Stack -> Bool` s.t.

```
NotEmpty :: Stack -> Bool
```

Again we don't have to provide a result in case `s` is empty.

```
NotEmpty :: Stack -> Bool
```

Note that we don't have to invent a result of `f` if `s` is an empty tree.

```
NotEmpty :: Stack -> Bool
```

In order to use `f` we have to know that `l` is an oak tree.

```
NotEmpty :: Stack -> Bool
```

Example for the Use of False (Cont.)
True and False. •

There are formulae in type theory for which neither of these two holds.

– False is type-theoretically true, if it is provable: i.e.

. True is the type type-theoretically true.
. False is a proof of (true).
. True is inhabited.
• False.

– True

This means that we can neither prove them, nor derive from a proof a

non type-theoretically false

Atomic Formulae (Cont.)

There are formulae in type theory, which are neither type-theoretically true

nor type-theoretically false

Atomic Formulae (Cont.)

There are formulae in type theory, which are neither type-theoretically true

nor type-theoretically false

Atomic Formulae (Cont.)

{ {i : i} ← (true) case x of (true) =
  case x ::
  (true :: x) = 0

Case distinction will require to solve the case True:

data False =

The definition of True in Agda is straightforward:

True in Agda
We can map truth values to their corresponding formula:

\[
\begin{align*}
\text{atom} & \quad \text{if} \quad \text{true} \\
& \quad \text{if} \quad \text{false}
\end{align*}
\]

Using \( \text{atom} \) we can now define decidable predicates on sets.

Decidable Predicates

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

atom (cont.)
Decidable Predicates (Cont.)

Let now \( g : A \to \text{Set} \), \( g(a) = \text{atom}(f(a)) \).

- 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 \( g \) means:
- \( \forall a \in A. \exists b : B. g(a) = b \) or
- \( \forall a \in A. \forall b \in B. g(a) \neq b \), i.e. for all \( a \) there is at most one inhabitant.

Let now \( b : B. \forall a \in A. g(a) \neq b \) and \( A' \) also has the same safety assumption as \( A \).

The Traffic Light Example

Assume a road crossing, controlled by traffic lights:

\[ \begin{array}{c}
A \\
| \\
\hline
A' \\
\downarrow \\
B \\
\uparrow \\
B' \\
\end{array} \]

The Set of Physical States

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

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

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

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.

- \( \forall a \in A. \forall b \in B. g(a) \neq b \), i.e. for all \( a \) there is at most one inhabitant.
- \( \forall a \in A. \forall b \in B. g(a) \neq b \), i.e. for all \( a \) there is at most one inhabitant.

A complete set of control states consists of:

- \( A \) and \( A' \) always coincide, similarly \( B \) and \( B' \).
- Assume from each direction \( A \) and \( A' \), \( B \) and \( B' \). There is one traffic light.
- \( \forall a \in A. \exists b : B. g(a) = b \) and \( A' \) also has the same safety assumption as \( A \).
Wetherefore defines

\[
\text{data ControlState} = \text{Allred} \cup \text{Agreen} \cup \text{Bgreen}.
\]

\[
\text{Mapping Control States to Physical States}
\]

We define the state of signals \( A \) and \( B \) depending on a control state:

\[
\text{toSigA} : \text{ControlState} \to \text{Colour}
\]

\[
\begin{cases}
(A) \text{red} & \text{if} \ \text{Allred} \\
(A) \text{green} & \text{if} \ \text{Agreen} \\
(B) \text{red} & \text{if} \ \text{Bgreen}
\end{cases}
\]

\[
\text{toSigB} : \text{ControlState} \to \text{Colour}
\]

\[
\begin{cases}
(A) \text{red} & \text{if} \ \text{Allred} \\
(A) \text{red} & \text{if} \ \text{Agreen} \\
(B) \text{green} & \text{if} \ \text{Bgreen}
\end{cases}
\]

\[
\text{physState} : \text{ControlState} \to \text{PhysState}
\]

\[
\begin{cases}
\text{physState}(\text{Allred}) & = \text{red} \\
\text{physState}(\text{Agreen}) & = \text{red} \\
\text{physState}(\text{Bgreen}) & = \text{green}
\end{cases}
\]

\[
\text{Mapping Control States to Physical States}
\]

\[
\text{PhysState} = \text{PhysState}_{\text{red}} \cup \text{PhysState}_{\text{green}}.
\]

\[
\text{data ControlState} = \text{Allred} \cup \text{Agreen} \cup \text{Bgreen}.
\]

\[
\text{The Set of Control States (Cont.)}
\]

\[
\text{data PhysState} = \text{PhysState}_{\text{red}} \cup \text{PhysState}_{\text{green}}.
\]

\[
\text{The Set of Control States (Cont.)}
\]

\[
\text{physState} = \text{physState}_{\text{red}} \cup \text{physState}_{\text{green}}.
\]

\[
\text{physState} = \text{physState}_{\text{red}} \cup \text{physState}_{\text{green}}.
\]
### Safety of the System

#### Safety Predicate (Cont.)

Now we define:

\[
\text{Cor}(s) \::= \text{CorAux}(s) \cdot \quad \text{sigA} \cdot \quad \text{sigB}
\]

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

\[
\text{CorAux}(s) \quad \text{::} \quad \text{Cor}(\text{phys-state} \cdot s) \quad \text{::} \quad \text{cor-proof}(s)
\]

Now we show that all control states are safe.

