

4. The λ -Calc. with Prod. and Conj.

(4. The λ -Calculus with Products and Conjunction)

- (a) The typed λ -calculus with products.
- (b) Currying. (Omitted 2008).
- (c) The nondependent product in Agda.
- (d) Logic With Conjunction.
- (e) The λ -calculus and term rewriting.
- (f) Finite Sets and Decidable Formulae
- (g) Finite Sets and Decidable Formulae in Agda

(a) The Typed λ -Calc. with Products

- One can expand the set of λ -types and λ -terms as follows:
 - Types are defined as before, but we have additionally:
 - If σ, τ are types, so is $\sigma \times \tau$.

Example (Products)

- Assume we have some extra ground types

$\text{Name} \quad := \quad \text{String}$

$\text{Gender} \quad := \quad \{\text{female}, \text{male}\}$

The exact definition of Gender and String in type theory will be given later (String will be a list of characters).

- Then we can define

$\text{name-with-gender} \quad := \quad \text{String} \times \text{Gender}$

- Then we have $\langle \text{"John"}, \text{male} \rangle : \text{name-with-gender}$.
- If $s : \text{name-with-gender}$, then its first projection is a name.

Example2 (Products)

- Assume we have a type Term of terms, representing functions

$$\text{Int} \rightarrow \text{Int} .$$

- The set of terms Term together with the function, they denote, is given as

$$\text{Term} \times (\text{Int} \rightarrow \text{Int})$$

Products

- The set of typed- λ -terms are defined as before but we have:

- If $s : \sigma, t : \tau$ then $\langle s, t \rangle : \sigma \times \tau$:

$$\frac{\Gamma \Rightarrow s : \sigma \quad \Gamma \Rightarrow t : \tau}{\Gamma \Rightarrow \langle s, t \rangle : \sigma \times \tau} \text{ (Pair)}$$

- If $s : \sigma \times \tau$, then $\pi_0(s) : \sigma$ and $\pi_1(s) : \tau$:

$$\frac{\Gamma \Rightarrow s : \sigma \times \tau}{\Gamma \Rightarrow \pi_0(s) : \sigma} \text{ (Proj}_0\text{)}$$

$$\frac{\Gamma \Rightarrow s : \sigma \times \tau}{\Gamma \Rightarrow \pi_1(s) : \tau} \text{ (Proj}_1\text{)}$$

Example

● We show

$$(\lambda x^{(o \rightarrow o) \times (o \rightarrow o \rightarrow o)}. \pi_0(x)) \langle \lambda y^o. y, \lambda z^o. \lambda v^o. z \rangle : o \rightarrow o$$

$$\begin{array}{c}
 (\lambda x^{(o \rightarrow o) \times (o \rightarrow o \rightarrow o)}. \pi_0(\underbrace{x}_{:(o \rightarrow o) \times (o \rightarrow o \rightarrow o)})) \langle \lambda y^o. \underbrace{y}_{:o}, \lambda z^o. \lambda v^o. \underbrace{z}_{:o} \rangle \\
 \underbrace{\hspace{15em}}_{:(o \rightarrow o) \times (o \rightarrow o \rightarrow o)} \quad \underbrace{\hspace{10em}}_{:o \rightarrow o} \quad \underbrace{\hspace{10em}}_{o \rightarrow o} \\
 \underbrace{\hspace{15em}}_{((o \rightarrow o) \times (o \rightarrow o \rightarrow o)) \rightarrow o \rightarrow o} \quad \underbrace{\hspace{10em}}_{o \rightarrow o \rightarrow o} \\
 \underbrace{\hspace{15em}}_{(o \rightarrow o) \times (o \rightarrow o \rightarrow o)} \\
 \underbrace{\hspace{15em}}_{o \rightarrow o}
 \end{array}$$

β -Reduction for Pairs

- β -reduction for the pairs is the rule which allows to replace

- any subterm of the form $\pi_0(\langle r_0, r_1 \rangle)$ by r_0 ,
- any subterm of the form $\pi_1(\langle r_0, r_1 \rangle)$ by r_1 .

- The subterms

$$\pi_i(\langle r_0, r_1 \rangle)$$

are called β -redexes of the term in question

- In addition we have the β -redexes $(\lambda x.t)$ of the λ -calculus with \rightarrow .
- β -reduction for the typed λ -calculus with products includes both β -reduction for functions and β -reduction for pairs.

Example

$$\begin{aligned} & (\lambda x^{(o \rightarrow o) \times (o \rightarrow o \rightarrow o)}. \pi_0(x)) \langle \lambda y^o. y, \lambda z^o. \lambda v^o. z \rangle \\ & \longrightarrow_{\beta} \pi_0(\langle \lambda y^o. y, \lambda z^o. \lambda v^o. z \rangle) \\ & \longrightarrow_{\beta} \lambda y^o. y \end{aligned}$$

Products with many Components

- We write $\sigma_0 \times \cdots \times \sigma_n$ for $(\cdots ((\sigma_0 \times \sigma_1) \times \sigma_2) \cdots \times \sigma_n)$.
- Define for $s_0 : \sigma_0, \dots, s_n : \sigma_n$

$$\langle s_0, \dots, s_n \rangle := \langle \cdots \langle \langle s_0, s_1 \rangle, s_2 \rangle, \cdots s_n \rangle : \sigma_0 \times \cdots \times \sigma_n$$

- E.g. $\langle x, y, z \rangle := \langle \langle x, y \rangle, z \rangle$.

- One can easily define corresponding projections $\pi_i^n : (\sigma_0 \times \cdots \times \sigma_{n-1}) \rightarrow \sigma_i$, s.t.

$$\pi_i^n(\langle s_0, \dots, s_{n-1} \rangle) =_\beta s_i \quad .$$

- For instance in case $n = 3$ we need

$$\pi_i^3(\langle s_0, s_1, s_2 \rangle) = \pi_i^3(\langle \langle s_0, s_1 \rangle, s_2 \rangle) = s_i$$

Products with many Components

$$\pi_i^3(\langle\langle s_0, s_1 \rangle, s_2 \rangle) = s_i$$

● We obtain this by defining

● $\pi_0^3(x) := \pi_0(\pi_0(x))$

● Then $\pi_0^3(\langle\langle s_0, s_1 \rangle, s_2 \rangle)$
 $= \pi_0(\pi_0(\langle\langle s_0, s_1 \rangle, s_2 \rangle))$
 $= \pi_0(\langle s_0, s_1 \rangle)$
 $= s_0$

● $\pi_1^3(x) := \pi_1(\pi_0(x))$

● Then $\pi_1^3(\langle\langle s_0, s_1 \rangle, s_2 \rangle)$
 $= \pi_1(\pi_0(\langle\langle s_0, s_1 \rangle, s_2 \rangle))$
 $= \pi_1(\langle s_0, s_1 \rangle)$
 $= s_1$

Products with many Components

$$\pi_i^3(\langle\langle s_0, s_1 \rangle, s_2 \rangle) = s_i$$

- $\pi_2^3(x) := \pi_1(x)$

- Then
$$\begin{aligned}\pi_2^3(\langle\langle s_0, s_1 \rangle, s_2 \rangle) \\ &= \pi_1(\langle\langle s_0, s_1 \rangle, s_2 \rangle) \\ &= s_2\end{aligned}$$

η -Expansion for Products

- If we have a product $r : \sigma \times \tau$, then its projections are β -equal to the projections of $\langle \pi_0(r), \pi_1(r) \rangle$:
 - $\pi_0(\langle \pi_0(r), \pi_1(r) \rangle) =_{\beta} \pi_0(r)$,
 - $\pi_1(\langle \pi_0(r), \pi_1(r) \rangle) =_{\beta} \pi_1(r)$.
- Therefore, similarly to functions, we would like to have that every term $r : \sigma \times \tau$ is equal to $\langle \pi_0(r), \pi_1(r) \rangle$.
- The η -rule expresses that subterms $t : \sigma \times \tau$ can be η -expanded to $\langle \pi_0(t), \pi_1(t) \rangle$
- Details can be found on the next few slides, but won't be treated in the lecture.
- We jump over the rest of this Subsection and over SubSect. b.

η -Rule for Products

- However, as for functions, we need to impose some restrictions, in order to avoid circularities:
 - If t is of the form $\langle r_0, r_1 \rangle$, and if we allowed then the reduction $t \longrightarrow \langle \pi_0(t), \pi_1(t) \rangle$, we would get the following circular reduction:

$$\begin{aligned} t &\longrightarrow \langle \pi_0(t), \pi_1(t) \rangle \\ &\equiv \langle \pi_0(\langle r_0, r_1 \rangle), \pi_1(\langle r_0, r_1 \rangle) \rangle \\ &\longrightarrow_{\beta}^* \langle r_0, r_1 \rangle \\ &\equiv t \end{aligned}$$

η -Rule for Products

- If t occurs in the form $\pi_i(t)$, and if we then allowed to expand t , we would get $\pi_i(t) \longrightarrow \pi_i(\langle \pi_0(t), \pi_1(t) \rangle)$ and would get the following circular reduction:

$$\begin{aligned} \pi_i(t) &\longrightarrow \pi_i(\langle \pi_0(t), \pi_1(t) \rangle) \\ &\longrightarrow_{\beta} \pi_i(t) \quad , \end{aligned}$$

- All other terms can be expanded without obtaining a new redex.

η -Expansion for Products

- η -expansion for products is the rule which allows to replace in a typed λ -term t
 - one subterm $s : \sigma \times \tau$,
 - which is not of the form $\langle r_0, r_1 \rangle$,
 - and does not occur in the form $\pi_0(s)$ or $\pi_1(s)$ by $\langle \pi_0(s), \pi_1(s) \rangle$.
- η -expansion for the typed λ -calculus with products includes both η -expansion for functions and for pairs.

