4. Data Types

(a) The set of Booleans.
(b) The finite sets.
(c) Atomic formulae and the traffic light example. (The example will probably be omitted 2005).
(d) The disjoint union of sets and disjunction.
(e) The $\Sigma$-set. (Will be omitted 2005.)
(f) Predicate Logic in Dependent Type Theory.
(g) Natural Deduction and Dependent Type Theory. (Will be largely omitted 2005).
(h) The set of natural numbers.
(i) Lists. (Will probably be omitted 2005.)
(j) Universes. (Will probably be omitted 2005.)
(k) Algebraic types. (Will be omitted 2005.)

(a) The Set of Booleans

**Formation Rule**

$$\text{Bool} : \text{Set} \quad (\text{Bool-F})$$

**Introduction Rules**

$$\text{tt} : \text{Bool} \quad (\text{Bool-I}_0) \quad \text{ff} : \text{Bool} \quad (\text{Bool-I}_1)$$

**Elimination Rule**

$$\frac{C : \text{Bool} \rightarrow \text{Set} \quad \text{ic} : C \text{ tt} \quad \text{ec} : C \text{ ff} \quad \text{cond} : \text{Bool}}{\text{Case}_\text{Bool} C \text{ ic ec cond} : C \text{ cond} \quad (\text{Bool-El})$$

**Equality Rules**

$$\frac{C : \text{Bool} \rightarrow \text{Set} \quad \text{ic} : C \text{ tt} \quad \text{ec} : C \text{ ff}}{\text{Case}_\text{Bool} C \text{ ic ec tt} = \text{ic} : C \text{ tt} \quad (\text{Bool-Eq}_0)}$$

$$\frac{C : \text{Bool} \rightarrow \text{Set} \quad \text{ic} : C \text{ tt} \quad \text{ec} : C \text{ ff}}{\text{Case}_\text{Bool} C \text{ ic ec ff} = \text{ec} : C \text{ ff} \quad (\text{Bool-Eq}_1)}$$

Further we have equality rules of the formation-, introduction- and elimination-rules.

**Remarks**

- In the above
  - $\text{tt}$ stands for true, $\text{ff}$ stands for false.
  - $\text{ic}$ stands for “if-case”, $\text{ec}$ for “else-case”.
  - $\text{cond}$ for “condition”.
- Therefore $\text{Case}_\text{Bool} C \text{ ic ec b}$ can be read as
  $$\text{if b then ic else ec}$$
  where the additional argument $C$ is required in order to determine the type of $\text{ic}$, of $\text{ec}$, and of the result of this construct.
The argument $C : \text{Bool} \rightarrow \text{Set}$ denotes the set into which we are eliminating.

Instead of $C : \text{Set}$, we have $C : \text{Bool} \rightarrow \text{Set}$, since the set into which we are eliminating might depend on the Boolean value.

That is necessary in order to define functions $f : (b : \text{Bool}) \rightarrow D$ where $D$ depends on $b$.

If we define

$$f := \lambda b. \text{Case}_\text{Bool} C \ ic \ ec \ b$$

we have:

- $f \ tt : C \ tt$.
- $f \ ff : C \ ff$.
- $f : (b : \text{Bool}) \rightarrow C \ b$.

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

$$\text{Case}_\text{Bool} : (C : \text{Bool} \rightarrow \text{Set}) \rightarrow (ic : C \ tt) \rightarrow (ec : C \ ff) \rightarrow (\text{cond} : \text{Bool}) \rightarrow C \ \text{cond}$$

$tt$, $ff$ are the constructors of $\text{Bool}$.

The argument $C$ above has no computational content. It is not needed in order to compute $\text{Case}_\text{Bool} C \ ic \ ec \ tt$ and $\text{Case}_\text{Bool} C \ ic \ ec \ ff$.

$C$ is only needed in order to obtain decidable type checking:

In the presence of arguments like this we can decide whether a judgement $a : B$ is derivable.

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

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

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

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

So we obtain functions from $\text{Bool}$ into other sets without having to write $\lambda b. \text{Bool} \ldots$.

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:

- \((+) : \text{int} \to \text{int} \to \text{int}\).
- \((+) 3\) is the function which takes an integer and adds to it 3.
- **Shorter** than writing \(\lambda x. \text{int}. 3 + x\).

Select Example

- Assume we have introduced in type theory:

  
  \[
  \begin{align*}
  \text{Names} & : \text{Bool} \to \text{Set} , \\
  \text{Names} \text{ tt} & = \text{MaleNames} , \\
  \text{Names} \text{ ff} & = \text{FemaleNames} .
  \end{align*}
  \]

Remarks (Cont.)

- Note that we have the following **order of the arguments** of \(\text{CaseBool}\):
  - 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 \(\text{CaseBool}\) is a "then _else _if " – the **condition (if ... ) is the last one**.
Select Example

We verify the correctness of SelectBool:

SelectBool \( tt \) = CaseBool Names Tim Sara \( tt \) = Tim,
SelectBool \( ff \) = CaseBool Names Tim Sara \( ff \) = Sara.

Jump over AND

---

Example: AND

We want to introduce conjunction

\[
\text{AND} : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}.
\]

This will be of the form

\[
\text{AND} = \lambda(b, c : \text{Bool}).t
\]

for some term \( t \).

\( t \) will be defined by case distinction on \( b \), so we get

\[
\text{AND} = \lambda(b, c : \text{Bool}).\text{CaseBool } e f b
\]

for some \( e, f \).
Two Meanings of Elements of Set

- All elements $A$ of Set have these two meanings:
  - They can be used as terms, which are elements of the type $\text{Set}$.
  - The corresponding judgements are $A : \text{Set}$, $A = A' : \text{Set}$.
- And they can be used as sets, which have elements.
  - The corresponding judgements are $a : A$ and $a = a' : A$.

Example: AND

- In total we define therefore
  $$\text{AND} = \lambda(b, c : \text{Bool}).\text{Case}_{\text{Bool}} (\lambda d : \text{Bool}.\text{Bool}) c \text{ ff } b$$
  $$: \quad \text{Bool} \to \text{Bool} \to \text{Bool}$$
  
- We verify the correctness of this definition:
  - $\text{AND tt c} = \text{Case}_{\text{Bool}} (\lambda d : \text{Bool}.\text{Bool}) c \text{ ff tt} = c.$
    as desired.
  - $\text{AND ff c} = \text{Case}_{\text{Bool}} (\lambda d : \text{Bool}.\text{Bool}) c \text{ ff ff} = \text{ff}.$
    Correct as desired.
  
Jump over derivation of AND

Example: AND

- So
  $$\text{AND} = \lambda(b, c : \text{Bool}).\text{Case}_{\text{Bool}} (\lambda d : \text{Bool}.\text{Bool}) e f b$$
  
  for some $e, f$.
  
  For conjunction we have:
  - If $b$ is true then
    $$b \land c = \text{tt} \land c = c$$
    
    So the if-case $e$ above is $c$.
  - If $c$ is false then
    $$b \land c = \text{ff} \land c = \text{ff}$$
    
    So the else-case $f$ above is $\text{ff}$.

Derivation of AND

- We derive in the following $\text{AND} : \text{Bool} \to \text{Bool} \to \text{Bool}$.
  
- We write $\text{Bool}$, if it
  - is a type in boldface red,
  - and if it is a term, in italic blue.
Derivation of AND

First we derive

\[ b : \text{Bool}, c : \text{Bool} \Rightarrow \lambda (d^{\text{Bool}}). \text{Bool} : \text{Bool} \rightarrow \text{Set} : \]

\[ \frac{\begin{array}{c}
\text{Bool} : \text{Set} \quad (\text{Context}_1) \\
b : \text{Bool} \Rightarrow \text{Context} \quad (\text{Bool-F}) \\
b : \text{Bool} \Rightarrow \text{Bool} : \text{Set} \quad (\text{Context}_1) \\
b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context} \quad (\text{Bool-F}) \\
b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Bool} : \text{Set} \quad (\text{Context}_1) \\
b : \text{Bool}, c : \text{Bool}, d : \text{Bool} \Rightarrow \text{Context} \quad (\text{Bool-F}) \\
b : \text{Bool}, c : \text{Bool}, d : \text{Bool} \Rightarrow \text{Bool} : \text{Set} \quad (\rightarrow \text{-I}) \\
\end{array}}{b : \text{Bool}, c : \text{Bool} \Rightarrow \lambda d^{\text{Bool}}. \text{Bool} : \text{Bool} \rightarrow \text{Set}} \]

Similarly follows

\[ b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Bool} = (\lambda d^{\text{Bool}}. \text{Bool}) \text{ ff} : \text{Set} \]

We derive using part of the proof above, we derive

\[ b : \text{Bool}, c : \text{Bool} \Rightarrow c : (\lambda d^{\text{Bool}}. \text{Bool}) \text{ tt} \]

\[ \frac{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context} \quad (\text{Ass})}{b : \text{Bool}, c : \text{Bool} \Rightarrow c : (\lambda d^{\text{Bool}}. \text{Bool}) \text{ tt} : \text{Set}} \]

\[ \frac{b : \text{Bool}, c : \text{Bool} \Rightarrow c : (\lambda d^{\text{Bool}}. \text{Bool}) \text{ tt} \quad (\text{Transfer}_0)}{b : \text{Bool}, c : \text{Bool} \Rightarrow c : (\lambda d^{\text{Bool}}. \text{Bool}) \text{ tt} : \text{Set}} \]

We derive using (Transfer_0)

\[ b : \text{Bool}, c : \text{Bool} \Rightarrow \text{ff} : (\lambda d^{\text{Bool}}. \text{Bool}) \text{ ff} \]

\[ \frac{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context} \quad (\text{Ass})}{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{ff} : (\lambda d^{\text{Bool}}. \text{Bool}) \text{ ff} : \text{Set}} \]

\[ \frac{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{ff} : (\lambda d^{\text{Bool}}. \text{Bool}) \text{ ff} \quad (\text{Transfer}_0)}{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{ff} : (\lambda d^{\text{Bool}}. \text{Bool}) \text{ ff} : \text{Set}} \]
Derivation of AND

We derive \( b : \text{Bool} \), \( c : \text{Bool} \) using part of the proof above:

\[
\begin{align*}
\text{b : Bool, c : Bool \Rightarrow b : Bool} & \quad \text{(Ass)} \\
\end{align*}
\]

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

\[
\begin{align*}
b &: \text{Bool}, c &: \text{Bool} \Rightarrow \lambda d : \text{Bool}. \text{Bool} &: \text{Bool} \Rightarrow \text{Set} \\
b &: \text{Bool}, c &: \text{Bool} \Rightarrow c : (\lambda d : \text{Bool}. \text{Bool}) \text{ tt} \\
b &: \text{Bool}, c &: \text{Bool} \Rightarrow \text{ff} : (\lambda d : \text{Bool}. \text{Bool}) \text{ ff} & \\
b &: \text{Bool}, c &: \text{Bool} \Rightarrow \text{Case}_{\text{Bool}} (\lambda d : \text{Bool}. \text{Bool}) \text{ c ff b : Bool} \quad \text{(Bool-El)} \\
b &: \text{Bool} \Rightarrow \lambda c : \text{Bool}. \text{Case}_{\text{Bool}} (\lambda d : \text{Bool}. \text{Bool}) \text{ c ff b : Bool} \quad (\rightarrow - I) \\
\lambda : b : \text{Bool}. \text{Case}_{\text{Bool}} (\lambda d : \text{Bool}. \text{Bool}) \text{ c ff b : Bool} : \text{Bool} \Rightarrow \text{Bool} \\
\end{align*}
\]

Elimination into Type

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

Elimination Rule into Type

\[
\frac{C : \text{Type} \quad ic : C \text{ tt} \quad ec : C \text{ ff} \quad \text{cond} : \text{Bool}}{\text{Case}_{\text{Type}} C \text{ ic ec cond} : C \text{ cond}} \quad \text{(Bool-ElType)}
\]

Equality Rules into Type

\[
\frac{C : \text{Type} \quad ic : C \text{ tt} \quad ec : C \text{ ff}}{\text{Case}_{\text{Type}} C \text{ ic ec} \text{ tt = ec : C ff}} \quad \text{(Bool-EqType)}
\]

Elimination into Type (Cont.)

We can extend this into an elimination rule into Kind or other higher types.
Bool in Agda

Unfortunately, Bool, True, False are reserved keywords in Agda, so we use Bool’, True’, False’ instead.

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

\[
data \text{Bool’} = \text{tt} | \text{ff}
\]

This introduces as well constants

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

With this syntax, each constructor can occur at most once in a data type, i.e. we cannot define a second type having constructor tt, e.g. for defining True’ (which is used later):

\[
data \text{True’} = \text{tt}
\]

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

\[
\begin{align*}
\text{Bool’} &:: \text{Set} \\
&= \text{data tt} | \text{ff} \\
\text{tt} &:: \text{Bool’} \\
&= \text{tt@Bool’} \\
\text{ff} &:: \text{Bool’} \\
&= \text{ff@Bool’}
\end{align*}
\]

The official notation for a constructor of a set \( A \) is \( C@A \).

The notation \( C@A \) is, what is displayed, when evaluating expressions in Agda.

This notation is necessary, since a constructor might belong to different sets.

For instance we can introduce both

\[
\begin{align*}
\text{Bool’} &:: \text{Set} \quad = \quad \text{data tt} | \text{ff} \\
\text{True’} &:: \text{Set} \quad = \quad \text{data tt}
\end{align*}
\]

In this situation we need to be able to distinguish between \( \text{tt@Bool’} \) and \( \text{tt@True’} \) in order to get decidable type checking.
Notation for Constructors

It was not possible in Agda to avoid the use of @ by forcing the user to use different names for constructors; we will later introduce as well sets of the form

\[ D \ (a :: A) \]
\[ :: \text{Set} \]
\[ = \text{data } C \cdot \cdot \cdot \]

Then \( C@D \cdot \cdot \cdot \) can be a constructor of \( Da \) for any \( a : A \).

Using \( C \) alone would cause problems with decidable type checking.

If Agda can resolve the type itself, one can write however \( C@_ \) instead of \( C@A \).

Evaluation of terms in Agda

Agda has several methods for evaluating expressions:

- \texttt{agda-compute-whnf}, “Compute weak head normal form”,
- \texttt{agda-compute-whnfs}, “Compute weak head normal form strict”,
- \texttt{agda-nfC}, “Compute to a depth”,
- \texttt{agda-nfC100}, “Compute to depth 100”.

Notation for Constructors

However the abbreviation

\[ \text{data } A = C | D | \cdot \cdot \cdot \]

can be used only if

- one is defining a set \( A \) (and not a type \( A \)),
- and if the set one is defining has no parameter.