If one makes a mistake which results in an unsafe situation: •

#### Safety of the System (Cont.)

If one makes a mistake which results in a physical state: •

Physical state: •

This works only because each control state corresponds to a correct state. •

Similarly for the other two elements. •

reduce to True. •

The first element true was an element of Cor(phys-state Alred), which
The Disjoint Union of Sets (Cont.)

Introduction Rules

\[ A : \text{Set} \]
\[ B : \text{Set} \]
\[ a : A \]
\[ \text{inl}(A B a) : A + B \]
\[ b : B \]
\[ \text{inr}(A B b) : A + B \]

Elimination Rule

\[ A : \text{Set} \]
\[ B : \text{Set} \]
\[ C : (A + B) ! \text{Set} \]
\[ \text{sl} : (a : A) ! C (\text{inl}(A B a)) \]
\[ \text{sr} : (b : B) ! C (\text{inr}(A B b)) \]
\[ d : A + B ! C \]

A more compact notation is:

\[ \text{disj-union} = \begin{array}{c}
\text{data} \ \text{inl} \ (a : A) : (A + B) \ \\
\text{data} \ \text{inr} \ (b : B) : (A + B) \ \\
\end{array} \]

Disjoin Union in Agda

```
(+): (A :: Set) (B :: Set) :: Set
  | (\_ : \_ :: \_ + \_ :: \_ :: \_)
  \text{inl}(a :: A) :: (A + B)
  \text{inr}(b :: B) :: (A + B)
```

Disjoint Union using the Logical Framework

A more compact notation is:

\[ \text{disj-union} = \begin{array}{c}
d : \text{Set} \to (A + B) \\
\end{array} \]
Thenotation (+) means, that (+) can be used in x.

Now we have, if A;B :: Set:

\[
\begin{align*}
\text{inl} &: A + B \rightarrow A \land (A + B) \\
\text{inr} &: A + B \rightarrow B \land (A + B)
\end{align*}
\]

If one can type into the goal c and choose menu „agda-case“.

\[
\begin{align*}
\{ i \} &= \text{(void)} \\
\{ i, j \} &= \text{(g + v :: c) f}
\end{align*}
\]

So if want to define for A, B :: Set for instance.

Elimination is again represented by case distinction. 

Use of Concrete Disjoint Sets

It is usually more convenient to define concrete disjoint unions directly with

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

Now one can define for instance

```
isFlower (p :: Plant) :: Bool = case p of
  f (tree t) \rightarrow !f \\
  (\text{flower} f) \rightarrow !f \\
  g 
```

and insert into the first goal e:\n
\[
\{ i \} \leftarrow \text{(true)} \\
\{ i, j \} \leftarrow \text{(true)}
\]

which would be of the form

\[
\text{case of}
\begin{cases}
\text{Bool} &\rightarrow f \\
(G + V :: c) &\rightarrow !f
\end{cases}
\]

We obtain

```
\text{data Plant = tree (t :: Tree) | flower (f :: Flower)}
```

```
isFlower (p :: Plant) :: Bool = case p of
  f (tree t) \rightarrow !f \\
  (\text{flower} f) \rightarrow !f \\
  g 
```

```
\text{case of}
\begin{cases}
\text{Bool} &\rightarrow f \\
(G + V :: c) &\rightarrow !f
\end{cases}
```

```
\text{case of}
\begin{cases}
\text{Bool} &\rightarrow f \\
(G + V :: c) &\rightarrow !f
\end{cases}
```

```
\text{case of}
\begin{cases}
\text{Bool} &\rightarrow f \\
(G + V :: c) &\rightarrow !f
\end{cases}
```

```
\text{case of}
\begin{cases}
\text{Bool} &\rightarrow f \\
(G + V :: c) &\rightarrow !f
\end{cases}
```

```
\text{case of}
\begin{cases}
\text{Bool} &\rightarrow f \\
(G + V :: c) &\rightarrow !f
\end{cases}
```

```
\text{data Plant = tree (t :: Tree) | flower (f :: Flower)}
```

```
isFlower (p :: Plant) :: Bool = case p of
  f (tree t) \rightarrow !f \\
  (\text{flower} f) \rightarrow !f \\
  g 
```
The $\exists$-Set (Cont.)

\[
\begin{align*}
\{ & p : \exists x : p \} \\
\{ (q \land p \land \exists y : q) : \} \\
\{ & (v : \exists x : (v : \exists y : q) \land (v : \exists y : q)) : \} \\
\{ & (\exists x : (v : \exists y : q) \land (v : \exists y : q)) : \}
\end{align*}
\]

The more compact notation is:

\[ p : \exists x : p \land \exists y : q \land \exists z : (v : \exists y : q) \land (v : \exists y : q) \]

The $\exists$-Set using the Logical Framework (Cont.)

\[
\begin{align*}
\{ & p : \exists x : p \land \exists y : q \land \exists z : (v : \exists y : q) \land (v : \exists y : q) \\
\{ & (q \land p \land \exists y : q) : \} \\
\{ & (v : \exists x : (v : \exists y : q) \land (v : \exists y : q)) : \} \\
\{ & (\exists x : (v : \exists y : q) \land (v : \exists y : q)) : \}
\end{align*}
\]
The dependent product and the dependent product are very similar.

1. Both have similar introduction rules (for the dependent product, the constructors have additional arguments $A, B$ necessary for bureaucratic reasons only).
2. One can define the projections $\pi_0, \pi_1$ using Sigma-split:

\[
\Sigma_{x:A} \pi_0 \cdot (x) = 1^A \quad \Sigma_{x:A} \pi_1 \cdot (x) = 0^B
\]

3. Because of the lack of $\prod$-rule, $\Sigma$ works usually better than the dependent product implemented in Agda.

However, the dependent product has the $\prod$-rule, which is however not implemented in Agda.