Example

Assume $g : (o \times o) \rightarrow o$.

$$\begin{aligned} & (\lambda f^{(o \times o) \rightarrow o}. \lambda x^{o \times o}. f \ x) \ g \\ & \longrightarrow_{\beta} \lambda x^{o \times o}. g \ x \\ & \longrightarrow_{\eta} \lambda x^{o \times o}. g \ \langle \pi_0(x), \pi_1(x) \rangle \end{aligned}$$

$\lambda x^{o \times o}. g \ \langle \pi_0(x), \pi_1(x) \rangle$ is therefore the β, η -normal form of $(\lambda f^{(o \times o) \rightarrow o}. \lambda x^{o \times o}. f \ x) \ g$

Theorem

- The typed λ -calculus with products, β -reduction and with (or without) η -expansion is confluent and strongly normalising.
- We can introduce products as well for the untyped λ -calculus. Then we obtain a confluent (but of course non normalising) reduction system.

η -Rule

- With the η -rule we obtain now that if $r : \sigma \times \tau$, then $r =_{\beta, \eta} \langle \pi_0(r), \pi_1(r) \rangle$.

- If $r : \sigma \times \tau$ is of the form $\langle r_0, r_1 \rangle$ then we have $r =_{\beta} \langle \pi_0(r), \pi_1(r) \rangle$:

$$\begin{aligned} \langle \pi_0(r), \pi_1(r) \rangle &\equiv \langle \pi_0(\langle r_0, r_1 \rangle), \pi_1(\langle r_0, r_1 \rangle) \rangle \\ &\longrightarrow_{\beta}^* \langle r_0, r_1 \rangle \\ &\equiv r \end{aligned}$$

- Otherwise $r \longrightarrow_{\eta} \langle \pi_0(r), \pi_1(r) \rangle$.
- Therefore, every element of a product type is of the form $\langle \text{something}_0, \text{something}_1 \rangle$.

Jump over Currying/Uncurrying

(b) Currying

- In the λ -calculus with products, there are two versions of a function f taking an integer and a floating point number and returning a string:
 - $f_1 : (\text{Int} \times \text{Float}) \rightarrow \text{String}$
 - $f_2 : \text{Int} \rightarrow \text{Float} \rightarrow \text{String}$.
- We say
 - that f_1 is in Uncurried form,
 - and f_2 is in Curried form.
- The name “Curry” honours Haskell Curry.
- The application of these two functions to arguments x and y is written as

$$f_1 \langle x, y \rangle , \quad f_2 x y .$$

Haskell Brooks Curry



**Haskell Brooks Curry
(1900 - 1982)**

Curried/Uncurried Functions

- The above generalises to functions with arbitrarily (but finitely) many arguments of different type.
- The Curried version of a function f with arguments of types $\sigma_0, \dots, \sigma_{n-1}$ and result type ρ is of type

$$\sigma_0 \rightarrow \dots \rightarrow \sigma_{n-1} \rightarrow \rho \text{ .}$$

- Its Uncurried version has type

$$(\sigma_0 \times \dots \times \sigma_{n-1}) \rightarrow \rho \text{ .}$$

Uncurrying

- From a Curried function we can obtain an Uncurried function.

- This is called Uncurrying.

- **Example:**

- Assume

$$f : \text{Int} \rightarrow \text{Float} \rightarrow \text{String} .$$

- Then

$$\lambda x^{\text{Int} \times \text{Float}} . f \ \pi_0(x) \ \pi_1(x) : (\text{Int} \times \text{Float}) \rightarrow \text{String}$$

is the **Uncurried** form of f .

Currying

- From a Uncurried function we can obtain an Curried function.
 - This is called Currying.
 - **Example:**
 - Assume

$$f : (\text{Int} \times \text{Float}) \rightarrow \text{String} .$$

- Then

$$\lambda x^{\text{Int}} . \lambda y^{\text{Float}} . f \langle x, y \rangle : \text{Int} \rightarrow \text{Float} \rightarrow \text{String}$$

is the **Curried** form of f .

- On the next 2 slides follows a treatment of the general case.
Jump over general case.

Uncurrying

- We can obtain from the Curried form f_{Curry} of a function its Uncurried form f_{Uncurry} by

$$f_{\text{Uncurry}} = \lambda x. f_{\text{Curry}} \pi_0^n(x) \cdots \pi_{n-1}^n(x)$$

where $\pi_i^n : (\sigma_0 \times \cdots \times \sigma_{n-1}) \rightarrow \sigma_i$ are the projections.

- One can as well define a λ -term

$$\text{Uncurry} : (\sigma_0 \rightarrow \cdots \sigma_{n-1} \rightarrow \rho) \rightarrow (\sigma_0 \times \cdots \times \sigma_{n-1}) \rightarrow \rho$$

$$\text{Uncurry} := \lambda f, x. f \pi_0^n(x) \cdots \pi_{n-1}^n(x)$$

s.t. $\text{Uncurry } f_{\text{Curry}} \longrightarrow_{\beta} f_{\text{Uncurry}}$.

- This transformation is called Uncurrying.

Currying

- We can obtain from the Uncurried form f_{Uncurry} of a function its Curried form f_{Curry} by

$$f_{\text{Curry}} = \lambda x_0, \dots, x_{n-1}. f_{\text{Uncurry}} \langle x_0, \dots, x_{n-1} \rangle$$

- Again we can define

$$\text{Curry} : ((\sigma_0 \times \dots \times \sigma_{n-1}) \rightarrow \rho) \rightarrow \sigma_0 \rightarrow \dots \rightarrow \sigma_{n-1} \rightarrow \rho$$

$$\text{Curry} := \lambda f, x_0, \dots, x_{n-1}. f \langle x_0, \dots, x_{n-1} \rangle$$

s.t. $\text{Curry } f_{\text{Uncurry}} \longrightarrow_{\beta} f_{\text{Curry}}$.

- This transformation is called Currying.
- It is an easy exercise to show $\text{Curry } (\text{Uncurry } f) =_{\beta, \eta} f$ and $\text{Uncurry } (\text{Curry } f) =_{\beta, \eta} f$.

(Un)Currying in Programming

- The Uncurried form of a function corresponds to the form functions are presented usually outside functional programming.
- There functions always need all arguments.
 - “3+” is something which outside functional programming usually doesn’t make much sense.

(Un)Currying in Programming

- In functional programming one often prefers the Curried form.
 - This allows to apply a functional partially to its arguments.
 - E.g. if we take $_ + _$ as usual in Curried form, then $_ + _ 3 : \text{Int} \rightarrow \text{Int}$ is the function taking x and returning $_ + _ 3 x$ which is $3 + x$.
 - Example:

$$\text{map } (_ + _ 3) [1, 2, 3] = [4, 5, 6]$$

If we apply the function increasing every x by 3 to the list $[1, 2, 3]$, we obtain the result of incrementing each list element by 3, i.e. $[4, 5, 6]$.

(Un)Currying in Programming

- One often avoids in functional programming (and as well in Agda) the formation of products (or record types).
 - Especially for **intermediate calculations**.
 - The packing and unpacking of products makes programming often harder.
 - E.g. instead of defining a function $f : \sigma \rightarrow (\rho \times \tau)$ it is often better to form two functions $f_1 : \sigma \rightarrow \rho$ and $f_2 : \sigma \rightarrow \tau$, (which are often defined simultaneously).
- Only, when delivering the **final program**, the use of products is often better, because the result is more compact.

(c) The Nondep. Product in Agda

- In Agda, there are two ways of defining the product.
- The first one represents the product as a **record type**.

Records in Pascal

- In many languages there exists the notion of a Record type.
- In Pascal we can form for instance the type of Students

```
Student = record
    begin
        StudentNumber : Integer;
        Name           : String;
    end
```

- Elements of this type can be formed by determining their StudentNumber and Name.
- If $x : \text{Student}$, then
 $x.\text{StudentNumber} : \text{Integer}$ and $x.\text{Name} : \text{String}$.

Records in Java

- Records correspond in Java to classes with public fields, no methods, and a standard constructor.
- E.g. the class `Student` is defined as follows:

```
class Student{  
    Integer    StudentNumber;  
    String     Name;  
    Student(Integer StudentNumber, String Name){  
        this.StudentNumber = StudentNumber;  
        this.Name          = Name  
    }  
}
```

The Record Type in Agda

- Assume we have introduced $A, B : \text{Set}$
Then we can introduce the record type

record $AB : \text{Set}$ where
field

$a : A$

$b : B$

Name Clashes in the Record Type

- You are not allowed to use a and b , if the identifiers a and b have been introduced before.
- However, you can use the same record selector in different records.
- So

$n : \mathbb{N}$

$n = Z$

record $A : \text{Set}$ where

field $n : \mathbb{N}$

causes an error.

Name Clashes in the Record Type

● However

record A : Set where
field $n : \mathbb{N}$

record A' : Set where
field $n : \mathbb{N} \rightarrow \mathbb{N}$

is accepted.

Longer Records

- We can introduce longer records as well, e.g.

record $ABCD$: Set where
field

a : A

b : B

c : C

d : D

The Product as a Record Type

- Elements of a record type are introduced as follows:
Assume we have $a' : A, b' : B$.
Then we can introduce in the above situation

$$\begin{aligned} ab & : AB \\ ab & = \text{record}\{a = a'; b = b'\} \end{aligned}$$

- Note that, since a, b cannot be record selectors and separate identifiers at the same time, the ambiguous definition

$$\text{record}\{a = a; b = b\}$$

is not possible.

The Product as a Record Type

- However, if we use let expressions, then we can obtain such an ambiguous situation:

$$ab \quad : \quad AB$$

$$ab \quad = \quad \text{let}$$

$$a \quad : \quad A$$

$$a \quad = \quad a'$$

$$b \quad : \quad B$$

$$b \quad = \quad b'$$

$$\text{in record}\{a = a; b = b'\}$$

- We recommend to avoid such definitions.