So it cannot be used in order to define

\[ D \ (a :: A) \]
\[ :: \text{Set} \]
\[ = \text{data } C \cdot \cdot \cdot \]
Evaluation of terms in Agda

- The above mentioned methods can be executed (directly or by using the goal-menu), while in a goal.
  - An expression typed into the goal will be taken as default input to that function.
  - But that can be modified.
- The methods follow different evaluating strategies.
  - Compute weak head normal form reduces a term until it starts with a constructor or the outer most function doesn’t reduce any further, even if its arguments are evaluated.
  - Compute to depth 100 seems to work best in most cases.

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:
    \[
    f \ (x :: \text{Bool}') \\
    :: \quad \text{Bool}' \\
    = \quad \{ ! ! \}
    \]

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 \( \text{tt} \) or \( \text{ff} \).
  - The goal expands to:
    \[
    f \ (x :: \text{Bool}') \\
    :: \quad \text{Bool}' \\
    = \quad \text{case } x \text{ of} \\
    \quad (\text{tt}) \rightarrow \{ ! ! \} \\
    \quad (\text{ff}) \rightarrow \{ ! ! \}
    \]

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 one of the methods for evaluating expressions, e.g. compute weak head normal form strict.
  - Then type into the mini-buffer \( x \).
  - One gets the answer
    \( \text{tt} \_ \_ \).
Case Distinction (Cont.)

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

Now we can solve the new goals by inserting
- \( \text{ff} \) into the first one,
- \( \text{tt} \) into the second one.

We obtain a function:
\[
f \quad (x :: \text{Bool}') :: \text{Bool}' = \text{case } x \text{ of } \\
(\text{tt}) \rightarrow \text{ff} \\
(\text{ff}) \rightarrow \text{tt}
\]
- \( f \; x \) is the negation of \( x \).

Testing the Defined Function

- We can test our function by using one of the evaluation methods of Agda, e.g. compute weak head normal form strict.
- 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 \( \text{tt} \) or \( \text{ff} \).
  - 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:
    \[
    \text{test} :: \text{Set} \\
    = \{! !\}
    \]
  - move to the new goal
  - choose compute weak head normal form strict or another evaluation method of Agda,
  - and type into the mini-buffer \( f \; \text{tt} \).
  - The result shown is \( \text{ff}@\).
(b) The Finite Sets

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

**Formation Rule**

\[
\text{Fin}_n : \text{Set} \quad (\text{Fin}_n-F)
\]

**Introduction Rules**

\[
\text{A}_k^n : \text{Fin}_n \quad (\text{Fin}_n-\text{I}_k)
\]

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

---

The Finite Sets (Cont)

**Equality Rules**

\[
\begin{align*}
C : \text{Fin}_n &\rightarrow \text{Set} \\
s_0 : C A_0^n \\
s_1 : C A_1^n \\
\cdots \\
s_{n-1} : C A_{n-1}^n \\
\text{Case}_n C s_0 \ldots s_{n-1} A_k^n = s_k : C A_k^n \quad \text{(Fin}_n\text{-Eq}_k)
\end{align*}
\]

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

We add as well equality versions of the formation-, introduction-, and elimination rules. **Remark:** Note that we have just introduced infinitely many rules (for each \(n \in \mathbb{N}\) and \(k = 0, \ldots, n - 1\)).

---

Rules for Fin\(_n\)

**Elimination Rule**

\[
\begin{align*}
C : \text{Fin}_n &\rightarrow \text{Set} \\
s_0 : C A_0^n \\
s_1 : C A_1^n \\
\cdots \\
s_{n-1} : C A_{n-1}^n \\
a : \text{Fin}_n \\
\text{Case}_n C s_0 \ldots s_{n-1} a : C a \quad \text{(Fin}_n\text{-El)}
\end{align*}
\]

**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, when writing down equality rules.

So we write for instance for the previous rule:

\[
\text{Case}_n C s_0 \ldots s_{n-1} A_k^n = s_k : C A_k^n
\]

We sometimes even omit the type:

\[
\text{Case}_n C s_0 \ldots s_{n-1} A_k^n = s_k
\]
More Compact Elimination Rules

Case \( n \) : \((C : \text{Fin}_n \rightarrow \text{Set}) \)
\[ \rightarrow (s_0 : C A^n_0) \]
\[ \rightarrow \ldots \]
\[ \rightarrow (s_{n-1} : C A^n_{n-1}) \]
\[ \rightarrow (a : \text{Fin}_n) \]
\[ \rightarrow C a \]

Elimination into Type

Similarly as for \( \text{Bool} \) 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 \( \text{Fin}_n \) for \( n = 1 \) (we write \( \text{true} \) for \( A^n_1 \)):

Formation Rule

\[ \text{True} : \text{Set} \] (True-F)

Introduction Rules

\[ \text{true} : \text{True} \] (True-I)

Elimination Rule

\[
\begin{array}{c}
C : \text{True} \rightarrow \text{Set} \\
\text{Case}_{\text{True}} c \quad t : \text{True} \\
c t : C t
\end{array}
\] (True-El)

Equality Rule

\[
\begin{array}{c}
C : \text{True} \rightarrow \text{Set} \\
\text{Case}_{\text{True}} c \quad \text{true} = c : C \text{true}
\end{array}
\] (True-Eq)

We add as well equality versions of the formation-, introduction-, and elimination rules.

Jump over next slide (advanced material)
Rules for True (Cont.)

- Case True is computationally not very interesting.
- Case True \( c \) is the constant function \( \lambda x True \cdot c \).
- However, in Agda we might not be able to derive
  \[ \lambda t True' \cdot c : (t : True') \rightarrow C \cdot t \]

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 \( \text{True} \) are indistinguishable from true, i.e. they are identical to true.
- This equality is called Leibnitz equality.

Formulæ as Types

- In type theory, formulæ are certain types.
- A formulæ expressed as a type, is type-theoretically true, if it has an element.
- The elements of such a type are proofs of this formulæ.
- Therefore
  - Truth in type theory means provability.
- True has exactly one proof, and corresponds therefore to the always always true formulæ.

Rules for False

False is the special case \( \text{Fin}_n \) for \( n = 0 \):

- **Formation Rule**
  \[ \text{False} : \text{Set} \quad (\text{False-} F) \]

- **There is no Introduction Rule**

- **Elimination Rule**
  \[ \frac{C : \text{False} \rightarrow \text{Set}}{f : \text{False} \rightarrow C \cdot f} \quad (\text{False-El}) \]

- **There is no Equality Rule**
  We add as well equality versions of the formation- and elimination rule.

False

- False has no elements.
- It is the formulæ, which is always false, since it has no proofs.
- Often called falsum or absurdity.
**False**

- **Case** \( \text{False} \) expresses: from an element \( f \) of \( \text{False} \) we obtain an element of any set (which might depend on \( f \)).
- Considered as a formula, this means: from a proof of \( \text{False} \) we obtain a proof of every other formula.
- I.e. \( \text{False} \) implies everything.
- In logic this principle is called “\text{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.)**

- **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 \( \text{False} \).

**Finite Sets in Agda**

- **Finite sets** can be introduced by giving one constructor for each element. E.g.
  
  ```agda
data Colour = blue | red | green
```
- With this we obtain \( \text{red} :: \text{Colour} \)

**Finite Sets in Agda (Cont.)**

- Elimination is done via case distinction.
- In the “Colour” example above for instance, we can define
  
  ```agda
  is_red (c :: Colour) :: Bool
  = case c of
    (red) → tt
    (green) → ff
    (blue) → ff
  ```
True in Agda

- In the current version of Agda, True and False are reserved keywords for the built-in type Bool. Therefore we have to rewrite them as `True'`, `False'`.
- The definition of True in Agda is straightforward:
  \[
  \text{data } \text{True}' = \text{true}
  \]
- Case distinction will require to solve the case `true`:
  \[
  g \ (x :: \text{True}')
  \lll \text{Bool}'
  = \text{case } x \text{ of}
  \lll (\text{true}) \rightarrow \text{! !}
  \]

False in Agda

- In Agda we can define the empty set as a “data”-set with no constructors:
  \[
  \text{data } \text{False}' =
  \]
- If we want to solve
  \[
  g \ (x :: \text{False}')
  \lll \text{Bool}'
  = \text{! !}
  \]
  we can insert into the goal `x` and choose menu-item “agda-case”.

True in Agda (Cont.)

- Alternatively, we can define True in Agda as the empty sig:
  \[
  \text{True}' = \text{sig}\{
  \}
  \]
- Then the element `true` of `True'` is defined as follows
  \[
  \text{true} = \text{struct}\{
  \} :: \text{True}'
  \]
- However, since we have no \(\eta\)-rule, we don’t get that for `a :: True'` we have `a = true`.

False in Agda (Cont.)

- The result is
  \[
  g \ (x :: \text{False}')
  \lll \text{Bool}'
  = \text{case } x \text{ of } \{ \}
  \]
- 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

1. Assume the type of trees:
   
   ```
   data Tree = oak | pine | spruce
   ```

2. Below we will show, how to introduce a function

   ```
   IsConifer :: Tree → Set
   ```

   s.t.

   ```
   IsConifer oak = False'
   IsConifer pine = True'
   IsConifer spruce = True'
   ```

Example 2 for the Use of False

Similarly we can introduce a stack of elements of type `A`, together with a predicate

```
NonEmpty :: Stack → Set
``` 

s.t.

```
NonEmpty nil = False'
``` 

where nil is the empty stack:

```
NonEmpty (s :: Stack) :: Set
``` 

= case s of 

```
  (nil) → False'
  (cons a l) → True'
```
Example 2 for the Use of False

Now we can define

top \((s :: \text{Stack})\)
\((p :: \text{NonEmpty} \ s)\)
:= \ A
    \text{case } s \text{ of }
    \ (\text{nil}) \rightarrow \text{case } p \text{ of } \{ \}
    (\text{cons } a \ s') \rightarrow a

(See exampleStack.agda).

Again we don’t have to provide a result, in case \(s\) is empty.

(c) Atomic Formulae

Full title of this section:
Atomic formulae and the Traffic Light Example.

We have introduced two formulae:
- True, the always true formula.
  Corresponds to truth value \(tt : \text{Bool}\).
- False, the always false formula.
  Corresponds to truth value \(ff : \text{Bool}\).

Atomic Formulae

A formula expressing equality between two elements of \(\text{Fin}_n\) (for fixed \(n\)) can now be introduced as follows:

Define a function

\[
\text{Eq}_{\text{In}, \text{Bool}} : \text{Fin}_n \rightarrow \text{Fin}_n \rightarrow \text{Bool}
\]

s.t.

\[
\text{Eq}_{\text{In}, \text{Bool}} \ A^n_i A^n_j = \begin{cases} 
\text{true} & \text{for } i = j \\
\text{false} & \text{for } i \neq j
\end{cases}
\]

\(\text{Eq}_{\text{In}, \text{Bool}}\) can be defined easily (for fixed \(n\)) by case distinction on its two arguments.

Now apply an operation

\[
\text{atom} : \text{Bool} \rightarrow \text{Set}
\]

which maps the truth value to the corresponding formula, i.e. define now

\[
\text{Eq}_n : \text{Fin}_n \rightarrow \text{Fin}_n \rightarrow \text{Set}
\]

\[
\text{Eq}_n \ x \ y = \text{atom}(\text{Eq}_{\text{In}, \text{Bool}} \ x \ y)
\]
Atomic Formulae

- Atom is defined s.t.
  
  \[
  \text{atom } \text{tt} = \text{True} \\
  \text{atom } \text{ff} = \text{False}
  \]

- So we get for \(\text{Eq}_n\) above
  
  \[
  \text{Eq}_n \ A^i_n \ A^j_n = \text{True} \\
  \text{Eq}_n \ A^j_n \ A^i_n = \text{False} \quad \text{for } i \neq j
  \]

- So
  
  \[
  \text{Eq}_n \ A^i_n \ A^j_n \text{ is inhabited, has a proof, is true;} \\
  \text{for } i \neq j, \text{Eq}_n \ A^j_n \ A^i_n \text{ is not inhabited, has not a proof, is false.}
  \]

atom in Agda

\[
\text{atom } (b :: \text{Bool}') \\
:: \text{Set} \\
= \text{case } b \text{ of} \\
(\text{tt}) \rightarrow \text{True}' \\
(\text{ff}) \rightarrow \text{False}'
\]

Decidable Predicates

- In general, atom allows us to define decidable predicates on sets.
- A predicate is decidable if it can be decided by a Boolean valued function.
- E.g. equality on natural numbers is decidable, since we can define a function
  \(\text{Eq}_{\text{N, Bool}} : \text{N} \rightarrow \text{N} \rightarrow \text{Bool}\) which decides it.
- Equality on functions \(\text{N} \rightarrow \text{N}\) is undecidable, since we cannot define such a function – in order to check equality between \(f\) and \(g\) we need to check equality between \(f \ n\) and \(g \ n\) for all \(n : \text{N}\).
Decidable Predicates (Cont.)

- Assume we have a set of states of a system \( A \).
  - E.g. the set of states a railway controller can choose.
- Assume we have a function \( f : A \to \text{Bool} \).
  - E.g. \( f \ a \) means: state \( a \) is safe.

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 a \( h : (a : A) \to g \ a \) 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.

Jump over Traffic Light Example.

The Traffic Light Example

- Assume a road crossing, controlled by traffic lights:

The Set of Physical States

- For simplicity assume that each traffic light is either red or green:
  
  \[
  \text{data Colour} = \text{red} \mid \text{green}
  \]

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

\[
\text{Phys\_State} :: \text{Set} \\
= \text{sig}
\]

\[
\text{sigA :: Colour} \]

\[
\text{sigB :: Colour}
\]
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
  
  ```haskell
  data Control_State = Allred | Agreen | Bgreen
  ```

Control States to Physical States

- We define the state of signals A, B depending on a control state:
  
  ```haskell
  toSigA (s :: Control_State) :: Colour
  = case s of
    (Allred)   -> red
    (Agreen)   -> green
    (Bgreen)   -> red
  ```

- Control States to Physical States
  
  ```haskell
  toSigB (s :: Control_State) :: Colour
  = case s of
    (Allred)   -> red
    (Agreen)   -> red
    (Bgreen)   -> green
  ```
Now we can define the **physical state corresponding to a control state**:

\[
\text{phys\_state} \ (s :: \text{Control\_State}) \\
:: \text{Phys\_State} \\
= \text{struct} \\
\text{sigA} = \text{toSigA} \ s \\
\text{sigB} = \text{toSigB} \ s
\]

**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:

\[
\text{CorAux} \ (a, b :: \text{Colour}) \\
:: \text{Set} \\
= \text{case} \ a \ \text{of} \\
\text{red} & \rightarrow \text{true}' \\
\text{green} & \rightarrow \text{case} \ b \ \text{of} \\
\text{red} & \rightarrow \text{true}' \\
\text{green} & \rightarrow \text{false}'
\]

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

- In the current example this wouldn’t have caused problems, but in more complex examples it does (due to the lack of the \(\eta\)-rule).