I personally don’t use the dependent product of Agda much.

The dependent product in Agda can be defined as a data-set with constructor $p$:

\[
\text{data \, Plant} \rightarrow \text{Plant-Group} \rightarrow \text{Plants-in-Group} =
\]

\[
(\text{Plant} \rightarrow \text{Plant-Group} = (\text{Plant-Group} \rightarrow \text{Plants-in-Group})
\]

\[
(\text{Plant} \rightarrow \text{Plant-Group} \rightarrow \text{Plants-in-Group})
\]

\[
\text{Plant-Group} \rightarrow \text{Plants-in-Group}
\]

Example: Assume we have defined a set $\text{Plant-Group}$ for groups of plants (e.g. "tree", "flower") and a set $\text{Plants-in-Group}$ for groups of plants (e.g. "trees", "flowers").

\[
\text{Plants-in-Group} = \text{Plant} \rightarrow \text{Plant-Group}
\]

Again one usually defines concrete $\Sigma$-sets more directly.
The -Set in Agda (Cont.)

Notsurprisingly, for elimination we use casedistinction, e.g.:

```
f(p :: Plant) :: Plant group = case p of
  f(plant gpg) ! g; g
```

Critical Systems, CS 411, Lent Term 2003, Sec. B3

B3-71

Conjunction

\[\begin{align*}
&\frac{A \lor B}{A} \\
&\frac{A \lor B}{B} \\
\end{align*}\]

This is what is expressed by the ordinary introduction rule for \(\lor\).

This means that we can derive \(A \lor B\) from \(A\) and \(B\).

\[\begin{align*}
&\frac{A \lor B : (b,d)}{B : b} \\
&\frac{A \lor B : (b,d)}{A : d} \\
\end{align*}\]

With this identification, the introduction rule for \(\lor\) allows to form a proof of \(A \lor B\).

Formulae in Dependent Type Theory

We have seen how to represent atomic decidable formulae.

Now treatment of complex formulae constructed using logical connectives.

Critical Systems, CS 411, Lent Term 2003, Sec. B3

B3-72

Conjunction (Cont.)

With this identification, the introduction rule for \(\land\) allows to form a proof of \(A \land B\).

\[\begin{align*}
&\{ A \land B \in \text{A} \times \text{B} \} \\
&\text{Therefore the set of proofs of } A \land B \text{ is the set of pairs of elements of } A \text{ and } B, \\
&\text{it is therefore a pair } (b,d) \text{ consisting of a proof } b \text{ of } A \text{ and a proof } d \text{ of } B. \\
&\text{Therefore a proof of } A \land B \text{ consists of a proof } b \text{ of } A \text{ and a proof } d \text{ of } B. \\
&\text{Conjunction is true iff both } A \text{ is true and } B \text{ is true.} \\
\end{align*}\]

B3-73

The 2-set in Agda (Cont.)

\[\begin{align*}
&\{ \text{case } d \text{ of } \} \\
&\text{piece-branch} :: d; \text{piece} :: d \} f
\end{align*}\]

Not surprisingly, for elimination we use case distinction.

Critical Systems, CS 411, Lent Term 2003, Sec. B3

B3-74
The elimination rule for \( \land \) allows to project a proof of \( A \land B \) to a proof of \( A \) and a proof of \( B \):

\[
\frac{A \land B}{\frac{A}{d}} \quad \frac{A \land B}{\frac{B}{d}}
\]

This is what is expressed by the ordinary introduction rule for \( \land \):

\[
\frac{A}{A \land B} \quad \frac{B}{A \land B}
\]

Therefore, a proof of \( A \land B \) consists of a proof of \( A \) or a proof of \( B \).

\[
\frac{A \lor B}{A \lor B}
\]

Therefore the set of proofs of \( A \lor B \) is the disjoint union of the sets of proofs of \( A \) and \( B \).
**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 \( B \):

\[
\begin{align*}
g & : b \quad (g : b \quad (v : d)) \quad (\forall \quad d \quad v) \quad v \quad d
\end{align*}
\]

This means that, if we, from assumptions \( A \) \& \( B \) can prove \( C \), we can derive from \( A + B \) a formula \( C \), if we can derive \( C \) from \( A \) and from \( B \).

- This means that we can derive from a formula \( A \lor B \), if we can derive \( C \) from \( A \) and from \( B \).

This is what is expressed by the ordinary elimination rules for \( \lor \):

\[
\begin{align*}
\text{from } A \lor B, \text{ we get: } \quad & C
\end{align*}
\]

- This means that we can derive from \( A \lor B \) a formula \( C \), if we can derive

**Implication (Cont.)**

The can identify \( A \lor B \) with \( A \rightarrow B \).

We can identify \( A \lor B \) with \( A \rightarrow B \).

- Therefore the set of proofs of \( A \lor B \) is the function type \( A \rightarrow B \).

**Implication (Cont.)**

**Disjunction (Cont.)**

Then we can derive \( A \lor B \) without assuming \( A \lor B \):

\[
\begin{align*}
g & : b \quad (g : b \quad (v : d)) \quad (\forall \quad d \quad v) \quad v \quad d
\end{align*}
\]

This means that, if we, from assumptions \( A \) \& \( B \) can prove \( C \), we can derive from \( A + B \) a formula \( C \), if we can derive

Therefore the set of proofs of \( A \lor B \) is the function type \( A \rightarrow B \).