Projections

● If we have

record AB : Set where
field

a : A

b : B

then Agda provides us with the following projection functions:

$AB.a$: $AB \rightarrow A$

$AB.b$: $AB \rightarrow B$

Projections

● If we define

$$\begin{aligned} ab &: AB \\ ab &= \text{record}\{a = a'; b = b'\} \end{aligned}$$

then we obtain

$$AB.a \ ab = a' \quad AB.b \ ab = b'$$

Records with Dependencies

- We can define a generic product $\text{rProd } A \ B$ depending on $A : \text{Set}$, $B : \text{Set}$ (rProd stands for record-product):

$\text{record rProd } (A \ B : \text{Set}) \text{ where}$

field

$\text{first} \quad : \ A$

$\text{second} \quad : \ B$

- The projections are denoted as follows:
If $ab : \text{rProd } A \ B$, then

$\text{rProd.first} \quad ab \quad : \ A$

$\text{rProd.second} \quad ab \quad : \ B$

Hidden Arguments

- When we use

$$\text{rProd.first} : \text{rProd } A \ B \rightarrow A$$

it is not always clear, which sets A and B one is referring to.

- In fact A and B are **hidden arguments** of rProd.first .
- In case one needs to make them explicit, this can be done as follows:

$$\text{rProd.first } \{ A' \} \{ B' \} ab$$

stands for rProd.first applied to ab , where $ab : \text{rProd } A' \ B'$.

Hidden Arguments

- We can make any argument of a function hidden.
- For instance

$$\begin{aligned}\text{id} & : \{A : \text{Set}\} \rightarrow A \rightarrow A \\ \text{id } a & = a\end{aligned}$$

defines the identity function, which for any set A and $a : A$ returns a .

- This function is used in the form

$$\text{id } a$$

without adding the parameter A .

Hidden Arguments

- If we want to make the hidden parameter A explicit we can do so by writing

$$\text{id } \{ A \} a$$

Hidden Arguments

- There is no deep theory about when arguments can be hidden or not.
- Any argument of a function can be declared to be hidden.
- If when type checking the code Agda cannot determine a hidden argument, then Agda will get **unsolved hidden goals**.

Example

- Take the following code

$$\begin{aligned}\text{strange} & : \{a : A\} \rightarrow A \\ \text{strange } \{a\} & = a\end{aligned}$$
$$\begin{aligned}a & : A \\ a & = \text{strange}\end{aligned}$$

- Agda doesn't complain about the definition of `strange`.
- However, when checking the definition of `a`, it notices that it cannot figure out the hidden argument of `strange`.

Example

strange : $\{a : A\} \rightarrow A$

strange $\{a\} = a$

$a : A$

$a =$ strange

● It complains by

- Marking the word strange in yellow.
- Displaying a **hidden goal** in the buffer *All Goals*
_184 : A [at /home/csetzerlocal/test.agda:166,7-14]
- This means that for the missing hidden argument of strange a hidden goal has been introduced, which is of type A , and the position (line 166, column 7 - 14) is displayed.

The Product using “data”

- The second version of the product uses the more general **data** construct for defining so called **algebraic types**.
- With this construction we are leaving the so called **logical framework**.
 - λ -terms and the record type form the **logical framework**, the basic types of Agda and of Martin-Löf type theory.
 - The **data**-construct allows to introduce **user-defined types**.

The Product using “data”

- The “data”-product is introduced as follows (dProd stands for data-product):

$$\text{data dProd } (A \ B : \text{Set}) : \text{Set where}$$
$$p : A \rightarrow B \rightarrow \text{dProd } A \ B$$

- Here
 - dProd $A \ B$ depends on two sets A, B .
 - p is the constructor of this set.
 - The name (here p) is up to the user, we could have used any other valid Agda identifier.
- The idea is:
 - The elements of Prod' are exactly the terms $p \ a \ b$ where $a : A$ and $b : B$.

Pattern Matching

- In order to decompose an element of $\text{dProd } A \ B$ in Agda, we can use **pattern matching**.
- This is best explained by an example.
- We postulate $A, B : \text{Set}$, and abbreviate $\text{dProd } A \ B$ as AB :

postulate $A : \text{Set}$

postulate $B : \text{Set}$

$AB \quad : \quad \text{Set}$

$AB \quad = \quad \text{dProd } A \ B$

Pattern Matching

postulate $A : \text{Set}$

postulate $B : \text{Set}$

$AB : \text{Set}$

$AB = \text{dProd } A \ B$

- Assume we want to define the first projection

$$\text{proj0} : AB \rightarrow A ,$$

s.t.

$$\text{proj0 } (\text{p } a \ b) = a$$

- This can be defined as follows:

$$\text{proj0} : AB \rightarrow A ,$$

$$\text{proj0 } (\text{p } a \ b) = a$$

Pattern Matching

postulate $A : \text{Set}$

postulate $B : \text{Set}$

$AB : \text{Set}$

$AB = \text{dProd } A \ B$

- The second projection can be defined similarly:

$$\text{proj1} : AB \rightarrow B \ ,$$
$$\text{proj1 } (\text{p } a \ b) = b$$

- Note the parentheses around $(\text{p } a \ b)$:

$$\text{proj1 } \text{p } a \ b = b$$

would read: proj1 applied to a variable p , a variable a and a variable b is equal to b .

This causes an error, because proj1 only allows one argument.

Deep Pattern Matching

postulate $A : \text{Set}$

postulate $B : \text{Set}$

$AB : \text{Set}$

$AB = \text{dProd } A \ B$

- Deeper pattern matching is as well possible: An element of $\text{dProd } (\text{dProd } A \ B) \ B$ is of the form

$$p \ (p \ a' \ b') \ b''$$

where $a' : A, b', b'' : B$.

- We can define

$$f : \text{dProd } (\text{dProd } A \ B) \ B \rightarrow A$$

$$f \ (p \ (p \ a \ b) \ b') = a$$

Deep Pattern Matching

- We are not allowed to use the same variable twice in a pattern (unless specially flagged – flagged repeated variables occur only in advanced data types like the identity type).
- So

$$f : \text{dProd} (\text{dProd } A \ B) \ B \rightarrow A$$

$$f \ (p \ (p \ a \ b) \ b) = a$$

causes an error.

Coverage Checker

- The **coverage checker** of Agda will make sure that the patterns cover all possible cases.
- So

$$\begin{aligned} f &: \mathbb{N} \rightarrow \mathbb{N} \\ f \ Z &= Z \end{aligned}$$

will not pass the coverage checker, because $f \ (S \ n)$ is not defined.

Hidden Arguments in dProd

● p in

$$\text{data dProd } (A \ B : \text{Set}) : \text{Set where}$$
$$p : A \rightarrow B \rightarrow \text{dProd } A \ B$$

has hidden arguments $\{A : \text{Set}\}$ and $\{B : \text{Set}\}$.

● In case one needs to make them explicit, one can do so:

$$c \quad : \quad \text{dProd } A \ B$$
$$c \quad = \quad p \ \{A\} \ \{B\} \ a \ b$$

Hidden Arguments in dProd

- If one wants to mention the first hidden argument, but not the second one, one simply omits the second one:

$$c \quad : \quad \text{dProd } A \ B$$

$$c \quad = \quad \text{p } \{A\} \ a \ b$$

- The following syntax allows to omit the first hidden argument, but to mention the second one:

$$c \quad : \quad \text{dProd } A \ B$$

$$c \quad = \quad \text{p } \{_\} \ \{B\} \ a \ b$$

- In general, variables which are not used later can be written as `_`.

Decomposing Record Type

- Let

$$D : \text{Set}$$

$$D = \text{rProd} (\text{dProd } A \ B) \ C$$

- Assume we want to define $f : D \rightarrow A$ which projects an element of D to the component A .
- Pattern matching is not possible for record types.
- What we can do is to use the “with”-construct

$$f : D \rightarrow A$$

$$f \ d \text{ with } \text{rProd.first } d$$

$$f \ d \mid p \ a \ b = a$$

Decomposing Record Type

$f : D \rightarrow A$

$f\ d$ with $\text{rProd.first}\ d$

$f\ d \mid p\ a\ b = a$

- The above reads as follows:
- We define $f\ d$ by looking at $\text{rProd.first}\ d$.
- We look at what happens when $\text{rProd.first}\ d = p\ a\ b$.
- In this case we define $f\ d$ as a .

Longer Example

- As an example we want to define in Agda, depending on
 - $A, B, C, D : \text{Set}$,
 - $ab : A \times B$
 - $a\text{-}c : A \rightarrow C$,
 - $b\text{-}d : B \rightarrow D$an element
 - $f\ ab\ a\text{-}c\ b\text{-}d : C \times D$.
- This means that f is a function which takes arguments $a\text{-}c$, $b\text{-}d$ and ab as above and returns an element of $C \times D$.
- Therefore

$$f : (A \times B) \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow D) \rightarrow (C \times D)$$

Longer Example

- $A, B, C, D : \text{Set}$ will be global assumptions (represented in Agda by postulates).
- So we have the following Agda code:

```
postulate A : Set
postulate B : Set
postulate C : Set
postulate D : Set
```

Longer Example

- Let AB and CD be names for $A \times B$ and $C \times D$, respectively.
- Then we obtain the following code:

```
record  $AB$  : Set where
  field
     $a : A$ 
     $b : B$ 
record  $CD$  : Set where
  field
     $c : C$ 
     $d : D$ 
```

Longer Example

- The goal to be solved is as follows:

$$f : (A \rightarrow C) \rightarrow (B \rightarrow D) \rightarrow AB \rightarrow CD$$
$$f \ a\text{-}c \ b\text{-}d \ ab = \{! \ !\}$$