**Safety of the System**

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

\[
\text{cor\_proof} \ (s :: \text{Control\_State}) \\
:: \text{Cor(phys\_state} \ s) \\
= \text{case} \ s \ \text{of} \\
\text{Allred} & \rightarrow \text{true} \\
\text{Agreen} & \rightarrow \text{true} \\
\text{Bgreen} & \rightarrow \text{true}
\]

\[
\text{Cor} \ (s :: \text{Phys\_State}) \\
:: \text{Set} \\
= \text{CorAux} \ s.sigmA \ s.sigmB
\]
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.

(d) The Disjoint Union of Sets

- The disjoint union \( A + B \) of two sets \( A \) and \( B \) is the union of \( A \) and \( B \), but defined in such a way that we can decide whether an element of this union is originally from \( A \) or \( B \).
- This is distinguished by having constructors \( \text{inl} : A \rightarrow A + B \) and \( \text{inr} \).
- Elements from \( a : A \) are inserted into \( A + B \) as \( \text{inl} a : A + B \).
- Elements from \( b : B \) are inserted into \( A + B \) as \( \text{inr} b : A + B \).
- \( \text{inl} \) stands for “in-left”, \( \text{inr} \) for “in-right”.
- If we have \( a : A \) and \( a : B \), then \( a \) is represented both as \( \text{inl} a \) and \( \text{inr} a \) in \( A + B \).

Disjoint Union

- Informally, if

\[
A = \{1, 2\}
\]

and

\[
B = \{3, 4, 5\}
\]

then

\[
A + B = \{\text{inl}(1), \text{inl}(2), \text{inr}(3), \text{inr}(4), \text{inr}(5)\}
\]

- Each element of \( A + B \) is
  - either of the form \( \text{inl}(a) \) for some \( a : A \)
  - or of the form \( \text{inr}(b) \) for \( b : B \).

Jump over Comparision with Product
Comparison with the Product

Note that if we have again
\[ A = \{1, 2\} \]
and
\[ B = \{3, 4, 5\} \]
then for the product we have informally
\[ A \times B = \{p(1, 3), p(1, 4), p(1, 5), p(2, 3), p(2, 4), p(2, 5)\} \]

Each element of \( A \times B \) is of the form \( p(a, b) \) where \( a : A \) and \( b : B \).

So each element of \( A \times B \) contains both an element of \( A \) and an element of \( B \).

Disjoint Union vs. Product

Note that, if \( A \) is empty, then
\[ A + B = \{\text{inr}(b) \mid b : B\} \], which has a copy of each element of \( B \).

\( A \times B \) is empty, since we cannot form a pair \( p(a, b) \) where \( a : A, b : B \), since there is no element \( a : A \).

Rules for \( A + B \)

**Formation Rule**

\[
\frac{A : \text{Set} \quad B : \text{Set}}{A + B : \text{Set}} (\text{+F})
\]

**Introduction Rules**

\[
\begin{align*}
\frac{A : \text{Set} \quad B : \text{Set}}{\text{inl} \ A \ B \ a : A + B} (\text{+I}_{\text{inl}}) \\
\frac{A : \text{Set} \quad B : \text{Set}}{\text{inr} \ A \ B \ b : A + B} (\text{+I}_{\text{inr}})
\end{align*}
\]

**Elimination Rules**

\[
\begin{align*}
\text{Case}_{+} \ A \ B \ C \ cl \ cr \ d : C \ d (\text{+E}_{\text{I}})
\end{align*}
\]

\((cl, cr) \text{ stand for “case left”, “case right”} \).
Rules for $A + B$

**Equality Rules**

\[
\begin{align*}
\text{Case}_{+} \ A \ B \ C \ \text{cl} \ \text{cr} \ (\text{inl} \ A \ B \ a) &= \ \text{cl} \ a : C \ (\text{inl} \ A \ B \ a) \quad (\text{+-Eq}_{\text{inl}}) \\
\text{Case}_{+} \ A \ B \ C \ \text{cl} \ \text{cr} \ (\text{inr} \ A \ B \ b) &= \ \text{cr} \ b : C \ (\text{inr} \ A \ B \ b) \quad (\text{+-Eq}_{\text{inr}})
\end{align*}
\]

Additionally we have the **equality versions** of the formation-, introduction and elimination rules.

---

**Disjoint Union in Agda**

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

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

---

**Logical Framework Version**

A more compact notation for the formation, introduction and equality rules is:

- $(+): \text{Set} \to \text{Set} \to \text{Set}$, written infix.
- $\text{inl}: (A, B : \text{Set}) \to A \to (A + B)$.
- $\text{inr}: (A, B : \text{Set}) \to B \to (A + B)$.
- \[\text{Case}_{+}: (A, B : \text{Set}) \rightarrow (C : (A + B) \rightarrow \text{Set}) \rightarrow ((a : A) \rightarrow C \ (\text{inl} \ A \ B \ a)) \rightarrow ((b : B) \rightarrow C \ (\text{inr} \ A \ B \ b)) \rightarrow (d : A + B) \rightarrow C \ d.\]

---

**Disjoint Union in Agda (Cont.)**

- The notation $(+)$ means, that $+$ can be used **infix**.
- Now we have, if $A, B :: \text{Set}$:
  - $\text{inl@}(A + B) :: A \to (A + B)$
  - $\text{inr@}(A + B) :: B \to (A + B)$
- This can be checked using the menu “infer type” in a dummy goal.
- Note that we cannot assign a type to $\text{inl@}_-$ or $\text{inr@}_-$.
- $(+)$ **cannot** be defined using the abbreviated data notation (which would be of the form data $(+) = \cdots$).
Disjoint Union in Agda (Cont.)

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

\[
\begin{align*}
  f & (c :: A + B) \\
  :: & \text{Bool}' \\
  = & \text{case } c \text{ of} \\
  (\text{inl } a) & \to \{! !\} \\
  (\text{inr } b) & \to \{! !\}
\end{align*}
\]

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

Use of Concrete Disjoint Sets

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

\[
\text{data Plant} = \text{tree}(t :: \text{Tree}) \mid \text{flower}(f :: \text{Flower})
\]

Now one can define for instance

\[
\text{isFlower } (p :: \text{Plant}) \\
:: \text{Bool}'
\]

\[
= \text{case } p \text{ of} \\
  (\text{tree } t) & \to \text{ff} \\
  (\text{flower } f) & \to \text{tt}
\]

Disjoint Union in Agda (Cont.)

We obtain

\[
\begin{align*}
  f & (c :: A + B) \\
  :: & \text{Bool}' \\
  = & \text{case } c \text{ of} \\
  (\text{inl } a) & \to \{! !\} \\
  (\text{inr } b) & \to \{! !\}
\end{align*}
\]

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

Disjunction

\( A \lor B \) is true iff \( A \) is true or \( B \) is true.

Therefore a proof of \( A \lor B \) consists of a proof of \( A \) or a proof of \( B \), plus the information which one.

It is therefore an element \( \text{inl } p \) for a proof \( p : A \) or an element \( \text{inr } q \) for a proof \( q : B \).

Therefore the set of proofs of \( A \lor B \) is the disjoint union of \( A \) and \( B \), i.e. \( A + B \).

We can identify \( A \lor B \) with \( A + B \).
Disjunction in Agda

- Or is represented as disjoint union in type theory.
- In Agda we can write \( \lor \) for it (on slides we write \( \lor \)) and define it as follows:

\[
(\lor) \quad (A, B :: \text{Set}) \\
:: \quad \text{Set} \\
= \quad \text{data or1}(a :: A) \mid \text{or2}(b :: B)
\]

- See `exampleproofproplogic7.agda`.

- On the blackboard \( A \rightarrow A \lor B \) and \( A \lor A \rightarrow A \) will now be shown in Agda.

---

Disjunction with more Args.

- As for the conjunction, it is useful to introduce special ternary versions of the disjunction (and versions with higher arities):

\[
\text{OR3} \quad (A, B, C :: \text{Set}) \\
:: \quad \text{Set} \\
= \quad \text{data or1}(a :: A) \mid \text{or2}(b :: B) \mid \text{or3}(c :: C)
\]

- See `exampleproofproplogic8.agda`.

Jump over \( \Sigma \)-Type.

---

Example (Disjunction)

- The following derives \( (A \lor B) \rightarrow (B \lor A) \):

\[
\text{lemma3} \quad (ab :: A \lor B) \\
:: \quad B \lor A \\
= \quad \text{case ab of} \\
(\text{or1 } a) & \rightarrow \text{or2@}_a \\
(\text{or2 } b) & \rightarrow \text{or1@}_b
\]

- See `exampleproofproplogic9.agda`.

---

(e) The \( \Sigma \)-Set

- The \( \Sigma \)-set is a second version of the dependent product of two sets.
- It depends on
  - a set \( A \),
  - and a second set \( B \) depending on \( A \), i.e. on \( B : A \rightarrow \text{Set} \).
- Similar to the standard product \( (x : A) \times (B \ x) \).
- In Agda
  - \( (x : A) \times (B \ x) \) is a in Agda a builtin construct,
  - the \( \Sigma \)-set is introduced by the user using a constructor, similar to the previous sets.
- The \( \Sigma \)-set behaves sometimes better than the standard product.
Rules for $\Sigma$

**Formation Rule**

$$
\begin{array}{c}
A : \text{Set} \\
B : A \rightarrow \text{Set}
\end{array}
\Rightarrow
\begin{array}{c}
\Sigma A B : \text{Set}
\end{array}
\quad (\Sigma\text{-F})
$$

**Introduction Rule**

$$
\begin{array}{c}
A : \text{Set} \\
B : A \rightarrow \text{Set} \\
a : A \\
b : B a \\
p : A B a b
\end{array}
\Rightarrow
\begin{array}{c}
p A B a b : \Sigma A B
\end{array}
\quad (\Sigma\text{-I})
$$

**Elimination Rule**

$$
\begin{array}{c}
A : \text{Set} \\
B : A \rightarrow \text{Set} \\
C : (\Sigma A B) \rightarrow \text{Set} \\
c : (a : A) \rightarrow (b : B a) \rightarrow C (p A B a b)
\end{array}
\Rightarrow
\begin{array}{c}
d : \Sigma A B \\
\text{Case}_\Sigma A B C c d : C d
\end{array}
\quad (\Sigma\text{-El})
$$

**Equality Rule**

$$
\text{Case}_\Sigma A B C c (p A B a b) = c a b : C (p A B a b) 
\quad (\Sigma\text{-Eq})
$$

Additionally we have the **Equality versions** of the formation-, introduction- and elimination-rules.

The $\Sigma$-Set using the Log. Framew.

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

The $\Sigma$-Set using the Log. Framew.
The $\Sigma$-Set and the Dep. Prod.

- Both the $\Sigma$-set and the dep. product have similar introduction rules.
- For the $\Sigma$-set, the constructors have additional arguments $A$, $B$ necessary for bureaucratic reasons only.
- One can define the projections $\pi_0$, $\pi_1$ using $\text{Case}_\Sigma$:

$$
\pi_0 = \text{Case}_\Sigma A B (\lambda x(\Sigma A B). A) (\lambda x.A. \lambda y(B x). x) \\
\pi_1 = \text{Case}_\Sigma A B (\lambda x(\Sigma A B). B \pi_0(x)) (\lambda x.A. \lambda y(B x). y)
$$

On the other hand, from $\pi_0$, $\pi_1$ we can define $\text{Case}_\Sigma$ as follows:

$$
\lambda A A \to \text{Set}. \lambda B A \to \text{Set}. \lambda C(\Sigma A B) \to \text{Set}. \\
\lambda s(a:A) \to (b:B a) \to C (p a b). \lambda d(\Sigma A B). s \pi_0(d) \pi_1(d).
$$

The $\Sigma$-Set and the Dep. Prod. (Cont.)

However the dependent product has the $\eta$-rule (which is however not implemented in Agda).

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

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

The $\Sigma$-Set in Agda

- $\Sigma$ can be defined as a “data”-set with a constructor, e.g.

$$
\text{Sigma} \quad (A :: \text{Set}) \\
(B :: A \to \text{Set}) \\
:: \quad \text{Set} \\
= \quad \text{data} \ p \ (a :: A) \ (b :: Ba)
$$

- Elimination uses case-distinction:

$$
f \quad (c :: \text{Sigma} A B) \\
:: \quad D \\
= \quad \text{case} \ c \ of \\
(p a b) \to \cdots
$$

The $\Sigma$-Set in Agda (Cont.)

Again one usually defines concrete $\Sigma$-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 $\text{Plants\_in\_group} \ g$ for plants in that group.

The set of plants can then be defined as

$$
data \ \text{Plant} = \text{plant} \ (g :: \text{Plant\_Group})(pg :: \text{Plants\_in\_group} \ g)
$$
The \( \Sigma \)-Set in Agda (Cont.)

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

\[
f \quad (p :: \text{Plant})
\begin{align*}
&:: \text{Plant\_group} \\
&= \text{case } p \text{ of} \\
&(\text{plant } g \text{ pg}) \rightarrow g
\end{align*}
\]

(f) Predicate Log. in Dep. Type Theo.

We have already seen how to represent the propositional connectives and decidable atomic formulae in Agda and therefore as well in dependent type theory:

- Implication

\[
A \rightarrow B
\]

is represented as the nondependent function type

\[
A \rightarrow B
\]

- Conjunction

\[
A \land B
\]

is represented as one of the two versions of the product of \( A \) and \( B \).

Negation

We haven’t introduced \( \neg A \) in dependent type theory.

- \( \neg A \) is true iff \( A \) is false iff there is no proof of \( A \).

Now we can show that there is no proof of \( A \) iff \( A \rightarrow \text{False} \) is true:

- If there is no proof of \( A \), then from every proof of \( A \) we can obtain a proof of \( \text{False} \) (since there is no proof of \( A \)); therefore \( A \rightarrow \text{False} \) is true.

- On the other hand, if we \( A \rightarrow \text{False} \) is true, i.e. has a proof, then there cannot be any proof of \( A \), because from it we could get a proof of \( \text{False} \), which is the empty set.

Therefore \( \neg A \) is true iff \( A \rightarrow \text{False} \) is true.

Therefore we can identify \( \neg A \) with \( A \rightarrow \text{False} \).
In this subsection we will investigate, how to represent universal and existential quantification in dependent type theory.

Since we have many types, we have to write when using quantifiers explicitly the type, the bound variable is ranging over:

We write therefore

\[ \forall x : A : B \] for "for all \( x \) of type \( A \), \( B \) holds" (where \( B \) usually depends on \( x \));

\[ \exists x : A : B \] for "there exists an \( x \) of type \( A \), s.t. \( B \) holds" (again \( B \) usually depends on \( x \)).