Therefore a proof of \( A \lor B \) is a function, which takes a proof of \( A \) and

Therefore if there is a proof of \( A \), there must be a proof of \( B \).

Therefore if \( A \lor B \) is true, \( B \) is true whenever \( A \) is true then \( B \) is true.

- Therefore if there is a proof of \( A \), then \( A \lor B \) is true.

- Below we see that \( C \) can be identified with \( A \rightarrow B \).

- The function type \( A \rightarrow B \) for logical implication is in order to distinguish it from

**Implication**

**Implication (Cont.)**

**Disjunction (Cont.)**
Implication (Cont.)

This is what is expressed by the ordinary introduction rule for:

\[ A \rightarrow B \]

The elimination rule for allows to apply a proof of \( A \rightarrow B \) and a proof \( B \rightarrow B' \) in order to obtain a proof of:

\[ \neg B \]

The elimination rule for allows to apply a proof of \( B \rightarrow \bot \) to a proof of \( \bot \rightarrow B \) to get a proof:

\[ \neg B \]

This means that we can derive from \( A \rightarrow B \) and \( A \rightarrow A \) that \( B \) holds.

\[ \neg A \]

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

\[ \neg A \rightarrow B \]

We can identify \( \forall x : A \rightarrow B \) with \( x : A \rightarrow B \), which takes an \( x : A \) and

\[ \forall x : A \rightarrow B \]

Therefore the set of proofs of \( \forall x : A \rightarrow B \) is the dependent function type

\[ \forall x : A \rightarrow B \]

Any \( \forall x : A \rightarrow B \) is true iff for all \( x : A \) there exists a proof of \( B \) (with that \( x \)).

We write the theorem \( \forall x : A \exists y : B \rightarrow \forall x : A \exists y : B \):

\[ \forall x : A \rightarrow \exists y : B \rightarrow \forall x : A \rightarrow \exists y : B \]

Since we have many types, we have to write when using quantifiers explicitly.

Universal Quantification

Therefore we can identify \( \forall x : A \rightarrow \bot \) with \( A \rightarrow \bot \).

Therefore we can identify \( \forall x : A \rightarrow \bot \) with \( A \rightarrow \bot \).

\[ \forall x : A \rightarrow \bot \]

Therefore \( \forall x : A \rightarrow \bot \) with \( (x : A) \rightarrow \bot \).

\[ (x : A) \rightarrow \bot \]

\[ \forall x : A \rightarrow \bot \]

\[ (x : A) \rightarrow \bot \]

\[ \forall x : A \rightarrow \bot \]

\[ (x : A) \rightarrow \bot \]
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 $x : A$.

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

$A : \forall x : A : B$ By $\forall$ which doesn't depend on $x : A$.

$B : \forall x : \forall y : A : B$ 

This is what is expressed by the ordinary introduction rule for $\forall$.

$A : \forall x : \forall y : A : B$ 

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 $\forall x : A : B$ can prove $B$. In more complex types, however, we have to carry out this derivation.

The elimination rule for the dependent function type allows to apply a proof of $A : B$ to an element of $A$. In order to obtain a proof of $\forall A : B$, we have to form a proof of $\forall A : B$. With this identification, the introduction rule for $\forall$ allows to form a proof.
Existential Quantification

$x : A \vdash B$ is true if there exists an $a : A$ such that $B[a/x]$ is true.

Therefore a proof of $\exists x : A \vdash B$ is a pair $(a, p)$ consisting of an element $a : A$ and a proof $p$ of $B[a/x]$.

This is what is expressed by the ordinary introduction rule for $\exists$.

The elimination rule for the dependent product allows to project a proof of $B$ to a proof of $A$.

Therefore the rule in natural deduction follows from the type theoretic

Existential Quantification (cont.)

not depending on $x : A$ or $y : B$.

Then we have $C : \forall x : A \forall y : B . C$, where $C : [d : \forall x : A \forall y : B. C \vdash \exists x : A . B$. Where does not depend on $x : A$ and $B$.

Assume $C$.

Then we have $C[a/x, y] : B[a/x, y]$ for any $y : B$.

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

The elimination rule for the dependent product allows to project a proof of $B$ to a proof of $A$.

Existential Quantification (cont.)

Critical Systems, CS 411, Lent term 2003, Sec. B3
Constructive 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
\[ \lambda f. \quad \text{for } x : A \text{ it follows } \exists y : B \text{ such that } C(x, y) \]
and
\[ \lambda g. \quad \text{for } x : A \text{ and } y : B \text{ such that } C(x, y) \text{ then we have } \exists z : C(x, y) \text{ for all } z \]
- Therefore we can decide the Turing-Halting problem, i.e., we cannot decide whether \( p = \text{inl}(a) \) or \( p = \text{inr}(b) \).
- Any function in type theory is recursive.
- Therefore we can decide, from a proof of a disjunction, which of the disjuncts holds.
- \( A \lor B \) is equivalent to \( \exists x : \text{if } p = \text{inl}(a) \text{ then } x = a \) \( \text{else } x = b \) \( \quad \text{in classical logic} \)

Critical Systems, CS 411, Lent term 2003, Sec. B3

B.3.96

(© Anton Setzer 2003 (except for pictures)
We show classically:

\[
\exists x : (B \land \forall y : (A \lor (y \land B)) \land (B \land \forall y : (A \lor (y \land B)))
\]

Now it follows classically:

\[
B : A \iff B, \text{ therefore a contradiction.}
\]

We show intuitionistically:

\[
\exists x : (B \land \forall y : (A \lor (y \land B)) \land (B \land \forall y : (A \lor (y \land B)))
\]

Proof (using classical logic) of

\[
\text{Constructive Logic (cont.)}
\]
The set \( \mathbb{N} \) is the type theoretic representation of the set \( \mathbb{N} := \{0, 1, 2, \ldots\} \).