Longer Example

- The idea for this function is as follows:
 - We first project $ab : A \times B$ to elements $a : A, b : B$.
 - Then we apply

$$a\text{-}c : A \rightarrow C$$

to $a : A$ and obtain an element

$$c : C \text{ .}$$

- And we apply

$$b\text{-}d : B \rightarrow D$$

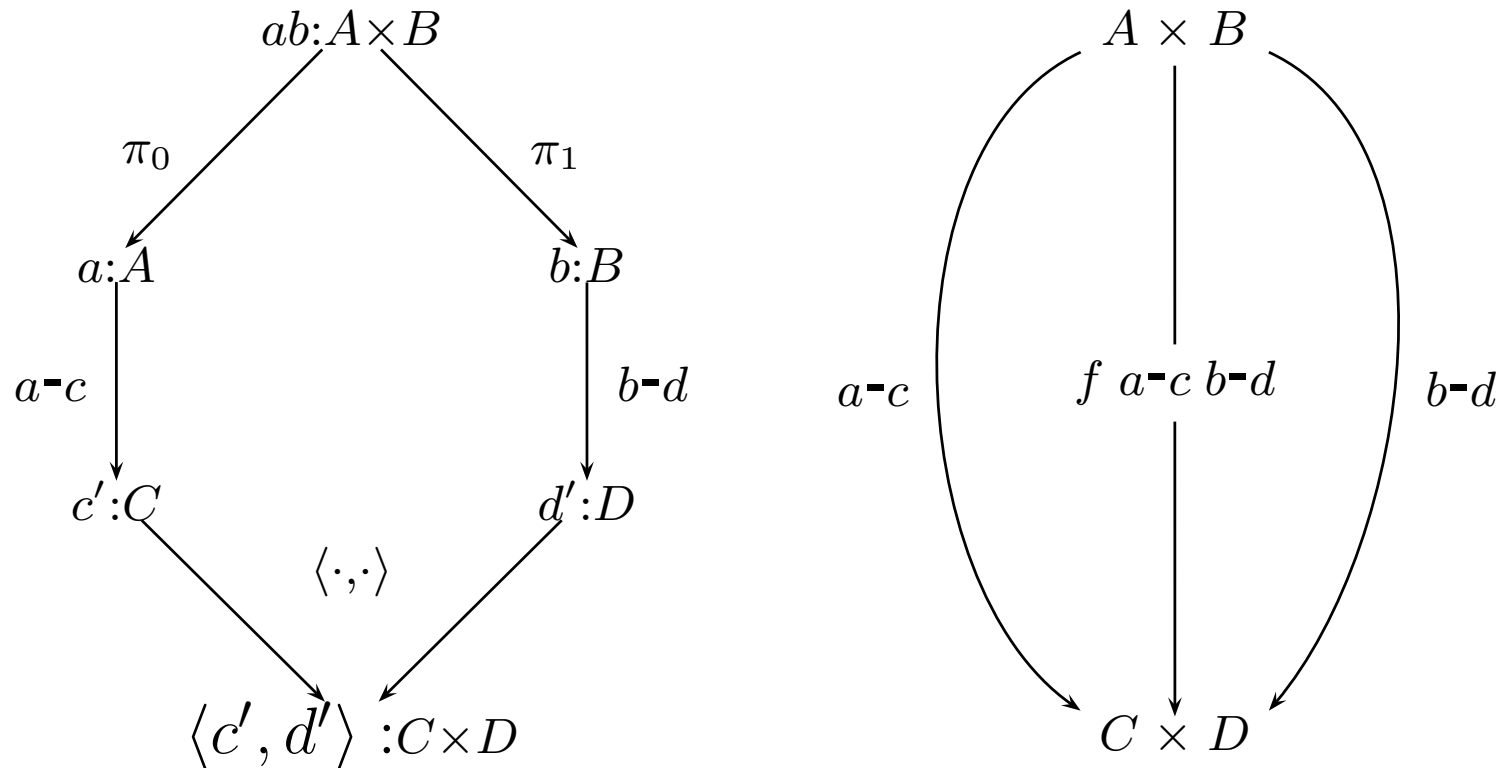
to $b : B$ and obtain an element

$$d : D \text{ .}$$

- Finally we form the pair $\langle b, d \rangle$.

Longer Example

- A diagram is as follows:



- We will use `let`-expressions in order to compute the intermediate values a , b , c' , d' .

Agda Code for the Above

$$\begin{aligned} f &: (A \rightarrow C)(B \rightarrow D) \rightarrow AB \rightarrow CD \\ f \ a\text{-}c \ b\text{-}d \ ab &= \text{let } a' : A \\ &\quad a' = AB.a \ ab \\ &\quad b' : B \\ &\quad b' = AB.b \ ab \\ &\quad c' : C \\ &\quad c' = a\text{-}c \ a \\ &\quad d' : D \\ &\quad c' = b\text{-}d \ b \\ &\text{in record}\{c = c'; d = d'\} \end{aligned}$$

See [exampleLetExpressionRecord.agda](#).

Remark on Previous Code

- In the previous code we used in the let expression variables c' and d' instead of c and d .
- This is to avoid the ambiguity in

$$\text{record}\{c = c; d = d\}$$

- Agda will interpret this example as intended, but it is not clear whether this will be always the case.

Concrete Products

- When using the `data`-construct, it is often more convenient to introduce concrete products in a more direct way.
- **Example:** Assume we have defined
 - a set `Gender` of genders,
 - a set `Name` of names.
 - The set of **persons**, given by a gender and a name, can then be defined as

`data Person : Set where`

`person : Gender → Name → Person`

Concrete Products

- Then one can define customized projections using pattern matching, e.g.

$\text{gender} : \text{Person} \rightarrow \text{Gender}$

$\text{gender} (\text{person } g \ n) = g$

(d) Logic with Conjunction

Constructive Meaning of \wedge

- $A \wedge B$ is true, if A is true and B is true.
- Therefore a proof $p : A \wedge B$ consists
 - of a proof $a : A$
 - and a proof $b : B$.
- So such a proof is a pair $\langle a, b \rangle$ s.t. $a : A$ and $b : B$.
- Therefore $A \wedge B$ is just the product $A \times B$ of A and B .
- We can identify $A \wedge B$ with $A \times B$.

Conjunction in Agda

- Conjunction is represented as a product.
- There are two products in Agda, therefore as well two ways of representing conjunction:
 - One using the record type:

```
record  $\_ \wedge r \_$  ( $A \ B : \text{Set}$ ) : Set where
  field
    and1  :  $A$ 
    and2  :  $B$ 
```

- The symbol \wedge can be introduced by typing in `\wedge`.

Conjunction in Agda

- And one using the product formed using `data`.
We use a more meaningful name for the constructor:

```
data _∧d_ (A B : Set) : Set where  
  and : A → B → A ∧d B
```

- See [exampleproofproplogic3.agda](#)

Typing in Special Symbols

- Typing in the special symbols (using the Emacs-package “mule”) can be cumbersome.
- A more convenient way is to use the abbreviation mode:
- To activate the abbreviation mode, use under emacs **M-x abbrev-mode**
- Then one can let an arbitrary sequence of characters to be automatically replaced by an abbreviation.

Typing in Special Symbols

- For instance if we want “andd” to expand to \wedge we do the following:
 - We type in “andd”.
 - We use the emacs command **C-x ail**
 - We type in the mini buffer our intended expansion, namely \wedge (typed in as “\wedge”).
 - Now whenever we type in a space-like character (blanks and some punctuations) followed by “andd” followed by a space-like character, then “andd” is replaced by \wedge .
 - You can edit the abbreviations you have defined by using **M-x edit-abbrevs** (when finished use **C-c C-c** in order to activate your definitions).

Abbreviation Mode

- You can prevent the expansion of an abbreviation by using **C-q** before adding any space-like character after “andd”.

Customising Agda with Abbreviation

- In order to load previous abbreviations and save the when exiting Agda you should add the following to your .emacs file:

```
(read-abbrev-file "~/.abbrev_defs" )
```

- However, for this to work you need first to create a file ~/.abbrev_defs
- This is done by following the steps on the next slide

Customising Agda with Abbreviation

- The creation of a file `~/.abbrev_defs` is done as follows (the steps need to be carried out only once):
 - Define at least one abbreviation as above (you can change this abbreviation later by using **M-x edit-abbrevs**.
 - For instance you can just type in `fooo`, type in **C-x ail**, and then type in the Mini-buffer `foo`, so that `fooo` is expanded to `foo`.
 - Then execute **M-x write-abbrev-file**, and when asked for a file name, enter in the mini-buffer `~/.abbrev_defs`
 - Now execute **M-x read-abbrev-file**, and when asked for a file name, enter in the mini-buffer `~/.abbrev_defs`

Customising Agda with Abbreviation

- If you now create a new abbreviation, and run **C-x s** which is the command for saving all buffers, it will ask as well whether you want to save the abbreviation file.

Customising Agda with Abbreviation

- In order to activate the abbreviation mode, whenever one enters an Agda file, add in your .emacs file after the line

```
(load "~/.emacs/agdainstall")
```

the following

```
(add-hook 'agda2-mode-hook  
          '(lambda nil (abbrev-mode 1)))
```

Example

- On the computer $A \rightarrow A \wedge A$ and $A \wedge B \rightarrow A$ will now be shown in Agda using both versions of \wedge .