### Universal Quantification

\[ \forall x : A : B \] 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 set of functions, mapping an element \( x : A \) to an element of \( B \).

This set is just the dependent function type \( (x : A) \to B \).

Therefore we can identify \( \forall x : A : B \) with \( (x : A) \to B \).

### Example \((\forall, \text{Cont.})\)

First we define a Boolean valued less-than relation on \( \text{Bool}' \) as follows:

\[
\text{LessBool} \ (a, b :: \text{Bool}') :: \text{Set} \\
\begin{cases} 
\text{Set} & \text{case } a \text{ of} \\
(tt) & \to \ ff \\
(ff) & \to \ b
\end{cases}
\]

Explanation of this definition:

- if \( a \) is true, then \( a \) is never less than \( b \).
- if \( a \) is false, then \( a \) is less than \( b \) iff \( b \) is true, so the truth value of \( \text{LessBool} \ a \ b \) is the same as \( b \).
Example (∀, Cont.)

Then we define \(<\) as follows

\[
(\langle a, b :: \text{Bool}' \rangle) \quad :: \text{Set} \\
= \text{atom (LessBool } a \ b) 
\]

Lemma4 :: Set

\[
= (a :: \text{Bool}') \rightarrow \text{Not } (a < a) 
\]

Since \(\text{Not } (a < a) = (a < a) \rightarrow \text{False}'\), we have

\[
\text{Lemma4} = (a :: \text{Bool}') \rightarrow \text{Not } (a < a) \\
= (a :: \text{Bool}') \rightarrow (a < a) \rightarrow \text{False}'
\]
Example (\(\forall\), Cont.)

\[ \text{lemma4 :: Lemma4} \]
\[ = \{ ! ! \} \]

Type of goal is \(\text{Lemma4} = (a :: \text{Bool'}) \rightarrow (a < a) \rightarrow \text{False}'\).

- An element of \((a :: \text{Bool'}) \rightarrow (a < a) \rightarrow \text{False}'\) can be introduced by \(\lambda\)-abstracting \(\lambda(a :: \text{Bool'})\) and \(\lambda(aa :: (a < a))\):

\[ \text{lemma4 :: Lemma4} \]
\[ = \lambda(a :: \text{Bool'}) \rightarrow \lambda(aa :: a < a) \rightarrow \{ ! ! \} \]

- The type of goal is now the conclusion of \((a :: \text{Bool'}) \rightarrow (a < a) \rightarrow \text{False}'\), namely \(\text{False}'\).

Example (\(\forall\), Cont.)

\[ \text{lemma4 :: Lemma4} \]
\[ = \lambda(a :: \text{Bool'}) \rightarrow \lambda(aa :: a < a) \rightarrow \text{case } a \text{ of} \]
\[ (tt) \rightarrow \{ ! ! \} \]
\[ (ff) \rightarrow \{ ! ! \} \]

- The type of both goals is the same as before, namely \(\text{False}'\), since it didn’t depend on \(a\).

Example (\(\forall\), Cont.)

We need to make use of our assumptions, namely \(a :: \text{Bool'}\) and \(aa :: a < a\).

- \(a < b\) is defined by case disjunction on \(a\) and \(b\).
- Unless we know that \(a = tt\) or \(a = ff\), we don’t know much about \(a < a\).
- So it seems to be a good step to make case distinction on \(a\).

However, we know now more about the assumptions \(aa :: a < a\).

- In case of \(a = tt\), we have \(a < a = (tt < tt) = \text{False}'\)
- In case of \(a = ff\), we have \(a < a = (ff < ff) = \text{False}'\)
Example (∀, Cont.)

\[ \text{lemma4 :: Lemma4} \]
\[ = \lambda(a :: \text{Bool}) \rightarrow \lambda(aa :: a < a) \rightarrow \]
\[ \text{case } a \text{ of} \]
\[ (tt) \rightarrow \{! !\} \]
\[ (ff) \rightarrow \{! !\} \]

Since in both goals we have \( aa :: (a < a) = \text{False}' \), we can can make case distinction on \( aa \), which is the empty case distinction.

Example (∀, Cont.)

In the previous example,
- the type of goal was \( \text{False}' \),
- and \( aa : \text{False}' \).

So, instead of using the empty case distinction directly, we could have as well inserted \( aa \) in those goals:

\[ \text{lemma4a :: Lemma4} \]
\[ = \lambda(a :: \text{Bool}) \rightarrow \lambda(aa :: a < a) \rightarrow \]
\[ \text{case } a \text{ of} \]
\[ (tt) \rightarrow aa \]
\[ (ff) \rightarrow aa \]

Example (∀, Cont.)

We finish our proof as follows:

\[ \text{lemma4 :: Lemma4} \]
\[ = \lambda(a :: \text{Bool}) \rightarrow \lambda(aa :: a < a) \rightarrow \]
\[ \text{case } a \text{ of} \]
\[ (tt) \rightarrow \text{case } aa \text{ of} \{\} \]
\[ (ff) \rightarrow \text{case } aa \text{ of} \{\} \]

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 \( (x : A) \times B \).
\( \exists \) in Agda

\( \exists x : A.B \) is represented therefore in Agda by one of the two dependent products in Agda.

Using meaningful names, we can define \( \exists x : A.B \) as follows:

\[
\text{Version1} :: \text{Set} \quad = \quad \text{sig} \\
\hspace{1cm} a :: A \\
\hspace{1cm} b :: B[x := a]
\]

\[
\text{Version2} :: \text{Set} \quad = \quad \text{data exists \((a :: A)(b :: B[x := a])\)}
\]

Example (\( \exists \))

As an example,

- we define negation \( \neg \) on Bool,
- define an equality \( == \) on Bool,
- and show \( \forall a : \text{Bool}', \exists b : \text{Bool}'. a == \neg b \).

See \texttt{exampleproofprologic11.agda}.

\( \exists \) in Agda

Above \( B[x := a] \) is the result of substituting in \( B \) for \( x \) the variable \( a \).

Example (\( \exists \), Cont.)

\( \neg \) is defined as follows:

\[
\text{neg} \quad (a :: \text{Bool}') \quad :: \quad \text{Bool}' \\
\quad = \quad \text{case } a \text{ of} \\
\hspace{1cm} (\text{tt}) \rightarrow \text{ff} \\
\hspace{1cm} (\text{ff}) \rightarrow \text{tt}
\]
Example (9)

A Boolean valued equality on $\text{Bool}'$ is defined as follows:

\[
\text{EqBool} \quad (a, b :: \text{Bool}')
\]
\[
:: \text{Bool}'
\]
\[
= \text{case } a \text{ of}
\]
\[
(tt) \rightarrow b
\]
\[
(ff) \rightarrow \neg b
\]

Then we define

\[
(==) \quad (a, b :: \text{Bool}')
\]
\[
:: \text{Bool}'
\]
\[
= \text{atom (EqBool } a b)
\]

Example (9, Cont.)

$(==)$ can be written infix, i.e. we can write $a == b$ for $(==) a b$.

Example (9, Cont.)

In order to introduce the statement mentioned above, we introduce first the formula $\exists b : \text{Bool}'.a == \neg b$ depending on $a : \text{Bool}'$:

\[
\text{Lemma5aux} \quad (a :: \text{Bool}')
\]
\[
:: \text{Set}
\]
\[
= \text{sig}
\]
\[
b :: \text{Bool}'
\]
\[
ab :: a == \neg b
\]

The statement $\forall a : \text{Bool}'.\exists b : \text{Bool}'.a == \neg b$ is now as follows:

\[
\text{Lemma5} :: \text{Set}
\]
\[
= (a :: \text{Bool}') \rightarrow \text{Lemma5aux } a
\]

Example (9, Cont.)

A proof of Lemma5 is an element

\[
\text{lemma5} :: \text{Lemma5}
\]

and we get the goal

\[
\text{lemma5} :: \text{Lemma5}
\]
\[
= \{! !\}
\]

The type of goal is

\[
\text{Lemma5} :: (a :: \text{Bool}') \rightarrow \text{Lemma5aux } a
\]

Any goal of function type is usually best solved by using $\lambda$-abstracting.
Example (3, Cont.)

Lemma5 :: Set
    = (a :: Bool') → Lemma5aux a

We get

lemma5 :: Lemma5
    = λ(a :: Bool')
        → {! !}

The type of the goal is

Lemma5aux a = sig
    b :: Bool'
    ab :: a == neg b

Example (3, Cont.)

We get

lemma5 :: Lemma5
    = λ(a :: Bool')
        → case a of
            (tt) → {! !}
            (ff) → {! !}

In case of \( a = \text{tt} \), the type of goal is

Lemma5aux tt = sig
    b :: Bool'
    ab :: tt == neg b

So we can use goal menu \texttt{intro} and obtain:
Example (§, Cont.)

\[
\text{lemma5 :: Lemma5} = \lambda(a :: \text{Bool}') \rightarrow \text{case } a \text{ of}
\]
\[
\begin{align*}
\text{(tt) } & \rightarrow \text{ struct } \\
& b = \{! !\} \\
& ab = \{! !\}
\end{align*}
\]
\[
\text{(ff) } \rightarrow \{! !\}
\]

The first goal can be solved by setting \(b := \text{ff}\).

Then the type of the second goal is
\[
\begin{align*}
\text{(tt == neg } b) & = \text{(tt == neg } \text{ff)} \\
& = \text{(tt == tt)} \\
& = \text{True}'
\end{align*}
\]
which can be solved by setting \(ab := \text{true}\).

Example (§, Cont.)

So we get:

\[
\text{lemma5 :: Lemma5} = \lambda(a :: \text{Bool}') \rightarrow \text{case } a \text{ of}
\]
\[
\begin{align*}
\text{(tt) } & \rightarrow \text{ struct } \\
& b = \text{ff} \\
& ab = \text{true} \\
\text{(ff) } & \rightarrow \{! !\}
\end{align*}
\]

In case \(a = \text{ff}\), we use again intro and obtain:

Example (§, Cont.)

The case \(a = \text{ff}\) can be solved in a similar way by setting \(b = \text{tt}, ab = \text{true}\).
Complex Example

We assume $A, B : \text{Set}$ and equality relations on $A, B$:

\[
\begin{align*}
\text{postulate } & A :: \text{Set} \\
\text{postulate } & EqA :: A \rightarrow A \rightarrow \text{Set} \\
\text{postulate } & B :: \text{Set} \\
\text{postulate } & EqB :: B \rightarrow B \rightarrow \text{Set}
\end{align*}
\]

We will introduce

- the disjoint union $AB$ of $A$ and $B$
- an equality $EqAB$ on $AB$
- and show that if $EqA$ and $EqB$ are symmetric, so is $EqAB$.

See exampleDisjointUnionEqual.agda.

Equality Sets

$EqA$ (and $EqB$) could be decidable equalities,

i.e. $EqA = \lambda(a, b :: A) \rightarrow \text{atom}(\text{eqboolA } a \ b)$,

where $\text{eqboolA} :: A \rightarrow A \rightarrow \text{Bool}$.

Or an undecidable equality.

E.g. the equality on $N \rightarrow N$ is in standard logic

\[
 f = g :\Leftrightarrow \forall n : N. f(n) = g(n)
\]

which reads in Agda as follows:

\[
\begin{align*}
\text{EqN} \_ \_ (f, g :: N \rightarrow N) \\
:: & \text{Set} \\
= & (n :: N) \rightarrow f \ n == g \ n
\end{align*}
\]

where $==$ is the equality on $N$.

Undecidable Equalities

The last equality is undecidable, since in order to check whether $\text{EqN} \_ N \ f \ g$ holds we have to check for all $n : N$ whether $f \ n = g \ n$ holds.

Complex Example (Cont.)

The formation of $A + B$ is straightforward:

\[
\begin{align*}
(+) & (A, B :: \text{Set}) \\
:: & \text{Set} \\
= & \text{data } \text{inl}(a :: A) \mid \text{inr}(b :: B)
\end{align*}
\]
Complex Example (Cont.)

- We define the equality $\text{Eq}_{AB}$ on $A + B$ as follows:
  - Assume $ab, ab' : A + B$.
  - If one is of the form $\text{inl} \ a$ and the other of the form $\text{inr} \ b$, then $\text{Eq}_{AB} \ ab \ ab'$ should be false, so we define
    \[
    \text{Eq}_{AB} (\text{inl} \ a) (\text{inr} \ b) = \text{Eq}_{AB} (\text{inr} \ a) (\text{inl} \ a) = \text{False}'.
    \]
  - If they are of the form $\text{inl} \ a$ and $\text{inl} \ a'$, respectively, then $\text{Eq}_{AB} \ ab \ ab'$ should be true if $\text{Eq}_A \ a \ a'$ holds.
    - This can be achieved by defining
      
      \[
      \text{Eq}_{AB} (\text{inl} \ a) (\text{inl} \ a') = \text{Eq}_A \ a \ a'.
      \]

Complex Example (Cont.)

- The definition of $\text{Eq}_{AB}$ is as follows:

\[
\begin{align*}
\text{Eq}_{AB} & :: (A + B) \to (A + B) \to \text{Set} \\
& = \lambda (ab, ab' :: A + B) \to \\
& \quad \text{case } ab \text{ of} \\
& \quad \quad (\text{inl} \ a) \to \text{case } ab' \text{ of} \\
& \quad \quad \quad \quad (\text{inl} \ a') \to \text{Eq}_A \ a \ a' \\
& \quad \quad (\text{inr} \ b') \to \text{False}' \\
& \quad \quad (\text{inr} \ b) \to \text{case } ab' \text{ of} \\
& \quad \quad \quad \quad (\text{inl} \ a') \to \text{False}' \\
& \quad \quad (\text{inr} \ b') \to \text{Eq}_B \ b \ b'
\end{align*}
\]

Complex Example (Cont.)

- The following code is equivalent to the previous code:

\[
\begin{align*}
\text{Eq}_{AB} \ (ab, ab' :: A + B) \\
& :: \text{Set} \\
& = \text{case } ab \text{ of} \\
& \quad (\text{inl} \ a) \to \text{case } ab' \text{ of} \\
& \quad \quad (\text{inl} \ a') \to \text{Eq}_A \ a \ a' \\
& \quad (\text{inr} \ b') \to \text{False}' \\
& \quad (\text{inr} \ b) \to \text{case } ab' \text{ of} \\
& \quad \quad (\text{inl} \ a') \to \text{False}' \\
& \quad (\text{inr} \ b') \to \text{Eq}_B \ b \ b'
\end{align*}
\]

The above equations will be definitional equalities, i.e. for instance $\text{Eq}_{AB} (\text{inl} \ a) (\text{inl} \ a')$ will \text{rewrite} to $\text{Eq}_A \ a \ a'$. An equation is a statement of equality between two expressions. Here, we define $\text{Eq}_{AB}$ as a function from $A + B$ to $A + B$, which is true if the inputs are equal in the chosen form. This is achieved by defining the function case by case, considering the forms $\text{inl} \ a$, $\text{inr} \ b$, and all their combinations.