\( \mathbb{N} \) can be generated by:

- \( 0 \in \mathbb{N} \)
- For any \( n \in \mathbb{N} \), \( n + 1 \in \mathbb{N} \)

Then the type theoretic rules are:

Let \( \ast \) be a type theoretic notation for the operation \( x + 1 \). Then we can define \( f : \mathbb{N} \to \mathbb{N} \) as follows:

\[
\begin{align*}
\text{If } n = 0, \text{ then the result is } a. \\
\text{Otherwise } n = S(n). \\
\text{Compute } n.
\end{align*}
\]

\( f \) expresses:

- \( f_0 = a \)
- \( f_{S(n)} = g_n(f_n) \)

Assume we have:

- \( g_n(f_n) \) can be computed, since we know how to compute \( f_n \).
- \( f_n \) reduces to \( g_n(f_n) \).
- We assume that we have determined already how to compute \( f_n \).

\( f \) reduces to \( g_n(f_n) \).

\( g_n(f_n) \) can be computed, since we know how to compute \( f_n \).

\( f \) is the type theoretic representation of the set \( \mathbb{N} := \{0, 1, 2, \ldots\} \).
The function \( f : \mathbb{N} \to \mathbb{N} \) with \( f(x) = 2x \) can be defined recursively by:

\[
\begin{align*}
\{ & f(0) = 0, \\
& f(Sn) = S(S(n)) \
\}
\end{align*}
\]

Therefore take in the definition above:

\[
\{ & a = 0, \\
& gn(x) = S(S(x)) \
\}
\]

We can generalize primitive recursion as follows:

First we can replace the range of \( f \) by an arbitrary set \( C \). Let us then allow for any set \( C \).

The conclusion of the elimination rule reads:

\[ f \in C \text{ and } g \in \{ f(0) = a, g(Sn) = fn(g(n)) \} \]

The equality rules read:

\[ g(0) = a, g(Sn) = fn(g(n)) \]

Example

The function with \( x \cdot 2 = (x)f \) can be defined primitive.
Rules for \( \mathbb{N} \) using the Logical Framework

The more compact notation is:

\[
\{ \text{\texttt{i}} :: \text{\texttt{Z}} \} \\
\{ \text{\texttt{i}} :: \text{\texttt{S}} \}
\]

Elimination Rules for \( \mathbb{N} \) in Agda (cont.)
Example of the Power of Termination Check

The following definition of the Fibonacci numbers can’t be defined directly using the rules of type theory, but it can be defined in Agda as follows and Agda-check-termination accepts it:

\[
\begin{align*}
\text{one} &:: \text{SZ} \\
\text{b}(n::\text{N})::\text{N} &= \begin{cases} 
\text{one} & \text{if } n = \text{Z} \\
\text{case } n \text{ of } f(\text{Z}) & ! \text{one}; \\
S(n) & ! \text{case } n \text{ of } f(\text{Z}) ! \text{one}; \\
S(n) & ! \text{case } n \text{ of } f(\text{Z}) ! \text{one}; \\
S(n) & ! \text{case } n \text{ of } f(\text{Z}) ! \text{one}; \\
\end{cases}
\end{align*}
\]

Example for Limitation of Termination Check

Assume we define the predecessor function:

\[
\begin{align*}
\text{pred} \ 0 &= u \ \text{if} \ I - u \\
\text{otherwise} &\ = \ (u)\text{pred}
\end{align*}
\]

The following definition of the Fibonacci numbers can’t be defined this way:

\[
\begin{align*}
\text{f}(n::\text{N})::\text{N} &= \begin{cases} 
\text{Z} & \text{if } n = \text{Z} \\
\text{f}(\text{pred}(n)) & \text{otherwise}
\end{cases}
\end{align*}
\]

If Agda-check-termination succeeds, the definition should be correct.

Example for Limitations of Termination Check (Cont.)

However, Agda-check-termination fails.

(Finally for all \( u \in \text{N} \), the value \( \langle Z \rangle \))

Terminates always:

\[
\begin{align*}
\{ \langle u \text{ pred} \rangle f \ &\rightarrow \langle u \ S \rangle \\
\langle Z \rangle \ &\rightarrow \langle Z \rangle \}
\end{align*}
\]

Theorem

\[
\begin{align*}
\begin{cases} 
\{ \langle u \text{ pred} \rangle f \ &\rightarrow \langle u \ S \rangle \\
\langle Z \rangle \ &\rightarrow \langle Z \rangle \}
\end{cases}
\end{align*}
\]
Limitations of the Termination Check (Cont.)

Because of the undecidability of the Turing Halting Problem, there is no extension of Agda-check-termination, which accepts exactly those recursively defined functions which terminate or not.