Example (Conjunction)

- We prove $A \wedge B \rightarrow B \wedge A$ (see [exampleproofproplogic6.agda](#)):

Lemma : Set

Lemma = $A \wedge_r B \rightarrow B \wedge_r A$

lemma : Lemma

lemma *ab* = record{and1 = $_ \wedge_r _.and2$ *ab*;
and2 = $_ \wedge_r _.and1$ *ab*}

Example (Conjunction)

Lemma' : Set

Lemma' = $A \wedge_d B \rightarrow B \wedge_d A$

lemma' : Lemma'

lemma' (and a b) = and b a

Conjunction with more Conjuncts

- If one has a conjunction with more than two conjuncts, e.g. $A \wedge B \wedge C$, one can always express it using the binary \wedge :
 - As $(A \wedge_r B) \wedge_r C$ or $A \wedge_r (B \wedge_r C)$.
 - If one adds

infixl 30 \wedge_r

one can write

$A \wedge_r B \wedge_r C$

for

$(A \wedge_r B) \wedge_r C$

Conjunction with more Conjuncts

- Especially when using the record version of \wedge it is more convenient to use a ternary version of conjunction (using one of the two versions of the product).
- Similarly one can introduce conjunctions of 4 or more conjuncts.
- Definition of the ternary and using a record:

record And3r ($A \ B \ C : \text{Set}$) : Set where
 field
 and1 : A
 and2 : B
 and3 : C

Conjunction with more Conjuncts

- Definition of the ternary and using “data”:

$\text{data And3d } (A \ B \ C : \text{Set}) : \text{Set where}$
 $\text{and3d} : A \rightarrow B \rightarrow C \rightarrow \text{And3d } A \ B \ C$

See [exampleproofproplogic5.agda](#)

(e) The λ -Calc. and Term Rewriting

- One can combine the λ -calculus with term writing.
- This means that we have apart from the rules of the typed or untyped λ -calculus additional rules like $x + 0 \longrightarrow x$.

- Then we obtain for instance

$$\lambda y. \lambda z. y + 0 \longrightarrow \lambda y. \lambda z. y \quad .$$

- More details are given on the following slides, but will not be treated in this lecture.
Jump over rest of this section.

λ -Calculus and Term Rewriting

- Consider the λ -calculus with terms using additional constants.
- Assume some term rewriting rules as before (which might involve some λ -terms).
- As in case of ordinary term rewriting, we form instantiations \longrightarrow' of the rules by replacing variables by arbitrary λ -terms (in the extended language).

λ -Calculus and Term Rewriting

- Then $s \longrightarrow t$, if
 - s β -reduces (or η -expands, if one allows the η -rule) to t
 - or there exists an instantiation $s' \longrightarrow t'$ s.t. s' is a subterm of s and t is the result of replacing this subterm in s by t' .
 - s' is called as usual a redex of s .

λ -Calculus and Term Rewriting

- Assume for instance the rule

$$\text{double} \longrightarrow \lambda x. x + x$$

Then we have

$$\begin{aligned} & (\lambda f. \lambda x. f (f x)) \text{double} \\ & \longrightarrow \lambda x. \text{double} (\text{double } x) \\ & \longrightarrow \lambda x. \text{double} ((\lambda x. x + x) x) \\ & \longrightarrow \lambda x. \text{double} (x + x) \\ & \longrightarrow \lambda x. (\lambda x. x + x) (x + x) \\ & \longrightarrow \lambda x. (x + x) + (x + x) \end{aligned}$$

What does Subterm Mean?

- When referring to ordinary term rewriting rules, then for a term t to have subterm s meant essentially that there is a term t' in which a new variable x occurs exactly once, and $t = t'[x := s]$.
- Replacing this subterm by s' means that we replace t by $t'[x := s']$.

What does Subterm Mean?

- When referring to λ -terms, this is no longer the case:
 - Assume for instance the rewrite rule $x + 0 \longrightarrow_{\text{Rule}} x$.
 - $\lambda x.x + 0$ has subterm $x + 0$, but there is no term t s.t. $\lambda x.x + 0 = t[y := x + 0]$:
If we substitute for instance in $\lambda x.y$ y by $x + 0$ we obtain $\lambda z.x + 0$.
- The reason is that when matching a rewrite rule, free variables in the instantiation of the rule used might become bound.
- So we can apply $x + 0 \longrightarrow_{\text{Rule}} x$ to $\lambda x.x + 0$ and have therefore $\lambda x.x + 0 \longrightarrow \lambda x.x$.
- Replacing a subterm by another subterm is to be understood verbally.

Higher Order Rewrite Systems

- The full definition of so called higher order term rewriting systems imposes more restrictions on the reduction rules.
- For our purposes the naive interpretation just presented suffices.

Jump over next part.

Reduction to Closed Terms

- One can always replace term rewriting rules for the λ -calculus by one in which for all rules $s \longrightarrow_{\text{Rule}} t$ we have that s, t are closed.
- This can be done in such a way that equality (modulo the rewriting rules, β and possibly η) in both systems coincide:
- Assume a rule

$$s \longrightarrow_{\text{Rule}} t$$

and let x_1, \dots, x_n be the free variables in s .

- Then replace this rule by

$$\lambda x_1, \dots, x_n. s \longrightarrow_{\text{Rule}'} \lambda x_1, \dots, x_n. t \ .$$

Proof

- We write in the following \vec{x} for x_1, \dots, x_n .
- Assume a term r reduces using this rule in the original system to a term u :
- Then r contains a subterm of the form s' where s' is the result of substituting in s x_i by some terms t_i .
- Let t' be the result of substituting in t x_i by t_i . Then u is the result of replacing s' in r by t' .
- Let then r' be the result of replacing s' by $(\lambda \vec{x}.s) t_1 \cdots t_n$, and u' be the result of replacing in s s' by $(\lambda \vec{x}.t) t_1 \cdots t_n$.
- Then we have $r =_{\beta} r' \longrightarrow_{\text{Rule}'} u' =_{\beta} u$, so the reduction can be simulated in the second system.

Proof

- On the other hand, if $r \longrightarrow u$ by using in the second system the rule $\lambda\vec{x}.s \longrightarrow_{\text{Rule}'} \lambda\vec{x}.t$, then $r \longrightarrow u$ in the previous system by using the rule $s \longrightarrow_{\text{Rule}} t$
 - r contains a subterm equal to $\lambda\vec{x}.s$ and u is the result of substituting this subterm in r by $\lambda\vec{x}.t$.
 - But then r contains the subterm s and t is the result of substituting this subterm in r by t .

Example

- We can replace the rewriting rules

$$\begin{aligned}x + 0 &\longrightarrow x \\x + S\ y &\longrightarrow S\ (x + y)\end{aligned}$$

by

$$\begin{aligned}\lambda x.x + 0 &\longrightarrow \lambda x.x \\ \lambda x, y.x + S\ y &\longrightarrow \lambda x, y.S\ (x + y)\end{aligned}$$

- That

$$S\ (0 + S\ 0) \longrightarrow S\ (S\ (0 + 0)) \longrightarrow S\ (S\ 0)$$

becomes in the new system

$$\begin{aligned}S\ (0 + S\ 0) &=_{\beta} S\ ((\lambda x, y.x + S\ y)\ 0\ 0) \\ &\longrightarrow S\ ((\lambda x, y.S\ (x + y))\ 0\ 0) =_{\beta} S\ (S\ (0 + 0)) \\ &=_{\beta} S\ (S\ ((\lambda x.x + 0)\ 0)) \longrightarrow S\ (S\ ((\lambda x.x)\ 0)) =_{\beta} S\ (S\ 0)\end{aligned}$$

Extended Typed λ -Calculus

- Finally, we can combine the typed λ -calculus (with or without products, with or without η -expansion) with term rewriting rules.
- Essentially this means that we have additional constants with types and reduction rules for them.
- The details (which are given on the following slides) will not be treated in the lecture itself.

Extended Typed λ -Calculus

- For introducing the new rewrite rules, we have to make the following modifications:
 - We assign a type to each additional constant.
 - The set of typed λ -terms is then introduced by the same rules as before, but we have as additional rule:
 - If c is a constant of type σ , then we have

$$\Gamma \Rightarrow c : \sigma$$

Example

- Assuming $_ + _ : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$ and writing as usual $r + s$ for $_ + _ r s$ we have the following derivation of $\lambda x^{\text{nat}}.x + x : \text{nat} \rightarrow \text{nat}$:

$$\frac{\frac{x:\text{nat} \Rightarrow _ + _ : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \quad x:\text{nat} \Rightarrow x:\text{nat}}{\quad} (\text{Ap})}{\frac{x:\text{nat} \Rightarrow _ + _ \quad x:\text{nat} \rightarrow \text{nat} \quad x:\text{nat} \Rightarrow x:\text{nat}}{\quad} (\text{Ap})} (\text{Abs})$$
$$(\lambda x^{\text{nat}}.x + x) : \text{nat} \rightarrow \text{nat}$$

- The left most leaf in this derivation follows by the rule for the constant $_ + _$.

Example

Then we have

$$\begin{aligned} & (\lambda f^{\text{nat} \rightarrow \text{nat}}. \lambda x^{\text{nat}}. f (f x)) \text{ double} \\ & \longrightarrow \lambda x^{\text{nat}}. \text{ double } (\text{ double } x) \\ & \longrightarrow \lambda x^{\text{nat}}. \text{ double } ((\lambda x^{\text{nat}}. x + x) x) \\ & \longrightarrow \lambda x^{\text{nat}}. \text{ double } (x + x) \\ & \longrightarrow \lambda x^{\text{nat}}. (\lambda x^{\text{nat}}. x + x) (x + x) \\ & \longrightarrow \lambda x^{\text{nat}}. (x + x) + (x + x) \end{aligned}$$

Extended Typed λ -Calculus

- Reduction rules should now be of the form $\Gamma \Rightarrow s \longrightarrow_{\text{Rule}} t : \sigma$ (instead of $s \longrightarrow_{\text{Rule}} t$) where we have $\Gamma \Rightarrow s : \sigma$ and $\Gamma \Rightarrow t : \sigma$.
- As before, s shouldn't be a variable, and all variables in t should occur in s .
 - Best guaranteed by demanding that all variables in Γ occur free in s .
- One usually omits Γ, σ , if it is clear from the context.
- Very often, the reduction rules will be of the form $c \longrightarrow_{\text{Rule}} t : \sigma$ where c is a constant and therefore t a closed term.

Extended Typed λ -Calculus

- Instantiations of a rule $\Gamma \Rightarrow s \longrightarrow_{\text{Rule}} t : \sigma$ are now obtained by replacing variables x of type τ by terms $r : \tau$ (possibly depending on some context Δ).
- Reductions w.r.t. the rules are obtained by replacing subterms $r : \sigma$, which coincide with the left hand side of an instantiation of a rule $r \longrightarrow' r' : \sigma$ by the right hand side r' .