The definitions are designed to ensure that the equality is preserved under the operations $\text{inl}$ and $\text{inr}$, and that the equality holds if the inputs are of the same form. This approach is common in formal definitions, where we aim to capture the essence of equality in a controlled and precise manner.
Complex Example (Cont.)

- We introduce the formulae expressing that EqA, EqB, EqAB are symmetric:

- Symmetry of EqA is the formula

\[ \forall a, a': A. EqA a a' \rightarrow EqA a' a \]

which translates as follows:

\[
\begin{align*}
\text{SymA} &:: \text{Set} \\
&= (a, a' :: A) \rightarrow EqA a a' \rightarrow EqA a' a
\end{align*}
\]

The others are similar:

\[
\begin{align*}
\text{SymB} &:: \text{Set} \\
&= (b, b' :: B) \rightarrow EqB b b' \rightarrow EqB b' b \\
\text{SymAB} &:: \text{Set} \\
&= (ab, ab' :: A + B) \rightarrow EqAB ab ab' \rightarrow EqAB ab' ab
\end{align*}
\]

Complex Example

- What we want to show is that SymA and SymB implies SymAB.

- So we need to solve

\[
\begin{align*}
\text{symAB} &:: \text{SymA} \rightarrow \text{SymB} \rightarrow \text{SymAB} \\
&= \{! !\}
\end{align*}
\]

- As pointed out before, it is equivalent and more convenient to define symAB as follows:

\[
\begin{align*}
\text{symAB} (\text{symA :: SymA}) \\
&\quad (\text{symB :: SymB}) \\
&:: \text{SymAB} \\
&= \{! !\}
\end{align*}
\]

Formulæ vs. Proofs

- Note that SymA is the statement expressing that EqA is symmetric.

- It is not a proof that EqA is symmetric.

- We can define SymA independently of whether EqA is symmetric or not.

- A proof that EqA is symmetric is an element of SymA, i.e. a term symA s.t.

\[
\text{symA} :: \text{SymA}
\]

- Note that we don't have to show that SymA holds.

- We have to show that if SymA and SymB hold, then SymAB holds.
Complex Example

\[ \text{symAB} \ (\text{symA} :: \text{SymA}) \]
\[ \quad (\text{symB} :: \text{SymB}) \]
\[ \quad :: \text{SymAB} \]
\[ = \{! !\} \]

The type of the goal is SymAB which is
\( (ab, ab' :: A + B) \rightarrow \text{EqAB} \ ab \ ab' \rightarrow \text{EqAB} \ ab' \ ab. \)

An element of this type can be introduced by a \( \lambda \)-term, and using agda-goal-menu “intro” results in the code on the next slide.

Complex Example (Cont.)

In case \( ab = \text{inl} \ a \) and \( ab' = \text{inl} \ a' \) we
    have to show
\[ \text{EqAB} \ ab' \ ab , \text{ which is equal to } \text{EqA} \ a' \ a \]
    and have as assumption
\[ abab' :: \text{EqAB} \ ab \ ab' , \text{ which is equal to } \text{EqA} \ a \ a'. \]
 So we have to derive from \( abab' :: \text{EqA} \ a \ a' \) an
 element of \( \text{EqA} \ a' \ a. \)
 We have
\[ \text{symA} : (a, a' :: A) \rightarrow \text{EqA} \ a \ a' \rightarrow \text{EqA} \ a' \ a . \]
 Therefore we can apply \( \text{symA} \) to \( a, a' \) and \( abab' \).
Complex Example (Cont.)

- (Case \(ab = \text{inl} \ a\), \(ab' = \text{inl} \ a'\))
  - We obtain
    \[
    \text{symA} \ a \ a' \ \text{abab'} : \text{EqA} \ a' \ a
    \]
  - Since \(\text{EqA} \ a' \ a = \text{EqAB} \ ab' \ ab\) we get as well
    \[
    \text{symA} \ a \ a' \ \text{abab'} : \text{EqAB} \ ab' \ ab
    \]
    and can use this to solve our first goal.

Complex Example

\[
\text{symAB} \quad (\text{symA :: SymA})
\]

\[
(\text{symB :: SymB})
\]

\[
:: \quad \text{SymAB}
\]

\[
= \lambda(ab, ab' :: A + B) \to
\]

\[
\lambda(\text{abab'} :: \text{EqAB} \ ab \ ab') \to
case \ ab \ of
\]

\[
(\text{inl} \ a) \quad \to \quad \text{case} \ ab' \ of
\]

\[
(\text{inl} \ a') \quad \to \quad \text{symA} \ a \ a' \ \text{abab'}
\]

\[
(\text{inr} \ b') \quad \to \quad \{! \ !\}
\]

\[
(\text{inr} \ b) \quad \to \quad \text{case} \ ab' \ of
\]

\[
(\text{inl} \ a') \quad \to \quad \{! \ !\}
\]

\[
(\text{inr} \ b') \quad \to \quad \{! \ !\}
\]

Complex Example (Cont.)

- In case \(ab = \text{inr} \ b\) and \(ab' = \text{inr} \ b'\) we can similarly use
  \[
  \text{symB} \ b \ b' \ \text{abab'}
  \]
  in order to solve our goal.

- In case \(ab = \text{inl} \ a\), and \(ab' = \text{inr} \ b\)
  - we have \(\text{EqAB} \ ab \ ab' = \text{False}'\),
  - therefore
    \[
    \text{abab'} : \text{False}'
    \]
    - therefore empty case distinction on \(\text{abab'}\) solves the goal.

Complex Example (Cont.)

- Similarly, in case \(ab = \text{inl} \ a\), and \(ab' = \text{inr} \ b\) we have that
  \[
  \text{abab'} : \text{False}'
  \]
  and again empty case distinction on \(\text{abab'}\) solves the goal.

- The complete solution is on the next slide.
Complex Example

When we made the empty case distinctions, our goal was of type `False'.

Since in those cases `abab : False', we could have solved the goal as well by directly inserting `abab' in those cases.

The next slide shows this alternative solution.

Alternative Solution

Remark on Case Distinction

Case distinction over complex expressions causes problems in Agda.

Example

```agda
Assume we have defined `ProdBool' as the product of two Boolean values:

```ProdBool :: Set = sig
  first :: Bool'
  snd :: Bool'```
Remark on Case Distinction

Assume we want to define a function as follows:

\[
\begin{align*}
f (\text{pair} & :: \text{ProdBool}) \\
(p & :: \text{atom (and \text{pair.first} \text{pair snd}))} \\
\text{:: atom pair.first} \\
= \text{case pair.first of} \\
(\text{tt}) & \rightarrow \{! !\} \\
(\text{ff}) & \rightarrow \{! !\}
\end{align*}
\]

Although in the case distinction
- we know that \text{pair.first} = \text{tt},
- therefore the type of the goal should be \text{atom pair.first} = \text{True'},
- Agda won’t accept to insert \text{true} there.

Workaround

One can work around this problem by defining an auxiliary function, which depends on a variable representing the complex expression.

Then make case distinction on this single variable.

In the example above define:

\[
\begin{align*}
h (a, b & :: \text{Bool'}) \\
(p & :: \text{atom (and a b))} \\
\text{:: atom a} \\
= \text{case a of} \\
(\text{tt}) & \rightarrow \text{true} \\
(\text{ff}) & \rightarrow \text{case p of} \{ \}
\end{align*}
\]

Remark on Case Distinction

The reason is that Agda will use in its reduction mechanism
- only reductions from variables to other expressions,
- but no reductions of complex expressions to other expressions.

It would be very expensive to check reductions for complex expressions:
- This would mean to check whether any subexpression of an expression matches the left side of any of those reductions.
- Checking whether a variable which reduces occurs in an expression is instead a cheap operation.

Workaround

Now one can define the function in question in terms of the auxiliary function:

\[
\begin{align*}
f (\text{pair} & :: \text{ProdBool}) \\
(p & :: \text{atom (and \text{pair.first} \text{pair snd}))} \\
\text{:: atom pair.first} \\
= h \text{pair.first} \text{pair.second}
\end{align*}
\]

In the example h had only as arguments the subexpressions of the complex expression in question.

In general it might depend on other variables which form the context of the complex expression in question.
There seems to be a confusion about the two expressions

\[(x :: A) \rightarrow B \text{ vs. } \lambda(x :: A) \rightarrow s\]

\((x :: A) \rightarrow B\) is the dependent function set.
- It is a set (or a type or a kind or a higher kind).
- Because it is a set, it makes sense to talk about \(r :: ((x :: A) \rightarrow B)\).
- \(r :: C\) makes only sense if \(C\) is a set or a type or a kind or a higher kind.

\(\lambda(x :: A) \rightarrow s\) is a function, which applied to \(x :: A\) returns \(s\).
- \(a :: (\lambda(x :: A) \rightarrow s)\) never makes sense, since \(\lambda(x :: A) \rightarrow s\) is not a set or type or (higher) kind.

Especially, \(\lambda(x :: A) \rightarrow \text{Set}\) is a function which returns for \(x :: A\) the type \(\text{Set}\).

Note that

\[
A \quad (b :: B) \\
:: \quad \text{Set} \\
= \quad d
\]

is an abbreviation for

\[
A :: (b :: B) \rightarrow \text{Set} \\
= \quad \lambda(b :: B) \rightarrow d
\]

So \(A\) defined as such is a function, not a set.
- It does not make sense to talk about \(c :: A\).
- Would be the same as \(c :: (\lambda(b :: B) \rightarrow d)\).

It does make sense to talk about

\[c :: (b :: B) \rightarrow A b\]

Since \((b :: B) \rightarrow A b\) is a set.

In this section we study, how derivations in dependent type theory correspond to derivations in natural deduction. (Omitted 2005)

We will as well introduce constructive logic.

Jump to constructive logic.
Conjunction

- We have seen before that we can identify in type theory conjunction with the non-dependent product.
- With this interpretation, the introduction rule for the product allows to form a proof of \( A \land B \) from a proof of \( A \) and a proof of \( B \):

\[
\frac{p : A}{\langle p, q \rangle : A \land B} \quad (\times\text{-I})
\]

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

Conjunction and Natural Ded.

- In so called natural deduction, one has rules for deriving and eliminating formulas formed using the standard connectives.
- There the rule for introducing proofs of \( A \land B \) is

\[
\frac{A \quad B}{A \land B} \quad (\land\text{-I})
\]

- The type theoretic introduction rule corresponds exactly to this rule.

Example 1

- For instance, assume we want to prove that a function \( \text{sort} \) from lists to lists is a sorting algorithm.
- Then we have to show that for every list \( l \) the application of \( \text{sort} \) to \( l \) is sorted, and has the same elements of \( l \).
- In order to show this, one would assume a list \( l \) and show
  - first that \( \text{sort} \ l \) is sorted,
  - then, that \( \text{sort} \ l \) has the same elements as \( l \)
  - and finally conclude that it fulfils the conjunction of both properties.
- The last operation uses the introduction rule for \( \land \).

Conjunction (Cont.)

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

\[
\begin{align*}
\frac{p : A \land B}{\pi_0(p) : A} \quad (\times\text{-El}_0) & & \frac{p : A \land B}{\pi_1(p) : B} \quad (\times\text{-El}_1)
\end{align*}
\]

- This means that we can derive from \( A \land B \) both \( A \) and \( B \).
- This corresponds to the natural deduction elimination rule for \( \land \):

\[
\begin{align*}
\frac{A \land B}{A} \quad (\land\text{-El}_0) & & \frac{A \land B}{B} \quad (\land\text{-El}_1)
\end{align*}
\]

Omit Example 1

Omit Example 2
Example 2

Assume we have defined a function \( f \), which takes a list of natural numbers \( l \), a proof that \( l \) is sorted, and a natural number \( n \), and returns the Boolean value \( \text{tt} \) or \( \text{ff} \) indicating whether \( n \) is in this list or not.

Assume now a sorting function \( \text{sort} \) from lists of natural numbers to natural numbers, plus a proof that it is a sorting function, i.e. that \( \text{sort} \ l \) is sorted and has the same elements as \( l \) for every list \( l \).

We want to apply \( f \) to \( \text{sort} \ l \) and need therefore a proof that \( \text{sort} \ l \) is sorted.

We have that the conjunction of “\( \text{sort} \ l \) is sorted” and “\( \text{sort} \ l \) has the same elements as \( l \)” holds.

Using the elimination rule for \( \wedge \) one can conclude the desired property, that \( \text{sort} \ l \) is sorted.

Example 3

Assume a proof of \( A \wedge B \).

We want to show \( B \wedge A \).

By \( \wedge \)-elimination we obtain from \( A \wedge B \) that \( B \) holds.

Similarly we conclude that \( A \) holds.

Using \( \wedge \)-introduction we conclude \( B \wedge A \).

In natural deduction, this proof is as follows:

\[
\frac{A \wedge B}{B} (\wedge \text{El}_0) \quad \frac{A \wedge B}{A} (\wedge \text{El}_1) \quad \frac{B}{A \wedge A} (\wedge \text{I})
\]

We have seen in the previous section how to derive this in Agda.

Disjunction

We have seen before that we can identify in type theory disjunction with the disjoint union.

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

\[
\frac{A : \text{Set} \quad B : \text{Set} \quad p : A}{\text{inl} \ A \ B \ p : A + B} (+ \text{I}_{\text{inl}})
\]

\[
\frac{A : \text{Set} \quad B : \text{Set} \quad p : B}{\text{inr} \ A \ B \ p : A + B} (+ \text{I}_{\text{inr}})
\]

Disjunction (Cont.)

Omitting the premises \( A, B : \text{Set} \) and omitting them as arguments of \( \text{inl} \) and \( \text{inr} \) (which is needed only for type checking purposes in the presence of the identity type – this type is not treated in this module) we get:

\[
\frac{A : \text{Set} \quad B : \text{Set} \quad p : A}{\text{inl} \ p : A + B} (+ \text{I}_{\text{inl}})
\]

\[
\frac{A : \text{Set} \quad B : \text{Set} \quad p : B}{\text{inr} \ p : A + B} (+ \text{I}_{\text{inr}})
\]
Disjunction (Cont.)

- This means that we can derive \( A \lor B \) from \( A \) and from \( B \).
- This is what is expressed by the natural deduction introduction rules for \( \lor \):

\[
\frac{A}{A \lor B} (\lor \text{-I}_{\text{inl}}) \quad \frac{B}{A \lor B} (\lor \text{-I}_{\text{inr}})
\]

Omit Example 1

Example 1

- Assume we want to show that every prime number is equal to 2 or odd.
- In order to show this one assumes a prime number.
  - If it is 2, it is trivially equal to 2.
  - Using the introduction rule for \( \lor \) one concludes that it is equal to 2 or odd.
- Otherwise, one argues (using some proof) that it is odd.
  - Using the introduction rule for \( \lor \) one concludes again that it is equal to 2 or odd.