Example: Addition

\[
I + (\mu v . u) = (I + \mu v) + u \\
0 = 0 + u
\]

The definition expresses:

\[
\{ I + (\mu v . u) \} \leftarrow \{ (\mu v) \} \\
\begin{array}{l}
\text{case m of} \\
\quad \text{Z} \\
\quad \text{S n} \\
\end{array}
\]

\[
\text{N} \mapsto (\text{N} \mapsto \text{m . u}) (+)
\]

Definition of + in Agda:

Example: Multiplication

\[
(+)(n;m) :: N \mapsto N = \text{case } m \text{ of} \\
\quad \text{Z} \\
\quad \text{S m} 0 \\
\]

\[
\text{N} \mapsto (\text{N} \mapsto \text{m . u}) (+)
\]

The definition expresses:

\[
\text{N} \mapsto (\text{N} \mapsto \text{m . u}) (+)
\]

Definition: Multiplication

\[
\text{N} \mapsto (\text{N} \mapsto \text{m . u}) (+)
\]

\[
\text{N} \mapsto (\text{N} \mapsto \text{m . u}) (+)
\]

Example: Addition

\[
\text{N} \mapsto (\text{N} \mapsto \text{m . u}) (+)
\]
Again is treated in $\mathbb{N}$.

Agda has a built-in that $+$ binds more than $\times$.

Equality on $\mathbb{N}$ (cont.)

Equality on $\mathbb{N}$

Equality on $\mathbb{N}$ (cont.)
This can now be shown using case distinction:

Symmetry of == (cont.)

\[
\begin{align*}
\text{Case } n = Z & \text{ is trivial.} \\
\text{Case } n = S n_0 & \text{ can be solved using real } n \text{ (which is defined before real } n\text{).}
\end{align*}
\]

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

Symmetry of == (Cont.)

This can now be shown using case distinction:

\[
\begin{align*}
\text{Symmetry of == (cont.)}
\end{align*}
\]

Type theoretically this means that we have to define a function \( \text{sym} \):

\[
\begin{align*}
\text{sym}(n; m) & = \\
(\text{n} = \text{m}) & : \\
(\text{m} = \text{n} : d) & : \\
(\text{N} : \text{w}, \text{w}) & : \\
\text{sym}
\end{align*}
\]

Symmetry of == is the formula:

\[
\begin{align*}
\text{Symmetry of ==}
\end{align*}
\]

\[
\begin{align*}
\text{Symmetry of == (cont.)}
\end{align*}
\]
The first goal can be solved by using true since it is of type True.

For the second goal, we know d is element of Z which is false.

The goal can be solved by using identity on d.

Similarly the third goal can be solved.

and have solved the second goal.

Therefore if we make case distinction on d we get

For the second goal we know d is element of Z which is false.

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

\begin{align*}
\text{data }\text{Nil} \equiv \text{Nil} & \quad \text{data }\text{Cons}(A;B) \equiv \text{Cons}(A;B) \\
\text{data }\text{cons}(a :: A) (b :: B) & \equiv \text{cons}(a :: A) (b :: B)
\end{align*}

Example: Tuples (or Vectors) of Length n

\begin{align*}
\text{data }\text{Nil} & \equiv \text{Nil} \quad \text{data }\text{Cons}(A;B) \equiv \text{Cons}(A;B) \\
\text{data }\text{cons}(a :: A) (b :: B) & \equiv \text{cons}(a :: A) (b :: B)
\end{align*}

Example: Tuples (or Vectors) of Length n

\begin{align*}
\text{data }\text{Nil} & \equiv \text{Nil} \\
\text{data }\text{Cons}(A;B) & \equiv \text{Cons}(A;B) \\
\text{data }\text{cons}(a :: A) (b :: B) & \equiv \text{cons}(a :: A) (b :: B)
\end{align*}

Example: Tuples (or Vectors) of Length n

\begin{align*}
\text{data }\text{Nil} & \equiv \text{Nil} \\
\text{data }\text{Cons}(A;B) & \equiv \text{Cons}(A;B) \\
\text{data }\text{cons}(a :: A) (b :: B) & \equiv \text{cons}(a :: A) (b :: B)
\end{align*}

Example: Tuples (or Vectors) of Length n

\begin{align*}
\text{data }\text{Nil} & \equiv \text{Nil} \\
\text{data }\text{Cons}(A;B) & \equiv \text{Cons}(A;B) \\
\text{data }\text{cons}(a :: A) (b :: B) & \equiv \text{cons}(a :: A) (b :: B)
\end{align*}
Remarks on Tuples of Length n

Therefore (with the obvious definition of $\text{Vec}$)

\[
\text{Vec}(A^n) = \text{cons}(A) (\text{cons}(A) (\ldots))
\]

The elements of $\text{Vec}(A^n)$ are $A^\times n$ times $\text{cons}(A)$.

Elements of $\text{Vec}(A^n)$

For elements $a_1, \ldots, a_n$ of $A$, we would write $(a_1, \ldots, a_n)$.


Example: Sum of Tuples of Length n

Define \( NVec(n) \) :: Set = VecN n

\( Nvec \) are tuples of natural numbers of length \( n \).