Example

- Assume
 - ground type nat ,
 - constants $_ + _ : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$ (written infix, i.e. $r + s$ for $_ + _ r s$),
 - and $\text{double} : \text{nat} \rightarrow \text{nat}$.
 - and the reduction rule
 $\text{double} \longrightarrow (\lambda x^{\text{nat}}. x + x) : \text{nat} \rightarrow \text{nat}$.

Example

Then we have

$$\begin{aligned} & (\lambda f^{\text{nat} \rightarrow \text{nat}}. \lambda x^{\text{nat}}. f (f x)) \text{ double} \\ & \longrightarrow \lambda x^{\text{nat}}. \text{ double } (\text{ double } x) \\ & \longrightarrow \lambda x^{\text{nat}}. \text{ double } ((\lambda x. x + x) x) \\ & \longrightarrow \lambda x^{\text{nat}}. \text{ double } (x + x) \\ & \longrightarrow \lambda x^{\text{nat}}. (\lambda x. x + x) (x + x) \\ & \longrightarrow \lambda x^{\text{nat}}. (x + x) + (x + x) \end{aligned}$$

(f) Finite Sets and Decidable Formu

- We want to add types containing finitely many elements to the λ -calculus.
- We treat first the special case `Bool` (finite set with 2 elements) and then generalise this to general finite sets.

The Type of Booleans

- We add a new type

Bool

to the set of ground types.

- We add constants

$\text{tt} : \text{Bool} , \quad \text{ff} : \text{Bool} .$

- Here tt stands for true, ff for false.

Case_{Bool}^σ

- Furthermore we add the principle of case distinction to the λ -calculus extended by Bool:
 - Assume we have a type σ and

$$case_{tt} : \sigma \qquad case_{ff} : \sigma$$

- Then we want to have that

$$Case_{Bool}^{\sigma} case_{tt} case_{ff} : Bool \rightarrow \sigma$$

- And we want that

$$Case_{Bool}^{\sigma} case_{tt} case_{ff} tt = case_{tt}$$

$$Case_{Bool}^{\sigma} case_{tt} case_{ff} ff = case_{ff}$$

If then else

$$\text{Case}_{\text{Bool}}^{\sigma} \text{ case}_{\text{tt}} \text{ case}_{\text{ff}} \text{ tt} = \text{case}_{\text{tt}}$$

$$\text{Case}_{\text{Bool}}^{\sigma} \text{ case}_{\text{tt}} \text{ case}_{\text{ff}} \text{ ff} = \text{case}_{\text{ff}}$$

● $\text{Case}_{\text{Bool}}^{\sigma} \text{ case}_{\text{tt}} \text{ case}_{\text{ff}} b$ corresponds to

If b then case_{tt} else case_{ff} :

- In case $b = \text{tt}$, the if-then-else-term should be equal to case_{tt} , as it is the case for $\text{Case}_{\text{Bool}}^{\sigma} \text{ case}_{\text{tt}} \text{ case}_{\text{ff}} b$.
- In case $b = \text{ff}$, the if-then-else-term should be equal to case_{ff} , as it is the case for $\text{Case}_{\text{Bool}}^{\sigma} \text{ case}_{\text{tt}} \text{ case}_{\text{ff}} b$.

Type of $\text{Case}_{\text{Bool}}^\sigma$

- We don't need to have a complex rule for forming $\text{Case}_{\text{Bool}}^\sigma \text{ case}_{\text{tt}} \text{ case}_{\text{ff}}$.
- All we need to do is add a constant $\text{Case}_{\text{Bool}}^\sigma$ of type

$$\text{Case}_{\text{Bool}}^\sigma : \sigma \rightarrow \sigma \rightarrow \text{Bool} \rightarrow \sigma$$

- Then it follows that, whenever $\text{case}_{\text{tt}} : \sigma$ and $\text{case}_{\text{ff}} : \sigma$, then

$$\text{Case}_{\text{Bool}}^\sigma \text{ case}_{\text{tt}} \text{ case}_{\text{ff}} : \text{Bool} \rightarrow \sigma \quad :$$

- The equalities are achieved by adding reductions

$$\text{Case}_{\text{Bool}}^\sigma \text{ case}_{\text{tt}} \text{ case}_{\text{ff}} \text{ tt} \longrightarrow \text{case}_{\text{tt}}$$

$$\text{Case}_{\text{Bool}}^\sigma \text{ case}_{\text{tt}} \text{ case}_{\text{ff}} \text{ ff} \longrightarrow \text{case}_{\text{ff}}$$

Example: Boolean Conjunction

- We define Boolean valued conjunction

$$_ \wedge_{\text{Bool}} _ : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} .$$

- We write

- $_ \wedge_{\text{Bool}} _$ for function symbol,
- \wedge_{Bool} for the symbol, written infix,

so $b \wedge_{\text{Bool}} c$ stands for $_ \wedge_{\text{Bool}} _ b c$.

- Note that this will be an operation on Booleans.
 - Above we introduced the operation on formulae, which takes two formulae A and B and forms the formula $A \wedge B$.
 - $b \wedge_{\text{Bool}} c$ will form the Boolean value corresponding to the conjunction of b and c .

Truth Table for \wedge_{Bool}

- \wedge_{Bool} has the following truth table:

\wedge_{Bool}	ff	tt
ff	ff	ff
tt	ff	tt

- So we have

$$\begin{aligned}\text{ff} \wedge_{\text{Bool}} b &= \text{ff} \\ \text{tt} \wedge_{\text{Bool}} b &= b\end{aligned}$$

Example: \wedge_{Bool}

- Below we will see how to define for every Boolean value $b : \text{Bool}$ a formula $\text{Atom } b$ corresponding to this value.
- Then one can show that $(\text{Atom } b) \wedge (\text{Atom } c)$ is equivalent to $\text{Atom } (b \wedge_{\text{Bool}} c)$.
- This means that $b \wedge_{\text{Bool}} c$ is true iff b is true and c is true.

Example: \wedge_{Bool}

$_ \wedge_{\text{Bool}} _ : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}.$

- $_ \wedge_{\text{Bool}} _$ will be introduced by λ -abstraction, so we get

$$_ \wedge_{\text{Bool}} _ = \lambda(b, c : \text{Bool}).t$$

for some term t .

- t will be defined by case distinction on b , and have result Bool , so we get

$$_ \wedge_{\text{Bool}} _ = \lambda(b, c : \text{Bool}).\text{Case}_{\text{Bool}}^{\text{Bool}} e f b$$

for some e, f .

Example: \wedge_{Bool}

$$_ \wedge_{\text{Bool}} _ = \lambda(b, c : \text{Bool}). \text{Case}_{\text{Bool}}^{\text{Bool}} e f b$$

● For conjunction we have:

● We have seen that

$$\text{tt} \wedge_{\text{Bool}} c = c$$

● So the if-case e above is c .

● Furthermore

$$\text{ff} \wedge_{\text{Bool}} c = \text{ff}$$

● So the else-case f above is ff .

Example: \wedge_{Bool}

- In total we define therefore

$$\underline{\wedge_{\text{Bool}}} = \lambda(b, c : \text{Bool}). \text{Case}_{\text{Bool}}^{\text{Bool}} c \text{ ff } b$$
$$: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$$

- We verify the correctness of this definition:
 - $\text{tt} \wedge_{\text{Bool}} c = _ \wedge_{\text{Bool}} _ \text{tt } c = \text{Case}_{\text{Bool}}^{\text{Bool}} c \text{ ff } \text{tt} = c.$
as desired.
 - $\text{ff} \wedge_{\text{Bool}} c = _ \wedge_{\text{Bool}} _ \text{ff } c = \text{Case}_{\text{Bool}}^{\text{Bool}} c \text{ ff } \text{ff} = \text{ff}.$
Correct as desired.

The Finite Sets

- Bool can be generalised to sets having n elements (n a fixed natural number):
 - We add for every $n \in \mathbb{N}$ a new ground type Fin_n .
 - We add for every $k \in \mathbb{N}$ s.t. $k < n$ a new constant

$$A_k^n : \text{Fin}_n$$

- Informally we will have have

$$\text{Fin}_n = \{A_0^n, A_1^n, \dots, A_{n-1}^n\}$$

especially in the cases $n = 2, 1, 0$ we have

$$\text{Fin}_2 = \{A_0^2, A_1^2\}$$

$$\text{Fin}_1 = \{A_0^1\}$$

$$\text{Fin}_0 = \emptyset$$

The Finite Sets

- We have not made use of dependent types yet. n, k are external natural numbers.
 - So we have for each n added one type Fin_n to the calculus.
 - We have for each n and $k < n$ added one constant A_k^n to the calculus.