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 \( B \):

\[
\begin{align*}
A & : \text{Set} \\
B & : \text{Set} \\
C & : (A \lor B) \rightarrow \text{Set} \\
sl & : (a : A) \rightarrow C \ (\text{inl } A \ B \ a) \\
sr & : (b : B) \rightarrow C \ (\text{inr } A \ B \ b) \\
d & : A \lor B \\
\text{Case}+ \ A \ B \ C \ sl \ sr \ d & : C \ d \ (+-\text{El})
\end{align*}
\]

Disjunction (Cont.)

- Omitting the dependency of \( C \) on \( A \lor B \), the premises \( A, B \) and \( C \), and the arguments \( A, B \) and \( C \), we get:

\[
\frac{d : A \lor B \quad sl : A \rightarrow C \quad sr : B \rightarrow C}{\text{Case}+ \ sl \ sr \ d : C \ (+-\text{El})}
\]

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

This is what is expressed by the natural deduction elimination rules for $\lor$:

$$
\frac{A \lor B \quad A \vdash C \quad B \vdash C}{C} \quad ($\lor$-El)
$$

In the above rule we have written $A \vdash C$

for

from assumption $A$ we can derive $C$.

This is written sometimes in the following form

$$
\frac{A}{C}
$$

Note that in natural deduction, from the premise $A \vdash C$ we obtain $A \rightarrow C$, which is the premise used in the corresponding rule in dependent type theory.

Example 2

Assume we want to show that every prime number is equal to 2, equal to 3, or $\geq 5$.

We want to make use of the proof above that every prime number is equal to 2 or odd.

We assume a prime number.

- We know that it is equal to 2 or odd.
- In case it is equal to 2 we conclude that it is equal to 2, equal to 3, or $\geq 5$.
- In case it is odd, we conclude using the fact that it is prime and 1 is not prime, that it is equal to 3 or $\geq 5$. Therefore it is equal to 2, equal to 3, or $\geq 5$.
- Now from the elimination rule for $\lor$ we conclude that the prime number chosen is equal to 2, equal to 3, or $\geq 5$.

Example 3

Assume a proof of $A \lor B$.

We want to show $B \lor A$.

- We have $A \lor B$.
- From assumption $A$ we obtain $A$ and therefore by $\lor$-introduction $B \lor A$.
- From assumption $B$ we obtain $B$ and therefore by $\lor$-introduction $B \lor A$.
- By $\lor$-elimination we obtain from these three premises $B \lor A$ without any premises.

Omit Example 2
Example 3 (Cont.)

In natural deduction, this proof is as follows (we write $A_1, \ldots, A_n \vdash B$ for $B$ follows under assumptions $A_1, \ldots, A_n$):

$$
\begin{align*}
A \lor B & \quad \frac{A \vdash A}{A \lor B} (\lor\text{-inr}) \\
& \quad \frac{B \vdash B}{B \lor A} (\lor\text{-inr}) \\
& \quad \frac{B \lor A}{B \lor A} (\lor\text{-El})
\end{align*}
$$

We have seen in the previous section how to derive this in Agda.

Implication

We have seen before that we can identify in type theory implication with the non-dependent function type. In order to distinguish between the function type and the logical implication we will write in this subsection $\supset$ instead of $\rightarrow$ for logical implication.

Implication (Cont.)

With this identification of logical implication and the function type, 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 $A$:

$$
\frac{p : A \Rightarrow q : B}{\lambda p^A.q : A \supset B} (\supset\text{-I})
$$

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 $p:A$.
Example

We extend the proof that, if we have \( A \lor B \), then we have \( B \lor A \), to a proof of
\[
(A \lor B) \supset (B \lor A)
\]
The previous proof can be easily transformed into a proof of \( A \lor B \vdash B \lor A \).

By \( \supset \)-introduction, it follows \( (A \lor B) \supset (B \lor A) \).

Example

The complete proof in natural deduction is as follows:

\[
\begin{array}{c}
A \lor B \vdash A \lor B \\
\hline
A \lor B \vdash A \\
A \lor B \vdash B \\
\hline
A \lor B \vdash A \lor B \\
\hline
A \lor B \vdash B \lor A \\
\hline
A \lor B \vdash (B \lor A) \\
\hline
(A \lor B) \supset (B \lor A) \\
\end{array}
\]

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 \( B \):

\[
\frac{p : A \supset B \quad q : A}{p \quad q : B \quad (\supset \text{-El})}
\]

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

This is what is expressed by the natural deduction elimination rule for \( \supset \):

\[
\frac{A \supset B \quad A}{B \quad (\supset \text{-El})}
\]

Example

Assume we want to show \( A \supset (A \supset B) \supset B \).

We can show this as follows:

- From assumptions \( A \) and \( A \supset B \) we can conclude \( A \supset B \).
- From assumptions \( A \) and \( A \supset B \) we can conclude as well \( B \).
- Using the elimination rule for \( \supset \), we conclude that under the same assumptions we get \( B \).
- Using the introduction rule for \( \supset \) we conclude from assumption \( A \) that \( (A \supset B) \supset B \) holds.
- Using again the introduction rule for \( \supset \) we conclude that \( A \supset (A \supset B) \supset B \) holds without any assumptions.
Example

A proof in natural deduction is as follows:

\[
\begin{align*}
A, A \supset B \vdash A \supset B & \quad A, A \supset B \vdash A \quad (\supset \text{El}) \\
A, A \supset B & \vdash B \quad (\supset \text{-I}) \\
A \vdash (A \supset B) \supset B & \quad (\supset \text{-I}) \\
A \supset (A \supset B) \supset B & \quad (\supset \text{-I})
\end{align*}
\]

Universal Quantification

We have seen before that we can identify in type theory universal quantification with the dependent function type.

With this identification, the introduction rule for the dependent function type 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 natural deduction introduction rule for \( \forall \):

\[
\frac{x : A \vdash B}{\forall x : A . B} \quad (\forall \text{-I})
\]

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 \( x:A \).

Universal Quantification (Cont.)

Note that we have written

\[
x : A \vdash B
\]

for

we can derive \( B \) from variable \( x : A \).

This is usually not mentioned as such in natural deduction.

We prefer this notation, since it

- makes the variable \( x \) explicit,

- and allows to deal with more complex types \( A \).
Universal Quantification (Cont.)

- The conclusion of the introduction rule will no longer depend on free variables $x$.
- This is made explicit by mentioning free variables $x : A$ in our notation.
- In type theory this corresponds to the fact that $x : A$ does no longer occur in the context of the conclusion.

Example

- Assume one wants to show that for every natural number $n$, $n + 0 == n$.
- In order to show this one assumes a natural number $n$ and shows then that $n + 0 == n$.
- Then using the introduction rule for $\forall$ one concludes $\forall n : N, n + 0 == n$.
- In natural deduction, this proof is as follows (where the prove of $n + 0 == n$ is not carried out):

$$
\frac{n + 0 == n}{\forall n : N, n + 0 == n} (\forall\text{-I})
$$

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 $B[x := a]$:

$$
\frac{p : (\forall x : A.B) \quad a : A}{p a : B[x := a]} (\to\text{-El})
$$

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

This is what is expressed by the natural deduction elimination rule for $\forall$.

- For the simple languages used in natural deduction, there is no need to derive that $a : A$; in more complex type theories we have to carry out this derivation.

$$
\frac{\forall x : A.B \quad a : A}{B[x := a]} (\forall\text{-El})
$$
Example

Assume a proof of $\forall n : \mathbb{N}. 0 + n == n$.

We want to conclude that $\forall n, m : \mathbb{N}. 0 + (n + m) == (n + m)$.

This can be done as follows:
- One assumes $n, m : \mathbb{N}$.
- Then one can conclude $n + m : \mathbb{N}$.
- Using $\forall n : \mathbb{N}. 0 + n == n$ and the elimination rule for $\forall$ one concludes $0 + (n + m) == (n + m)$ under assumption $n, m : \mathbb{N}$.
- Now using the introduction rule for $\forall$ twice it follows $\forall n, m : \mathbb{N}. 0 + (n + m) == (n + m)$.

Existential Quantification

We have seen before that we can identify in type theory existential quantification with the dependent product.

With this identification, the introduction rule for the dependent product allows to form a proof of $\exists x : A. B$ from an element $a : A$ and a proof $p : B[x := a]$:

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

This is what is expressed by the natural deduction introduction rule for $\exists$:

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

Example

In natural deduction, this proof is written as follows:

$$\frac{\forall n : \mathbb{N}. 0 + n == n \quad n : \mathbb{N}, m : \mathbb{N} \vdash (n + m) == (n + m)}{\mathbb{N} \vdash \forall m : \mathbb{N}. 0 + (n + m) == (n + m)}$$ (\forall-E +)

Example

Assume we want to show $\forall n : \mathbb{N}. \exists m : \mathbb{N}. m > n$.

- In order to prove this one assumes first $n : \mathbb{N}$.
- Then one concludes $S n : \mathbb{N}$ and $S n > n$.
- Using the introduction rule for $\exists$ one concludes $\exists m : \mathbb{N}. m > n$ under the assumption $n : \mathbb{N}$.
- Using the introduction rule for $\forall$ one concludes $\forall n : \mathbb{N}. \exists m : \mathbb{N}. m > n$. 
Example

In natural deduction, this proof reads as follows:

\[
\begin{align*}
  n & : N \vdash n : N \quad (N-IS) \\
  n & : N \vdash S\ n : N \\
  n & : N \vdash \exists m : N.\ m > n \quad (\exists-I) \\
  \forall n : N.\ \exists m : N.\ m > n \quad (\forall-I)
\end{align*}
\]

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

\[
\begin{align*}
  \exists x : A.B & \quad x : A, B \vdash C \\
  & \quad \frac{}{C} \quad (\exists-El)
\end{align*}
\]

where \(C\) does not depend on \(x : A\) and \(B\).

Here \(x : A, B \vdash C\) means that from \(x : A\) and assumption \(B\) we can derive \(C\).

As in the introduction rule for natural deduction, \(x : A\) is usually not mentioned explicitly, since the type structure there is very simple.

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 \(y:B\).

Example

Assume we have shown

\(\forall n : N.\exists m : N.m > n \wedge \text{Prime}(m)\).

We want to show that for all \(n\) there exist two primes above it, i.e.

\(\forall n : N.\exists m : N.\exists k : N.m > k \wedge k > n \wedge \text{Prime}(m) \wedge \text{Prime}(k)\).

We can derive this as follows:

Assume \(n : N\).
We have \(\exists m : N.m > n \wedge \text{Prime}(m)\).
So assume \(m : N\) and \(m > n \wedge \text{Prime}(m)\).
We have as well \(\exists k : N.k > m \wedge \text{Prime}(k)\).
So assume \(k : N\) and \(k > m \wedge \text{Prime}(k)\).
Example

Then we can conclude

\[ m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \]

and therefore as well

\[ \exists m, k : \forall. m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \]

Now by \( \exists \)-elimination twice follows

\[ n : N \vdash \exists m, k : N. m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \]

without assuming \( m, k \) as above.

By \( \forall \)-introduction follows

\[ \forall n : N. \exists m, k : N. m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \]

Example

First step: Under the global assumption

\[ n : N, m : N, m > n \land \text{Prime}(m), k : N, k > m \land \text{Prime}(k) \]

we prove the following

\[
\frac{\quad k : N \quad m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) \quad}{\exists k : N. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)} \quad (\exists-1)
\]

\[
\frac{\forall m, k : N. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)}{\exists m, k : N. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)} \quad (\exists-1)
\]

So we have shown

\[ n : N, m : N, m > n \land \text{Prime}(m), k : N, k > m \land \text{Prime}(k) \vdash \exists m, k : N. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) \]

Example

Second step: Under the assumption

\[ n : N, m : N, m > n \land \text{Prime}(m) \]

we can conclude

\[ \exists k : N. k > m \land \text{Prime}(k) \]

and then conclude by \( \exists \)-elimination and Step 1

\[
\frac{\exists k : N. k > m \land \text{Prime}(k) \quad }{\exists m, k : N. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)} \quad (\exists-1)
\]
Example

Third step: Again we can conclude

\[ n : N \vdash \exists m : N. m > n \land \text{Prime}(m) \]

and then conclude by \( \exists \)-elimination and Step 2

\[
\begin{align*}
\frac{n : N \vdash \exists m : N. m > n \land \text{Prime}(m) \quad n, m : N \vdash m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)}{\forall n : N. \exists m, k : N. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)} \tag{3-1}
\end{align*}
\]

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

\[
\forall x : \text{Turing\_Machine}.(x \text{ halts} \lor \lnot(x \text{ halts}))
\]

Construct. (or Intuit.) 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 of type \( A \to B \), 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 \( x \).

Turing Machines

A Turing machine (in short TM) is a program language which is according to Church’s thesis universal:
- Every computable function can be computed by a TM.
- TMs can have one input string, no interaction, and have as output one output string.
- Both these strings are usually interpreted as natural numbers.
- To run a TM with no input means to run it with the empty input string.
**Turing Complete Languages**

- Any programming language, which can simulate a TM, shares this property and is called **Turing complete**.
- Most standard programming languages, e.g. Java, Pascal, C, C++ are **Turing complete**.
- **Agda**, restricted to termination checked programs, is **not Turing complete**.
- No (decidable) language, which allows to write terminating programs only, can be Turing complete.

---

**Turing Halting Problem**

- The **Turing halting problem** is the question, whether a TM (with no inputs) terminates.
- One can introduce a predicate \( \text{halts} \, x \) depending on a TM \( x \) (which can be represented as a string, as a natural number, or as a specific data type) expressing that “\( \text{TM} \, x \) holds, if given no inputs”.
- Therefore the Turing halting problem is the question whether we can decide

\[
\text{halts} \, x \lor \neg \text{halts} \, x
\]

- It is known that the Turing halting problem is undecidable:
  - We cannot decide in a computable way for every \( x \) the Turing halting problem for \( x \).

---

**Unprovability in Type Theory**

- Similarly we cannot decide whether a Java program with no input and no interaction terminates or not.
- Because of the undecidability of the Turing halting problem, the following formula is unprovable in Martin-Löf Type Theory and as well in Agda:

\[
\forall x : \text{TM}. \text{halts} \, x \lor \neg \text{halts} \, x
\]

- Here \( \text{TM} \) is a data type which allows to encode all TM in a standard way.
- If we could prove it, we could get a function, which determines for \( x : \text{TM} \) whether \( \text{halts} \, x \) or not.
- But such a function needs to be computable, and such a computable function doesn’t exist.

---

**Constructive Logic (Cont.)**

- In classical logic we can **prove the above**, since we can derive \( A \lor \neg A \) (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 \( A \lor A \) and \( \neg A \lor A \) don’t hold for all formulae \( A \).
- As usual in natural deduction, we write in the following \( \bot \) for False.
- Jump over remaining slides
Constructive Logic (Cont.)