Critical Systems, CS 411, Lent term 2003, Sec. B3

Example: Componentwise Sum of Tuples of Length n

We define component-wise sum of tuples of length \( n \).

\[
\begin{align*}
\{ & 2, 3, 4 \} + \{ & 5, 6, 7 \} = \{ & 7, 9, 11 \}.
\end{align*}
\]

Using mathematical notation, this sum for instance as follows:

We define component-wise sum of tuples of length \( n \).

\[
\text{Sum}_{NVec(n)}(avec; bvec) :: Nvec n = \text{case } n \text{ of } \
\begin{aligned}
\text{n} (Z) & \Rightarrow \text{nil} ; \\
\text{n} (S n_0) & \Rightarrow \text{case } avec \text{ of } \
\begin{aligned}
\text{nil} & \Rightarrow \text{nil} ; \\
\text{cons} a avec & \Rightarrow \text{case } bvec \text{ of } \
\begin{aligned}
\text{nil} & \Rightarrow \text{nil} ; \\
\text{cons} b bvec & \Rightarrow \text{cons} (a + b) \left( \text{Sum}_{NVec(n)}(avec_0; 0); bvec_0; 0) \right).
\end{aligned}
\end{aligned}
\end{aligned}
\]

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

We define the set of lists of elements of type \( A \) in Agda.

\[
\text{list}(A) :: \text{Set} = \text{data nil} \mid \text{cons} a :: \text{list}(A) \text{.}
\]

Lists

\{ nil, cons \}

We have two constructors:

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

We have two constructors:

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

We define lists as:

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

We define lists as:

\[
\begin{align*}
\text{list}(A) :: \text{Set} & = \text{data nil} \mid \text{cons} a :: \text{list}(A) \text{.}
\end{align*}
\]
Elimination Rule for Lists

Elimination rule uses list recursion:

Assume \( A \) : Set, \( C \bull \) : Set depending on \( l \bull \) : list \( A \). Then we can define \( f \bull (l) : C \)

\[
\begin{align*}
\text{case} \ (l) \Rightarrow \\
\text{nil} & \Rightarrow \ C \\
\text{cons} (\text{set} \ (l), \ \text{set} \ (l)) & \Rightarrow \ C
\end{align*}
\]

and if we define \( \text{cons} :: \text{list} (A) \bull \text{list} (A) \bull \text{list} (A) \bull \text{list} (A) \bull \text{list} (A) \bull \text{list} (A) \bull \text{list} (A) \bullet \bullet \bullet \), then:

\( \text{nil} \bull \text{list} (A) \bull \text{list} (A) \bullet \bullet \bullet \)

E.g., if \( a, b, c \bull \) are elements of \( A \),

\( \text{cons} (\text{set} \ (l), \ \text{set} \ (l)) \)

is the result of appending the list \( l \) at the end of list \( l \).

Define \( \text{append} : (A \bull \text{list} (A)) \bull \text{list} (A) \).

Example: Length of a List

\[
\begin{align*}
\text{length} \ (l) & \Rightarrow \ \text{nat} \\
\text{case} \ (l) \Rightarrow \\
\text{nil} & \Rightarrow \ Z \\
\text{cons} (\text{set} \ (l), \ \text{set} \ (l)) & \Rightarrow \ \text{S(length (l))}
\end{align*}
\]

Example: Sum of the Elements of a List

\[
\begin{align*}
\text{sumlist} \ (l) & \Rightarrow \ \text{nat} \\
\text{case} \ (l) \Rightarrow \\
\text{nil} & \Rightarrow \ Z \\
\text{cons} (\text{set} \ (l), \ \text{set} \ (l)) & \Rightarrow \ \text{S(sumlist (l))}
\end{align*}
\]

Interesting Exercise

Define \( \text{append} : (A \bull \text{list} (A)) \bull \text{list} (A) \bull \text{list} (A) \bull \text{list} (A) \bull \text{list} (A) \bull \text{list} (A) \bull \text{list} (A) \bullet \bullet \bullet \).

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

\( \text{nil} := \text{nil} @ (\text{list} (A)) \),

then:

\( \text{append} A @ (\text{cons} a (\text{cons} b \text{nil})) (\text{cons} c (\text{cons} d \text{nil})) \)

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

\[ U : \text{Set} \]

We consider in the following a universe closed under

\[ \vdash \text{Set} \]

\[ \vdash \text{Sel} \]

\[ \vdash \text{Pair}, \text{Bool} \]

the dependent function type.

\[ \vdash (x : A, y : B) \rightarrow (y, x) : (B \times A) \]

\[ \vdash (p : \text{Pair}, f : \text{Pair}) \rightarrow f(p) : \text{Pair} \]

We have

\[ \vdash \exists x : X. \Psi(x) \rightarrow \exists x : X. \Phi(x) \]

\[ \vdash \forall x : X. \Phi(x) \rightarrow \forall x : X. \Psi(x) \]

\[ \vdash \forall x : X. \exists y : Y. \Phi(x, y) \rightarrow \forall y : Y. \exists x : X. \Phi(x, y) \]

\[ \vdash \forall x : X. \Phi(x) \rightarrow \forall x : X. \Psi(x) \]

We have

\[ \vdash \exists x : X. \Phi(x) \rightarrow \exists x : X. \Psi(x) \]

\[ \vdash \forall x : X. \Phi(x) \rightarrow \forall x : X. \Psi(x) \]

A universe $U$ is a set, the elements of which are codes for sets. So we have

\[ U : \text{Set} \]

\[ \vdash \exists x : X. \Phi(x) \rightarrow \exists x : X. \Psi(x) \]

\[ \vdash \forall x : X. \Phi(x) \rightarrow \forall x : X. \Psi(x) \]
There exist as well elimination rules and corresponding equality rules for the universe.

Example: Define

```
atom : Bool ! U
atom = Case Bool (x : Bool) { atom x }
```

Then

```
\[ \forall x : (x : Bool) \rightarrow (atom x) \] 
```

from which

```
atom tt = Fin 1 , atom = Fin 0 .
```

### Application of the Universe

**Applications of the Universe**

- Ordinary elimination rules don't allow to eliminate into Set and use \( T \) to obtain the required function.