Rules for Fin_n

- We add the principle of case distinction on Fin_n :
- Assume $n \in \mathbb{N}$, a type σ , and $\text{case}_i : \sigma$ for $i = 0, \dots, n - 1$.
- Then we want

$$\text{Case}_n^\sigma \text{ case}_0 \cdots \text{case}_{n-1} : \text{Fin}_n \rightarrow \sigma$$

- And we want

$$\text{Case}_n^\sigma \text{ case}_0 \cdots \text{case}_{n-1} A_i^n = \text{case}_i$$

Constants Case_n^σ

- As for Bool, this can be achieved by having constants

$$\text{Case}_n^\sigma : \underbrace{\sigma \rightarrow \cdots \rightarrow \sigma}_{n \text{ times}} \rightarrow \text{Fin}_n \rightarrow \sigma$$

- Then from $\text{case}_i : \sigma$ we obtain

$$\text{Case}_n^\sigma \text{ case}_0 \cdots \text{case}_{n-1} : \text{Fin}_n \rightarrow \sigma$$

- Furthermore we add the reduction rules

$$\text{Case}_n^\sigma \text{ case}_0 \cdots \text{case}_{n-1} A_i^n \longrightarrow \text{case}_i$$

Equality on Fin_n

- We can now define the Boolean valued function which determines for two elements of Fin_n , whether they are equal:

- Define

$$\text{Eq}_{\text{n},\text{Bool}} : \text{Fin}_n \rightarrow \text{Fin}_n \rightarrow \text{Bool}$$

s.t.

$$\text{Eq}_{\text{n},\text{Bool}} A_n^i A_n^i = \text{tt}$$

$$\text{Eq}_{\text{n},\text{Bool}} A_n^i A_n^j = \text{ff} \quad \text{for } i \neq j$$

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

Special Case Bool

- Bool can now be treated as the special case

Fin_n

with $n = 2$:

$$\begin{aligned}\text{Bool} &:= \text{Fin}_2 \\ \text{tt} &:= A_0^2 &: \text{Bool} \\ \text{ff} &:= A_1^2 &: \text{Bool} \\ \text{Case}_{\text{Bool}}^\sigma &:= \text{Case}_2^\sigma &: \sigma \rightarrow \sigma \rightarrow \text{Bool} \rightarrow \sigma\end{aligned}$$

Rules for \top

- \top (pronounced “top”) is the special case

$$\text{Fin}_n$$

for $n = 1$

(we write `true` for A_0^1):

- So we have a type $\top := \text{Fin}_1$,
- $\text{true} := A_0^1 : \top$,
- $\text{Case}_\top^\sigma := \text{Case}_1^\sigma : \sigma \rightarrow \sigma$.
- $\text{case}_\top^\sigma a \text{ true} \longrightarrow a$.

\top as the True Formula

- Above we have seen that
 - formulae can be identified with types
 - for a formula to be true means to have an element of its type.
- \top has exactly one proof, and corresponds therefore to the always **always true formula**.
 - That's why we call the element

true ,

since it is the proof of the always true formula.

- Example: we have

$$\lambda x^A.\text{true} : A \rightarrow \top$$

Rules for \perp

\perp (pronounced “bottom”) is the special case

$$\text{Fin}_n$$

for $n = 0$:

- $\perp := \text{Fin}_0$.
- \perp has no element (Fin_n has no element).
- Case distinction on Fin_0 is empty – the number of cases is 0, so we get the empty case distinction.
 - This means that we have

$$\text{Case}_{\perp}^{\sigma} : \perp \rightarrow \sigma$$

- We have no reduction rules.



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



- $\text{Case}_{\perp}^{\sigma}$ expresses: **from an element f of \perp we obtain an element of any set.**
 - Correct, since there is no element of \perp .
 - Considered as a formula, $\text{Case}_{\perp}^{\sigma}$ means: from a proof of \perp we obtain a proof of every other formula.
 - I.e. it means \perp implies everything.
 - In logic this principle is called “**Ex falsum quodlibet**” (from the absurdity follows anything).
E.g. A **false formula** like “ $0 = 1$ ” or “Swansea lies in Germany” **implies everything**.
- For any formula A we have a proof of $\perp \rightarrow A$:

$$\text{Case}_{\perp}^A : \perp \rightarrow A .$$

Negation

- The negation $\neg A$ of a formula 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 $A \rightarrow \perp$ is true:
 - If there is no proof of A , then from every proof of A we can obtain a proof of \perp (since there is no proof of A); therefore $A \rightarrow \perp$ is true.
 - On the other hand, if we $A \rightarrow \perp$ is true, i.e. has a proof, then there cannot be any proof of A , because from it we could get a proof of \perp , which is the empty set.
- Therefore $\neg A$ is true iff $A \rightarrow \perp$ is true.
- Therefore we can identify $\neg A$ with $A \rightarrow \perp$.

(g) Fin. Sets/Decid. Formul. in Agda

- We introduce `Bool` by **listing its constructors**

```
data Bool : Set where
  tt : Bool
  ff : Bool
```

Pattern Matching

- We can use pattern matching in order to make case distinction on an argument of type `Bool`:
- Assume we want to define

$$\neg \text{Bool} \quad : \quad \text{Bool} \rightarrow \text{Bool}$$
$$\neg \text{Bool } \text{tt} \quad = \quad \text{ff}$$
$$\neg \text{Bool } \text{ff} \quad = \quad \text{tt}$$

- The above is already the Agda code defining $\neg \text{Bool}$.
`exemplenegbool.agda`

Finite Sets in Agda

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

```
data Colour : Set where
  blue   : Colour
  red    : Colour
  green  : Colour
```

Finite Sets in Agda

- Case distinction on finite sets in Agda can be done using pattern matching.
- In the “Colour” example above for instance, we can define

`is-red : Colour → Bool`

`is-red red = tt`

`is-red _ = ff`

- The above has an **overlapping case distinction**: the line

`is-red _ = ff`

matches `is-red red`.

Finite Sets in Agda

`is-red` : `Colour` \rightarrow `Bool`

`is-red red` = `tt`

`is-red _` = `ff`

- The convention is that if there are overlapping patterns, then the first pattern is the one which is used.
- So `is-red red` will be computed by having the first pattern, we get

`is-red red` = `tt`

- `is-red blue` and `is-red green` are computed using the second pattern, we get

`is-red blue` = `is-red green` = `ff`

\top in Agda

- The definition of \top in Agda is **straightforward**:

```
data  $\top$  : Set where
  true :  $\top$ 
```

- We can define a function having an argument in \top by using pattern matching:

```
 $g$       :  $\top \rightarrow \text{Bool}$ 
 $g$  true = tt
```

\top in Agda

- Alternatively, we can define \top in Agda as the empty record (note that there is no keyword field):

$\text{record } \top' : \text{Set where}$

- Then the element true of \top is defined as follows

$\text{true}' : \top'$
 $\text{true}' = \text{record}\{ \}$

- Agda has a builtin η -rule, which says that every $x : \top$ is equal to $\text{record}\{ \}$.
exampletrue.agda

\perp in Agda

- \perp can be defined as the “data”-set **with no constructors**:

$\text{data } \perp : \text{Set where}$

- If we want to define

$g : \perp \rightarrow \text{Bool}$

by pattern matching, we see that there is no element in \perp , so there is no constructor case matching $g\ x$.

\perp in Agda

- We need to communicate this to Agda (this is needed in order to obtain decidability of pattern matching) by having the following code:

$$g : \perp \rightarrow \text{Bool}$$
$$g \ ()$$

- The code $g \ ()$ means:
the argument at the position $()$ is an element which matches no pattern, so this case is solved.

examplefalse.agda

\neg in Agda

- Above we have shown why we can define $\neg A$ as $A \rightarrow \perp$.
- Therefore negation can be defined in Agda as follows:

$$\begin{aligned}\neg & : \text{Set} \rightarrow \text{Set} \\ \neg A & = A \rightarrow \perp\end{aligned}$$

Example for the Use of \perp

- Assume the **type of trees**:

data Tree : Set where

oak : Tree

pine : Tree

spruce : Tree

- We can now define

IsConifer : Tree \rightarrow Set

IsConifer oak = \perp

IsConifer _ = \top

- So IsConifer x is the false formula, if $x = \text{oak}$, and the true formula otherwise.

Example for the Use of \perp

- If we want to define a function from trees, which are conifers, into another set, we can do so by requiring **an additional argument** “IsConifer”:

$$\begin{aligned} f &: (t : \text{Tree}) \rightarrow \text{IsConifer } t \rightarrow A \\ f \quad \text{pine} \quad _ &= \{! \ !\} \\ f \quad \text{spruce} \quad _ &= \{! \ !\} \\ f \quad \text{oak} \quad () & \end{aligned}$$

- So we need to define f only for `pine` and `spruce`, the case where the first argument is `oak` cannot appear, since in that case the second argument is an element of the empty set, i.e. it matches no pattern.

Example for the Use of \perp

- Note that we **don't have to invent a result** of f in case t is an oak tree.

exampletree1.agda

Jump over Example 2 (Stack)

Example 2 for the Use of \perp

- Assume the type `Stack` of stacks of elements of \mathbb{N} given by

`data Stack (A : Set) : Set where`
 `empty : Stack A`
 `push : A → Stack A → Stack A`

- We can then introduce a predicate `NonEmpty` expressing that the stack is nonempty:

`NonEmpty : {A : Set} → Stack A → Set`
`NonEmpty empty = \perp`
`NonEmpty (push _ _) = \top`

Example 2 for the Use of \perp

- Now we can define

$$\begin{aligned} \text{top} &: \{A : \text{Set}\} \rightarrow (s : \text{Stack } A) \rightarrow \text{NonEmpty } s \rightarrow \text{Set} \\ \text{top } \text{empty} & \quad () \\ \text{top } (\text{push } a \text{ } _) & \quad _ = a \end{aligned}$$

(See [exampleStack.agda](#)).

- Again we **don't have to provide a result, in case s is empty** (in general we couldn't provide such a result, since A might be empty).

Atomic Formulae

- We will now show how to convert in Agda a Boolean value into a formula.
- Here we will leave the simply-typed λ -calculus, and move to dependent types.
- The operation which converts Boolean values into atomic formulae is

$$\text{Atom} \quad : \quad \text{Bool} \rightarrow \text{Set}$$
$$\text{Atom tt} \quad = \quad \top$$
$$\text{Atom ff} \quad = \quad \perp$$

Atomic Formulae

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

$\text{Atom } \text{tt} = \top$

$\text{Atom } \text{ff} = \perp$

- So, in case $b = \text{tt}$, $\text{Atom } b$ is the true formula, which is provable.
- In case $b = \text{ff}$, $\text{Atom } b$ is the false formula, which is unprovable.

exampleAtom.agda

Example

- Above we introduced the Boolean valued equality on Fin_n , which for fixed n can be defined in Agda.

$$\begin{aligned}\text{Eq}_{\text{n},\text{Bool}} &: \text{Fin}_n \rightarrow \text{Fin}_n \rightarrow \text{Bool} \\ \text{Eq}_{\text{n},\text{Bool}} A_n^i A_n^i &= \text{tt} \\ \text{Eq}_{\text{n},\text{Bool}} _ _ &= \text{ff}\end{aligned}$$

Example

- For instance in case of the set

```
data Colour : Set where
  blue   : Colour
  red    : Colour
  green  : Colour
```

we define

```
eqColourBool : Colour → Colour → Bool
eqColourBool blue   blue   = tt
eqColourBool red    red    = tt
eqColourBool green  green  = tt
eqColourBool _      _      = ff
```

Example

- We can now convert this equality into a formula as follows:

$$\begin{aligned} _ == _ & : \text{Colour} \rightarrow \text{Colour} \rightarrow \text{Set} \\ c == c' & = \text{Atom} (\text{eqColourBool } c \ c') \end{aligned}$$

- $c == c'$ is the formula expressing that c and c' are the same colour.