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

- If we take decidable atomic formulae only and replace $\exists x : A.B$ and $A \lor B$ by the above formulae, then all formulae provable in classical logic are derivable.
  - This requires $\neg A \supset A$, which can be shown for all formulae built from decidable atomic formulae using $\neg, \supset, \land, \forall$.
  - The formula $A \lor \neg A$ translates into $\neg (\neg A \land \neg \neg A)$, which trivially holds, since $\neg A$ and $\neg \neg A$ implies $\bot$.

- In this sense, type theory contains classical logic, but is richer, since it has as well so called strong disjunction and existential quantification.

Weak disjunct. & Existent. Quant.

- Note that weak disjunction and existential quantification don’t have the same constructive content as strong disjunction and existential quantification.
  - From $p : \neg (\neg A \land \neg B)$ we cannot in general decide whether $A$ or $B$ holds.
  - From $p : \neg \forall x : A.\neg B$ we cannot in general extract an $a : A$ s.t. $B[x := a]$ holds.

Proof (using classical logic) of $\exists x : A.B \leftrightarrow (\neg \forall x : A.\neg B)$:

- We have classically:
  - $\neg A \supset A$:
    - If $A$ is true, then $\neg A \supset A$ holds.
    - If $A$ is false, then $\neg A$ is false, therefore $\neg \neg A \supset A$ holds.

Constructive Logic (Cont.)

- Weak disjunction and existential quantification is expressed by the formulae $\neg (\neg A \land \neg B)$ 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.
  - Remark: One can always obtain classical logic in Agda for arbitrary formulae by postulating tertium non datur for the formulae for which one needs it, i.e. writing
  - postulate $p :: A \lor \neg A$
  - Jump over the following proofs.
Constructive Logic (Cont.)

We show intuitionistically
\[ (\lnot \exists x : A.B) \leftrightarrow (\forall x : A. \lnot B) \] .

Assume \( \lnot \exists x : A.B, x : A \) and show \( \lnot B \).
If we had \( B \), then we had \( \exists x : A.B \), contradicting
\( \lnot \exists x : A.B \). Therefore \( \lnot B \).

Assume \( \forall x : A. \lnot B \). Show \( \lnot \exists x : A.B \):
Assume \( \exists x : A.B \). Assume \( x \) s.t. \( B \) holds.
By \( \forall x : A. \lnot B \) we get \( \lnot B \), therefore a contradiction.

Now it follows (classically):
\[ (\exists x : A.B) \leftrightarrow (\lnot \exists x : A.B) \leftrightarrow (\forall x : A. \lnot B) \]

Class. Logic for \( \exists, \lor \)-free Formulae

We show that for formulas \( A \) built from \( \lnot, \lor, \land, \forall \) and
decidable prime formulae we have
\[ \lnot A \lor A . \]

The formula \( \lnot A \lor A \) is called stability for \( A \).

This is done by induction over the buildup of these
formulae.

Case \( A \equiv \text{atom} \ c \).

We make case distinction on \( c \).
If \( c = \text{tt} \), then we have \( A \equiv \text{True} \), \( A \) is provable,
therefore as well \( \lnot A \lor A \).
If \( c = \text{ff} \), then we have \( A \equiv \text{False} \equiv \bot \).
Assume \( \lnot \lnot A \equiv (\bot \lor \bot) \lor \bot \).
\( \bot \lor \bot \) is provable.
Therefore we obtain \( \bot \), which is \( A \).
So we have
\[ \lnot A \lor A \]
and obtain
\[ \lnot \lnot A \lor A . \]
Case \( A \equiv B \supset C \), and assume we have already shown stability for \( B \) and \( C \).

We have to show that from \( \neg A \) we obtain \( A \), which is \( B \supset C \).

So assume \( \neg A \), \( B \) and show \( C \).

We show \( \neg C \), then by stability of \( C \) we obtain \( C \).

\( \neg C \equiv \neg C \supset \bot \).

Therefore assume \( \neg C \) and show \( \bot \).

We show \( \neg A \) which is \( A \supset \bot \).

So assume \( A \) and show \( \bot \). \( A \equiv B \supset C \), therefore by \( B \) we get \( C \), and by \( \neg C \) therefore \( \bot \).

By \( \neg A \), which is \( \neg A \supset \bot \), we get therefore \( \bot \), which completes the proof for this case.

Case \( A \equiv B \land C \), and assume we have already shown stability for \( B \) and \( C \).

Assume \( \neg A \) and show \( A \).

We show \( \neg B \), which implies by the stability of \( B \) that \( B \) holds.

Since \( \neg B \equiv \neg B \supset \bot \), we assume \( \neg B \) and have to show \( \bot \).

We show \( \neg A \), i.e. show that \( A \) implies \( \bot \).

\* Assume \( A \), which is \( B \land C \). Then we get \( B \), and by \( \neg B \) therefore \( \bot \).

By \( \neg A \) we obtain \( \bot \).

Therefore we have shown \( B \).

A similar proof shows \( C \), and therefore we get \( B \land C \), i.e. \( A \).

Case \( A \equiv \forall x : B.C \), and assume we have already shown stability for \( C \).

Assume \( \neg A \) and show \( A \).

So assume \( x : B \), and show \( C \).

We show \( \neg C \), which by the stability of \( C \) implies \( C \).

So assume \( \neg C \) and show \( \bot \).

We show \( \neg A \).

Assume \( A \), which is \( \forall x : B.C \).

Then we obtain \( C \), and by \( \neg C \) therefore \( \bot \).

By \( \neg A \) we therefore get \( \bot \), and are done.
(h) The Set of Natural Numbers

- The set \( N \) is the type theoretic representation of the set \( N := \{0, 1, 2, \ldots \} \).
- \( N \) 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
  \[
  \begin{align*}
  N : & \text{Set} \quad (N\text{-F}) \\
  0 : & N \quad (N\text{-I}_0) \\
  n : & N \quad (N\text{-I}_S) \\
  S n : & N \quad (N\text{-I}_S)
  \end{align*}
  \]

Primitive Recursion

- Primitive Recursion expresses:
  - Assume we have
    \[
    \begin{align*}
    a : & N, \\
    \text{and, if } n : N, x : N \text{ then } g n x : N.
    \end{align*}
    \]
  - Then we can define \( f : N \to N \), s.t.
    \[
    \begin{align*}
    f 0 &= a, \\
    f (S n) &= g n (f n).
    \end{align*}
    \]

Primitive Recursion (Cont.)

- The \textbf{computation of} \( f n \) 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 \( f n' \).
    - Now \( f n \) reduces to \( g n' (f n') \).
    - \( g n' (f n') \) can be computed, since we know how to compute
      \[
      \begin{align*}
      g & \\
      f n' & \text{.}
      \end{align*}
      \]
The function \( f : \mathbb{N} \to \mathbb{N} \) with \( f \ n = 2 \cdot n \) can be defined primitive recursively by:
\[
\begin{align*}
  f \ 0 &= 0, \\
  f \ (S \ n) &= S \ (S \ (f \ n)).
\end{align*}
\]

Therefore take in the definition above:
\[
\begin{align*}
  a &= 0, \\
  g \ n \ x &= S \ (S \ x).
\end{align*}
\]

**Generalised Primitive Recursion**

We can generalise primitive recursion as follows:

1. First we can replace the range of \( f \) by an arbitrary set \( C \)
2. i.e. we allow for any set \( C \)
   \[
   f : \mathbb{N} \to C
   \]
3. Further, \( C \) can now depend on \( \mathbb{N} \).
4. We obtain the following set of rules:

**Rules for the Natural Numbers**

**Formation Rule**
\[
\begin{align*}
  N : \text{Set} \quad (N-F)
\end{align*}
\]

**Introduction Rules**
\[
\begin{align*}
  0 : N \quad (N-I_0) \\
  n : N \quad (N-I_S)
\end{align*}
\]

**Elimination Rule**
\[
\begin{align*}
  C : \mathbb{N} \to \text{Set} \\
  a : C \ 0 \\
  f : (x : \mathbb{N}) \to C \ a \ f\ n : C \ n \quad (N-E_1)
\end{align*}
\]

**Equality Rules**
\[
\begin{align*}
  P \ C \ a \ f \ 0 &= a \quad (N-Eq_0) \\
  P \ C \ a \ f \ (S \ n) &= f \ n \ (P \ C \ a \ f \ n) \quad (N-Eq_S)
\end{align*}
\]

Additionally we have the **Equality versions** of the formation-, introduction- and elimination-rules.
Rules for the Natural Numbers

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

\[ \lambda n : N. \ g \ n : (n : N) \to C \ n . \]

The equality rules read:

\[ g \ 0 = a \]
\[ g \ (S \ n) = f \ n \ (g \ n) \]

Logical Framework Rules for N

The more compact notation is:

- \( N : \text{Set} \),
- \( 0 : N \),
- \( S : N \to N \),
- \( P : (C : N \to \text{Set}) \to C \ 0 \)
  \[ \to ((x : N) \to C \ x \to C \ (S \ x)) \]
  \[ \to (n : N) \]
  \[ \to C \ n \ . \]

The same equality rules as before.

Natural Numbers in Agda

\( N \) is defined using \textbf{data}:

\[ \text{data } N = Z \mid S(n :: N) \]

(We cannot use 0 for a constructor, since this denotes the builtin native natural number 0 in Agda).

Therefore we have

\[ Z :: N \]
\[ S :: N \to N \]

Elimination Rules for N in Agda

Elimination is represented in Agda as before via case distinction.

Assume we want to define

\[ f \ (n :: N) :: A \]
\[ = \{! \ !\} \]

\( A \) possibly depending on \( n \),

Then we can type into the goal \( n \) and use the menu agda-case.
Elimination Rules for N in Agda

- We get

\[ f \ (n :: N) :: A = \text{case } n \text{ of} \]
\[ (Z) \rightarrow \{ \} \]
\[ (S \ n') \rightarrow \{ ! ! \} \]

Elimination Rules for N in Agda

- If check-termination succeeds, the definition should be correct.
- (The lecturer hasn’t checked the algorithm).
- However, if check-termination fails, the definition might still be correct. Jump over Limitations of Termination Checker.

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 “check-termination”, we might obtain an error-message.

If we
- do not make use of \( f \) in the case \( n=Z \) and
- only use of \( f \ n' \) in case \( n = S \ n' \).

then check-termination succeeds.

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 check-termination accepts it:

\[
\text{fib} \ (n :: N) :: N = \text{case } n \text{ of} \]
\[ (Z) \rightarrow \text{one} \]
\[ (S \ n') \rightarrow \text{case } n' \text{ of} \]
\[ (Z) \rightarrow \text{one} \]
\[ (S \ n'') \rightarrow \text{fib } n' + \text{fib } n'' \]
Assume we define the **predecessor function**

```haskell
def pred (n :: N)
  :: N
  = case n of
    (Z) -> Z
    (S n') -> n'
```

i.e.

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

Then the function

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

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

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

Unfortunately, Agda does currently not deal with **simultaneous recursion**
- i.e. the situation, where we decrease in one case w.r.t. one variable, in another case w.r.t. another variable.
- In order to deal with this situation, one has to **rearrange proofs**.
- On next slide there is an example of a proof which results in non-termination, although each recursive call descends.
- Refers to a definition of $(\langle\rangle) :: (n, m :: N) \rightarrow \text{Set}$ which will be introduced below.
Example

\[ \text{mono} \ (n, k, m :: \mathbb{N}) (p :: n < k) :: (n + m) < (k + m) \]
\[ = \ \text{case } n \ of \]
\[ \quad (\mathsf{Z}) \quad \rightarrow \text{case } k \ of \]
\[ \quad \quad (\mathsf{Z}) \quad \rightarrow \text{case } p \ of \ \{ \} \]
\[ \quad \quad (S \ k') \quad \rightarrow \text{case } m \ of \]
\[ \quad \quad \quad (\mathsf{Z}) \quad \rightarrow \text{true} \]
\[ \quad \quad \quad (S \ m') \quad \rightarrow \text{mono} \ n \ k \ m' \ p \]
\[ (S \ n') \quad \rightarrow \text{case } k \ of \]
\[ \quad (\mathsf{Z}) \quad \rightarrow \text{case } p \ of \ \{ \} \]
\[ \quad (S \ k') \quad \rightarrow \text{case } m \ of \]
\[ \quad \quad (\mathsf{Z}) \quad \rightarrow \text{mono} \ n' \ k' \ m \ p \]
\[ \quad (S \ m') \quad \rightarrow \text{mono} \ n \ k \ m' \ p \]

Version accepted by Agda

- The following version will be accepted by the termination checker:
  - (this version corresponds exactly to induction on \( m \))

\[ \text{mono} \ (n, k, m :: \mathbb{N}) (p :: n < k) :: (n + m) < (k + m) \]
\[ = \ \text{case } m \ of \]
\[ \quad (\mathsf{Z}) \quad \rightarrow \ p \]
\[ \quad (S \ m') \quad \rightarrow \text{mono} \ n \ k \ m' \ p \]

Amendment of Non-Termin. Version

- If one cannot reduce a non-terminating version directly in one with only one descend, one can use auxiliary lemmata instead.
- For instance in the previous non-terminating version, if one doesn’t observe the previous much better solution, one can
  - replace the first reference to mono by a reference to a lemma.
  - (this change is not really necessary, since only the second reference is responsible for rejection by the termination checker)
  - and observe that the second reference can be replaced by \( p \).

\[ \text{lemma} \ (m, k :: \mathbb{N}) \]
\[ :: \ Z + m < S \ k + m \]
\[ = \ \text{case } m \ of \]
\[ \quad (\mathsf{Z}) \quad \rightarrow \text{true} \]
\[ \quad (S \ m') \quad \rightarrow \text{lemma} \ m' \ k \]
Example: Addition

Definition of \( + \) in Agda:

\[
(+) \quad (n, m : \mathbb{N}) \\
\quad : \quad \mathbb{N} \\
\quad = \quad \text{case } m \text{ of} \\
\quad \quad (Z) \quad \rightarrow \quad n \\
\quad \quad (S \ m') \quad \rightarrow \quad S \ (n + m')
\]

The definition expresses:

\[
\begin{align*}
n + 0 &= n \\
n + (m' + 1) &= (n + m') + 1
\end{align*}
\]

Example: Multiplication

Definition

\[
(*) \quad (n, m : \mathbb{N}) \\
\quad : \quad \mathbb{N} \\
\quad = \quad \text{case } m \text{ of} \\
\quad \quad (Z) \quad \rightarrow \quad Z \\
\quad \quad (S \ m') \quad \rightarrow \quad n * m' + n
\]

The definition expresses:

\[
\begin{align*}
n \cdot 0 &= 0 \\
n \cdot (m' + 1) &= (n \cdot m') + n
\end{align*}
\]
Example: Multiplication (Cont.)