- However, one can verify that all sets needed are elements of a "universe".

- There are codes in the universe representing them.

- \( \sqcup \) and \( \sqcap \) need to be defined simultaneously.

- Usually a type checks definitions in sequence, so no reference to later definitions is determined by indentation.

- Everything in the scope of \( U \) is type checked simultaneously.

- Special constructor \( U \) makes elimination possible.

- Scope determined by indentation.

- Scoped determined by indentation.
Universes in Agda (Cont.)

mutual

\begin{align*}
U :: \text{Set} = & \text{data} NHat \\
& \text{FinzeroHat} \\
& \text{FinoneHat} \\
& \text{BoolHat} \\
& \SigmaHat(a :: U)(b :: T(a) ! U) \\
& \PiHat(a :: U)(b :: T(a) ! U)
\end{align*}

\begin{align*}
T \text{ in the following is to be intended the same as } U
\end{align*}

(i) Algebraic Data Types.

The construct \texttt{data} in Agda is much more powerful than what is covered by type theoretic rules.

\begin{align*}
(\forall a :: \text{Set}) & \vdash (a ! U :: x) \rightsquigarrow (q ! n :: x) \\
\text{Finzero} & \vdash \text{Finzero} \\
\text{Finone} & \vdash \text{Finone} \\
\text{Bool} & \vdash \text{Bool} \\
(\forall a :: U)(b :: T(a) ! U) & \text{data} C \\
& \text{case } C :: \text{Set} \\
& \text{case } n :: \bigcup (\forall a :: U)(b :: T(a) ! U)
\end{align*}
Strictly Positive Algebraic Data Types

In the types $A_{ij}$ we can make use of $A_j$. However, it is difficult to understand $A_j$ if we have negative occurrences of $A_j$. Example: $A::Set = data C(f::A->A)$

What is the least set $A$ having a constructor $C@A::(f::A->A)$? If we have constructed some part of $A$ already, then $f$ might no longer be a function $A->A$. If we add $C@f$ to $A$, this might not be defined.

We should not make use of such definitions.

In fact, a “good” definition is the set of lists of natural numbers, defined as follows:

A “good” definition is the set of lists of natural numbers, defined as follows:

Because we can “construct” $N$ from the above, the above is an acceptable definition.

The definition of finite sets $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$ is $\mathbb{N}_0$.

The set $\mathbb{N}_0$ is defined as follows:

\[
\begin{align*}
\mathbb{N}_0 & = \{0\} \\
\mathbb{N}_0 & = \mathbb{N}_0 + 1
\end{align*}
\]

We have the following:

\[\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}\]

\[\mathbb{N}_0 = \mathbb{N}_0 + 1\]

The constructor consists of $\mathbb{N}_0$ itself, plus $\mathbb{N}_0 + 1$.

A “good” definition is the set of lists of natural numbers, defined as follows:

The set $\mathbb{N}_0$ is defined as follows:

\[
\begin{align*}
\mathbb{N}_0 & = \{0\} \\
\mathbb{N}_0 & = \mathbb{N}_0 + 1
\end{align*}
\]

We have the following:

\[\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}\]

\[\mathbb{N}_0 = \mathbb{N}_0 + 1\]

The constructor consists of $\mathbb{N}_0$ itself, plus $\mathbb{N}_0 + 1$.

A “good” definition is the set of lists of natural numbers, defined as follows:

The set $\mathbb{N}_0$ is defined as follows:

\[
\begin{align*}
\mathbb{N}_0 & = \{0\} \\
\mathbb{N}_0 & = \mathbb{N}_0 + 1
\end{align*}
\]

We have the following:

\[\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}\]

\[\mathbb{N}_0 = \mathbb{N}_0 + 1\]

The constructor consists of $\mathbb{N}_0$ itself, plus $\mathbb{N}_0 + 1$.

A “good” definition is the set of lists of natural numbers, defined as follows:
One further example: The set of binary trees can be defined as follows:

\[ \text{Bintree::Set} = \text{data leaf} \]

\[ \cup \text{branch(left::Bintree), right::Bintree)} \]

This is a strictly positive data type.

Example called "Kleene’s O":

\[ \text{Even::Set} = \text{data Z | } \text{succ(n::Odd)} \]

\[ \text{Odd::Set} = \text{data n::Even} \]

The last definition is unproblematic, since if we have \( f : n \gets 0 \) and

\[ f \text{ has definition } \]

\[ f(N+1) = f(N) \]

\[ \text{succ}(0) = 0 \]

\[ \text{data Even Set} = \text{Set} \]

\[ \text{Branch (Left::Bintree). Bintree = data Tree} \]

The set of binary trees can be defined as follows:

\[ \text{Bintree::Set} = \text{data leaf} \]

\[ \cup \text{branch(left::Bintree), right::Bintree) \]

Example (called "Kleene’s O"): Where \( A \) is one of the types introduced simultaneously.

\[ A = B \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]

\[ B = A \]
Examples

For instance, in the Bintree example, when defining $f :: \text{Bintree} \rightarrow A$ by case-distinction, then the definition of $f$ can make use of $f_{\text{left}}$ and $f_{\text{right}}$.

In the example of $O$, when defining $g :: \text{O} \rightarrow A$ by case-distinction, then the definition of $g$ can make use of $g_{\text{left}}$ and $g_{\text{right}}$.

- In the Bintree example, when defining $f$, $f :: \text{Bintree} \rightarrow A$.
- In the example of $O$, when defining $g$, $g :: \text{O} \rightarrow A$.

For instance,