Example

- `_==_` can be defined directly, by unfolding its definition.
- We obtain:

$$\begin{array}{llll} _ ==' _ : \text{Colour} \rightarrow \text{Colour} \rightarrow \text{Set} \\ \text{blue} \quad ==' \quad \text{blue} \quad = \quad \top \\ \text{red} \quad ==' \quad \text{red} \quad = \quad \top \\ \text{green} \quad ==' \quad \text{green} \quad = \quad \top \\ _ \quad ==' \quad _ \quad = \quad \perp \end{array}$$

exampleColourEquality.agda

Example 2

- Remember the definition of Boolean valued negation in Agda:

$$\neg \text{Bool} : \text{Bool} \rightarrow \text{Bool}$$
$$\neg \text{Bool} \quad \text{tt} \quad = \quad \text{ff}$$
$$\neg \text{Bool} \quad \text{ff} \quad = \quad \text{tt}$$

- We show

$$\text{Atom} (\neg \text{Bool} \, b) \rightarrow \neg (\text{Atom} \, b)$$

- Remember that we defined

$$\neg \quad : \quad \text{Set} \rightarrow \text{Set}$$
$$\neg \, A \quad = \quad A \rightarrow \perp$$

Example 2

- So our lemma is

Lemma : Set

Lemma = $(b : \text{Bool}) \rightarrow \text{Atom } (\neg \text{Bool } b) \rightarrow \neg (\text{Atom } b)$

- Since $\neg A = A \rightarrow \perp$ this is equivalent to

Lemma = $(b : \text{Bool}) \rightarrow \text{Atom } (\neg \text{Bool } b) \rightarrow \text{Atom } b \rightarrow \perp$

- We need to show

lemma : Lemma

lemma $b\ p\ q$ = $\{! \ !\}$

Example 2

Lemma : Set

Lemma = $(b : \text{Bool}) \rightarrow \text{Atom } (\neg \text{Bool } b) \rightarrow \text{Atom } b \rightarrow \perp$

lemma : Lemma

lemma $b\ p\ q$ = $\{! \ !\}$

- We need to make case distinction on $b = \text{tt}$ and $b = \text{ff}$ and replace the last line by the two cases:

lemma : Lemma

lemma $\text{tt}\ p\ q$ = $\{! \ !\}$

lemma $\text{ff}\ p\ q$ = $\{! \ !\}$

Example 2

Lemma : Set

Lemma = $(b : \text{Bool}) \rightarrow \text{Atom } (\neg \text{Bool } b) \rightarrow \text{Atom } b \rightarrow \perp$

lemma : Lemma

lemma tt $p\ q$ = $\{! \ !\}$

lemma ff $p\ q$ = $\{! \ !\}$

- In the first equation we have $b = \text{tt}$, therefore

$$p : \text{Atom } (\neg \text{Bool } b) = \text{Atom ff} = \perp$$

- So p matches no pattern, we can replace in this case p by $()$, and have solved this case.

Example 2

Lemma : Set

Lemma = $(b : \text{Bool}) \rightarrow \text{Atom } (\neg \text{Bool } b) \rightarrow \text{Atom } b \rightarrow \perp$

lemma : Lemma

lemma tt () q

lemma ff p q = {! !}

- In the second case we have $b = \text{ff}$, so

$$q : \text{Atom } b = \text{Atom } \text{ff} = \perp$$

- So q matches no pattern, we can replace in this case q by $()$, and have solved this case as well

Example 2

```
Lemma      : Set
Lemma      = (b : Bool) → Atom (¬Bool b) → Atom b → ⊥
lemma      : Lemma
lemma tt () q
lemma ff p ()
```

- Note that it becomes increasingly complicated to guarantee that all cases are covered. Therefore it is important to check that the code has passed the **coverage checker**.
[examplenegbool2.agda](#)
[Jump over Example 3 \(→Bool\)](#)

Example 3

- We introduce Boolean valued implication

$$_ \rightarrow \text{Bool} _ : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$$

and show that $\text{Atom } (b \rightarrow \text{Bool } b')$ implies $\text{Atom } b \rightarrow \text{Atom } b'$.

- The other direction can be shown as well.

Example 3

- Classically $A \rightarrow B$ is true iff A is false or B is true.
 - Something false implies everything, so $A \rightarrow B$ is true if A is false.
 - If A is true, then $A \rightarrow B$ is true if B is true.
- So we have $A \rightarrow B$ is false if A is true and B is false. In all other cases it is true.
- We can therefore define the Boolean valued implication as follows

$_ \rightarrow \text{Bool} _ : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

$\text{tt} \rightarrow \text{Bool} \text{ ff} = \text{ff}$

$_ \rightarrow \text{Bool} _ = \text{tt}$

Example 3

- We introduce the Lemma to be shown and the pattern for the proof:

Lemma : Set

Lemma = $(b\ b' : \text{Bool}) \rightarrow \text{Atom } (b \rightarrow \text{Bool } b') \rightarrow \text{Atom } b$
 $\rightarrow \text{Atom } b'$

lemma : Lemma

lemma $b\ b'\ b \rightarrow b'\ btrue = \{! \ !\}$

- We try to make a case distinction which makes as often as possible one of the two proof objects $b \rightarrow b' : \text{Atom } (b \rightarrow \text{Bool } b')$ or $btrue : \text{Atom } b$ false.

Example 3

lemma : $(b\ b' : \text{Bool}) \rightarrow \text{Atom } (b \rightarrow \text{Bool } b') \rightarrow \text{Atom } b$
 $\rightarrow \text{Atom } b'$

lemma $b\ b'\ b \rightarrow b'\ btrue = \{! \ !\}$

● If $b = \text{ff}$, then

$$btrue : \text{Atom } b = \perp$$

which matches the empty pattern.

● If $b = \text{tt}$, $b' = \text{ff}$, then

$$b \rightarrow b' : \text{Atom } (b \rightarrow \text{Bool } b') = \text{Atom } \text{ff} = \perp$$

which again matches the empty pattern.

Example 3

lemma : $(b\ b' : \text{Bool}) \rightarrow \text{Atom } (b \rightarrow \text{Bool } b') \rightarrow \text{Atom } b$
 $\rightarrow \text{Atom } b'$

lemma b b' b \rightarrow b' btrue = {! !}

● If $b = \text{tt}$, $b' = \text{tt}$, the goal is

$$\text{Atom } b' = \top$$

which is provable by using tt .

● We obtain the following proof:

Example 3

Lemma : Set

Lemma = $(b\ b' : \text{Bool}) \rightarrow \text{Atom } (b \rightarrow \text{Bool } b') \rightarrow \text{Atom } b$
 $\rightarrow \text{Atom } b'$

lemma : Lemma

lemma ff _ _ ()

lemma tt ff () _

lemma tt tt _ _ = true

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. the **equality on the natural numbers** is decidable, since we can define a function

$$\text{Eq}_{\mathbb{N}, \text{Bool}} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Bool}$$

which decides it.

- The equality on **functions** $\mathbb{N} \rightarrow \mathbb{N}$ 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 $n : \mathbb{N}$.

Decidable Predicates (Cont.)

- Assume we have a set of real world states

RealWorldState : Set

- e.g. the set of states of the signals and switches of a railway interlocking system,
- a set of control states

ControlState : Set

- e.g. the set of states a railway controller can choose,

Decidable Predicates (Cont.)

- and a function

$\text{control} \rightarrow \text{realWorld} : \text{ControlState} \rightarrow \text{RealWorldState}$

mapping control states to external states they represent.

- Furthermore, assume we have defined in Agda a function

$\text{safe}_{\text{Bool}} : \text{RealWorldState} \rightarrow \text{Bool} \ .$

- The intended meaning is that

$\text{safe}_{\text{Bool}} \ s$

means: **real world state s is safe.**

Decidable Predicates (Cont.)

● Let now

$$\begin{aligned}\text{Safe} & : \text{RealWorldState} \rightarrow \text{Set} , \\ \text{Safe } s & = \text{Atom}(\text{safe}_{\text{Bool}} s) .\end{aligned}$$

- If $\text{safe}_{\text{Bool}} s$ is **true** (e.g. s is safe), $\text{Safe } s$ is **inhabited**, i.e. provable.
- If $\text{safe}_{\text{Bool}} s$ is **false** (e.g. s is unsafe), $\text{Safe } s$ is **not inhabited**.

Decidable Predicates (Cont.)

- The existence of a
 $p : (s : \text{ControlState}) \rightarrow \text{Safe} (\text{control} \rightarrow \text{realWorld } s)$
means:
 - For every $s : \text{ControlState}$ we have that if
 $s' := \text{control} \rightarrow \text{realWorld } s$ is the corresponding real
world state, then $\text{Safe } s'$ is **inhabited**,
 - i.e. $\text{Safe } s'$ is **true**,
 - i.e. s' is **safe**.

Decidable Predicates (Cont.)

- So if we have a proof

$$p : (a : \text{ControlState}) \rightarrow \text{Safe} (\text{control} \rightarrow \text{realWorld } s)$$

we have shown that **the system is safe** w.r.t. the safety property expressed by `safeBool`.