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

Equality on N

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

\[
\begin{align*}
(Z == Z) &= \text{True}' \\
(Z == S n) &= (S n == Z) = \text{False}' \\
(S n == S m) &= (n == m).
\end{align*}
\]

Equality on N (Cont.)

From this one can now derive a definition in Agda:

\[
(==) \quad (n, m :: N) :: \text{Set} \\
\begin{align*}
&= \text{case } n \text{ of} \\
&(Z) \quad \rightarrow \text{case } m \text{ of} \\
&(Z) \quad \rightarrow \text{True}' \\
&(S m') \quad \rightarrow \text{False}' \\
&(S n') \quad \rightarrow \text{case } m \text{ of} \\
&(Z) \quad \rightarrow \text{False}' \\
&(S m') \quad \rightarrow (n' == m')
\end{align*}
\]

Alternatively, one could have defined first a Boolean valued equality

\[
\text{EqNBool} : N \rightarrow N \rightarrow \text{Bool}
\]

on \(N\) and then defined

\[
n == m = \text{atom(EqNBool } n m)\ .
\]
Reflexivity of ==

- Reflexivity of == is the formula:
  \[ \forall n : N. n == n \]
- Type theoretically this means that we have to define a function \( \text{refl} \):
  \[
  \text{refl} \ (n : N) \\
  :: \ n == n \\
  = \ \{! !\}
  \]

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:
  \[ \forall n, m : N. n == m \rightarrow m == n \]
- Type theoretically this means that we have to define a function \( \text{sym} \):
  \[
  \text{sym} \ (n, m : N) \\
  (p :: n == m) \\
  :: \ m == n \\
  = \ \{! !\}
  \]
Symmetry of == (Cont.)

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

\[
\begin{align*}
\text{sym} &\quad (n, m : N) \\
&\quad (p :: n == m) \\
&\quad :: \quad m == n \\
&\quad = \quad \text{case } n \text{ of} \\
&\quad \quad (Z) \quad \to \quad \text{case } m \text{ of} \\
&\quad \quad \quad (Z) \quad \to \quad \{! \} \\
&\quad \quad \quad (S m') \quad \to \quad \{! \} \\
&\quad (S n') \quad \to \quad \text{case } m \text{ of} \\
&\quad \quad (Z) \quad \to \quad \{! \} \\
&\quad \quad (S m') \quad \to \quad \{! \}
\end{align*}
\]

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 **sym n’ can be defined before sym n**.

  - This definition will be **accepted by check-termination**.

Symmetry of == (Cont.)

- The first goal can be solved by using **true** (since \((Z == Z) = \text{True}')\). 

- For the second goal we know \( p \) is an element of \( Z == S m' \) which is \( \text{False}' \).

  - Therefore if we make **case distinction on p** we get

    \[
    \text{case } p \text{ of } \{ \\
    \}
    \]

    and have solved the second goal.

- Similarly the third goal can be solved.

Example: < on N

- The following introduces < on N:

\[
\begin{align*}
(\langle) &\quad (n, m :: N) \\
&\quad :: \quad \text{Set} \\
&\quad = \quad \text{case } m \text{ of} \\
&\quad \quad (Z) \quad \to \quad \text{False} \\
&\quad \quad (S m') \quad \to \quad \text{case } n \text{ of} \\
&\quad \quad \quad (Z) \quad \to \quad \text{True} \\
&\quad \quad \quad (S n') \quad \to \quad n' < m'
\end{align*}
\]
**Example: Tuples of Length n**

- We define tuples (or vectors) of length \( n \) in Agda.
- Define first

\[
\begin{align*}
\text{data Nil} & \quad = \quad \text{nil} \\
\text{Cons} \quad (A, B :: \text{Set}) & \quad :: \quad \text{Set} \\
& \quad = \quad \text{data cons}(a :: A)(b :: B)
\end{align*}
\]

**Tuples of Length n**

- Therefore (with the obvious definition of two),

\[
\text{Tuple } A \ n = \underbrace{\text{Cons } A (\text{Cons } A \cdots (\text{Cons } A \text{Nil}) \cdots))}_{n \text{ times}}.
\]

- The elements of \( \text{Tuple } A \ n \) are

\[
\text{cons } a_1 (\text{cons } a_2 \cdots (\text{cons } a_n \text{Nil}) \cdots)
\]

for elements \( a_1, \ldots, a_n \) of \( A \).

- In ordinary mathematical notation, we would write \( \langle a_1, \ldots, a_n \rangle \) for such an element.

- Jump over next slides.

**Remarks on Tuples of Length n**

- In ordinary mathematics, we would define

\[
\begin{align*}
\text{Tuple}(A, 0) & \quad := \quad \{ \} , \\
\text{Tuple}(A, n + 1) & \quad := \quad \{ \langle a_1, \ldots, a_{n+1} \rangle \mid a_1, \ldots, a_{n+1} \in A \} .
\end{align*}
\]
Remarks on Tuples of Length n

If we define

\[
\text{nil} := \langle \rangle, \\
\text{cons } a_1 \langle a_2, \ldots, a_{n+1} \rangle := \langle a_1, \ldots, a_{n+1} \rangle,
\]

then this reads:

\[
\text{Tuple}(A, 0) := \{\text{nil}\}, \\
\text{Tuple}(A, n + 1) := \{\text{cons } a b \mid a \in A \land b \in \text{Tuple}(A, n)\}.
\]

Example: Sum of n-Tuples

Define

\[
\text{NTuple } (n :: N) :: \text{Set} \\
= \text{Tuple } N \ n
\]

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

Remarks on Tuples of Length n

In the type theoretic definition we have **constructors**

- \( \text{nil} :: \text{Tuple } A \ Z \)
- \( \text{cons} @ (\text{Tuple } A \ (S \ n)) :: A \rightarrow \text{Tuple } A \ n \rightarrow \text{Tuple } A \ (S \ n) \).

This is the **type theoretic analogue** of the previous definitions.

Componentwise Sum of n-Tuples

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

Using mathematical notation, this sum for instance as follows:

\[
\langle 2, 3, 4 \rangle + \langle 5, 6, 7 \rangle = \langle 7, 9, 11 \rangle.
\]
Componentwise Sum of n-Tuples

\[
\text{sumNTuple } (n :: N) \\
\quad (avec, bvec :: \text{NTuple } n) \\
\quad :: \text{NTuple } n \\
= \text{case } n \text{ of} \\
\quad (Z) \rightarrow \text{nil} \\
\quad (S n') \rightarrow \\
\quad \text{case } avec \text{ of} \\
\quad \quad (\text{cons } a \text{ avec}') \rightarrow \\
\quad \quad \text{case } bvec \text{ of} \\
\quad \quad \quad (\text{cons } b \text{ bvec}') \rightarrow \\
\quad \quad \quad \text{cons} (a + b) \\
\quad \quad \quad (\text{sumNTuple } n' \text{ avec'} \text{ bvec'})
\]

(i) Lists

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

\[
\text{list } (A :: \text{Set}) \\
\quad :: \text{Set} \\
\quad = \text{data } \text{nil} \\
\quad \quad | \text{cons}(a :: A)(l :: \text{list } A)
\]

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

\[
\begin{align*}
\text{f } (l :: \text{list } A) \\
\quad :: \ C \\
\quad = \text{case } l \text{ of} \\
\text{nil} \rightarrow \{! !\} \\
\text{cons } a \text{ l'} \rightarrow \{! !\}
\end{align*}
\]

and in the second goal we can make use of \( f \text{ l'} \).

Example: Length of a List

\[
\text{length } (l :: \text{list } N) \\
\quad :: \ N \\
\quad = \text{case } l \text{ of} \\
\quad \text{nil} \rightarrow Z \\
\quad \text{cons } a \text{ l'} \rightarrow S (\text{length } l')
\]
Example: sumlist

sumlist \( l \) will compute the sum of the elements of list \( l \).

```latex
\begin{align*}
\text{sumlist} & \quad (l :: \text{list } N) \\
& \quad :: \quad N \\
& = \quad \text{case } l \text{ of} \\
& \quad (\text{nil}) \quad \rightarrow \quad Z \\
& \quad (\text{cons } n \ l') \quad \rightarrow \quad n + \text{sumlist } l'
\end{align*}
```

Interesting Exercise

Define

\[
\text{append} : (A : \text{Set}) \rightarrow (\text{list } A) \rightarrow (\text{list } A) \rightarrow \text{list } A,
\]

s.t. append \( A \ l \ l' \) 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:

\[
\begin{align*}
\text{append } A \ (\text{cons } a \ (\text{cons } b \ \text{nil})) \ (\text{cons } c \ (\text{cons } d \ \text{nil})) \\
& = \text{cons } a \ (\text{cons } b \ (\text{cons } c \ (\text{cons } d \ \text{nil})))
\end{align*}
\]

(j) Universes

A universe \( U \) is a set, the elements of which are codes for sets.

So we have

\[
\begin{align*}
U & : \text{Set}, \\
T & : U \rightarrow \text{Set} \text{ (the decoding function)}.
\end{align*}
\]

We consider in the following a universe closed under

\[
\begin{align*}
& \text{False}, \text{True}, \text{Bool}, \text{N}, \\
& +, \\
& \Sigma, \\
& \text{the dependent function type}.
\end{align*}
\]

Rules for the Universe

Formation Rule

\[
\begin{align*}
U & : \text{Set} \quad (\text{U-F}) \\
\text{a} & : U \quad (\text{T-F})
\end{align*}
\]

\[
\text{T a} : \text{Set}
\]
Rules for the Universe

Introduction and Equality Rules

\[ \text{False : U (U-I_{\text{False}})} \quad T (\text{False}) = \text{False : Set (T-Eq_{\text{False}})} \]

\[ \text{True : U (U-I_{\text{True}})} \quad T (\text{True}) = \text{True : Set (T-Eq_{\text{True}})} \]

\[ \text{Bool : U (U-I_{\text{Bool}})} \quad T (\text{Bool}) = \text{Bool : Set (T-Eq_{\text{Bool}})} \]

\[ N : U (U-I_N) \quad T (\text{N}) = \text{N : Set (T-Eq_{\text{N}})} \]

Rules for the Universe

Introduction and Equality Rules (Cont.)

\[ a : U \quad b : T \quad a \rightarrow U \quad (U-I_{\tilde{\Pi}}) \]

\[ \tilde{\Pi} a b : U \]

\[ T (\tilde{\Pi} a b) = (x : T a) \rightarrow T (b x) : \text{Set (T-Eq_{\tilde{\Pi}})} \]

Elimination and Equality Rules

- 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.
- We have of course as well the equality versions of the formation-, introduction- and equality 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

$$\overline{\text{atom}} : \text{Bool} \rightarrow \text{U} ,$$
$$\overline{\text{atom}} := \text{CaseBool} (\lambda x . \overline{\text{U}}) \text{True} \text{False} ,$$

$$\overline{\text{atom}} : \text{Bool} \rightarrow \text{Set} ,$$
$$\overline{\text{atom}} : \lambda x . \overline{\text{U}} \rightarrow \text{T} (\overline{\text{atom}} x) ,$$

Then

- $\overline{\text{atom}} tt = \text{True},$
- $\overline{\text{atom}} ff = \text{False}.$

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 $\text{mutual}.$
- Everything in the scope of it is type checked simultaneously.
- Scope determined by indentation.
- It is necessary, since the definition of U refers to that of T, and the definition of T refers to that of U.
- In general mutual allows simultaneous inductive and/or recursive definitions.
- The termination checker can handle certain terminating simultaneous inductive and/or recursive definitions like the universe.
Universes in Agda (Cont.)

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

\[
T \ (u :: U) \\
:: \ Set \\
= \ case \ u \ of \\
\quad (Nhat) \rightarrow N \\
\quad (Falsehat) \rightarrow False \\
\quad (Truehat) \rightarrow True \\
\quad (Boolhat) \rightarrow Bool \\
\quad (Hat) \rightarrow N \\
\quad (Sigmahat \ a \ b) \rightarrow Sigma \ (T \ a) \\
\quad (Pihat \ a \ b) \rightarrow (x :: T \ a) \rightarrow T \ (b \ x)
\]

Meaning of “data”

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

\[
C_i @ A :: (a_{i1} :: A_{i1}) \\
\rightarrow \ldots \\
\rightarrow (a_{in_i} :: A_{in_i}) \\
\rightarrow A
\]

where a constructor always constructs new elements.

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

(k) Algebraic 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.

\[
A :: Set \\
= \ data \ C_1 \ (a_{11} :: A_{11}) \cdots \ (a_{1n_1} :: A_{1n_1}) \\
\mid C_2 \ (a_{21} :: A_{21}) \cdots \ (a_{2n_2} :: A_{2n_2}) \\
\mid \cdots \\
\mid C_m \ (a_{m1} :: A_{m1}) \cdots \ (a_{mn_m} :: A_{mn_m})
\]

Strictly Positive Algebraic 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 :: Set 
= data C \ (f :: A \rightarrow A) 

What is the least set A having a constructor

\[
C @ A :: (f :: A \rightarrow A) \\
\rightarrow A
\]
Strictly Positive Algebraic Types

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.

\[\text{CS}_336/\text{CS}_M36 \text{ (part 2) Interactive Theorem Proving; Lentterm 2005, Sec. 4 (k)}\]
One further Example

The set of binary trees can be defined as follows:

\[
\text{Bintree} :: \text{Set} \\
= \text{data} \ \text{leaf} \\
| \ \text{branch} \ (\text{left} :: \text{Bintree}) \ (\text{right} :: \text{Bintree})
\]

This is a strictly positive algebraic type.

Extensions of Strict. Pos. Alg. Types

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

Example: the even and odd numbers:

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

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

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

We can even allow \(A_{ij} = B_1 \rightarrow A\) or even \(A_{ij} = B_1 \rightarrow \cdots \rightarrow B_l \rightarrow A\), where \(A\) is one of the types introduced simultaneously.

Example (called “Kleene’s O”):

\[
\text{O} :: \text{Set} \\
= \text{data} \ \text{leaf} \\
| \ \text{suc} \ (o :: \text{O}) \\
| \ \text{lim} \ (f :: \text{N} \rightarrow \text{O})
\]

The last definition is unproblematic, since, if we have \(f :: \text{N} \rightarrow \text{O}\) and construct \(\text{lim} @ f\) out of it, adding this new element to \(\text{O}\) doesn’t destroy the reason for adding it to \(\text{O}\).

So again \(\text{O}\) can be “constructed”.

Elimination Rules for data

Functions \(f\) from strictly positive algebraic 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

\[ f :: \text{Bintree} \to A \]

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 \( O \), when defining

\[ g :: O \to A \]

by case-distinction, then the definition of

\[ g (\text{lim@}_\_ f) \]

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