CS_336/CS_M36 (part 2)/CS_M46 Interactive Theorem Proving

Course Notes

Lent Term Term 2008

Sect. 4 The λ-Calculus with Products and Conjunction

Anton Setzer

Dept. of Computer Science, Swansea University

http://www.cs.swan.ac.uk/∼csetzer/lectures/intertheo/07/index.html

May 24, 2017

4 (a) The Typed λ-Calculus with Products

4 (b) Currying

4 (c) The Nondependent Product in Agda

4 (d) Logic with Conjunction

4 (e) The λ-Calculus and Term Rewriting

4 (f) Finite Sets and Decidable Formulae

4 (g) Finite Sets and Decidable Formulae in Agda

4 (a) The Typed λ-Calculus 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

$$\begin{array}{lcl} \text{Name} & := & \text{String} \\ \text{Gender} & := & \{\text{female}, \text{male}\} \end{array}$$

  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

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

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

Example2 (Products)

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

$$\mathrm{Int} \to \mathrm{Int} \ .$$

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

$$\mathrm{Term} \times (\mathrm{Int} \to \mathrm{Int})$$

Products

- The set of typed-λ-terms are defined as before but we have:
  - If $s : \sigma$, $t : \tau$ then $\langle s, t \rangle : \sigma \times \tau$:

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

  - 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

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

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

$$o\to o$$

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 $(\lambda x.t)\ s$ 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

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

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 $s_0 : \sigma_0, \ldots, s_n : \sigma_n$

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

(Note by our convention that the type is equal to $(\cdots((\sigma_0 \times \sigma_1) \times \sigma_2) \cdots \times \sigma_n))$

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

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

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

- 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 $\pi_2^3(\langle\langle s_0, s_1\rangle, s_2\rangle)$
    $= \pi_1(\langle\langle s_0, s_1\rangle, s_2\rangle)$
    $= s_2$

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:

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

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:

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

- 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 $g : (\mathrm{o} \times \mathrm{o}) \to \mathrm{o}$.

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

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

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 $r =_\beta \langle \pi_0(r), \pi_1(r) \rangle$:

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

  - 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

4 (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:
  - $f_1 : (\mathrm{Int} \times \mathrm{Float}) \to \mathrm{String}$
  - $f_2 : \mathrm{Int} \to \mathrm{Float} \to \mathrm{String}$.
- 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

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

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 $\sigma_0, \ldots, \sigma_{n-1}$ and result type ρ is of type

$$\sigma_0 \to \cdots \to \sigma_{n-1} \to \rho \ .$$

  - Its **Uncurried version** has type

$$(\sigma_0 \times \cdots \times \sigma_{n-1}) \to \rho \ .$$

Uncurrying

- From a Curried function we can obtain an Uncurried function.
  - This is called **Uncurrying**.
  - **Example:**
    - Assume
      $$f : \mathrm{Int} \to \mathrm{Float} \to \mathrm{String} \ .$$
    - Then
      $$\lambda x^{\mathrm{Int} \times \mathrm{Float}}.f\ \pi_0(x)\ \pi_1(x) : (\mathrm{Int} \times \mathrm{Float}) \to \mathrm{String}$$
      is the **Uncurried** form of f.

Currying

- From a Uncurried function we can obtain an Curried function.
  - This is called **Currying**.
  - **Example:**
    - Assume
      $$f : (\text{Int} \times \text{Float}) \to \text{String} \ .$$
    - Then
      $$\lambda x^{\text{Int}}.\lambda y^{\text{Float}}.f\ \langle x, y\rangle : \text{Int} \to \text{Float} \to \text{String}$$
      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 $\pi_i^n : (\sigma_0 \times \cdots \times \sigma_{n-1}) \to \sigma_i$ are the projections.
- One can as well define a λ-term

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

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

  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

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

- Again we can define

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

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

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 $\_ + \_\ 3 : \mathrm{Int} \to \mathrm{Int}$ is the function taking x and returning $\_ + \_\ 3\ x$ which is $3 + x$.
  - Example:

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

    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 $f : \sigma \to (\rho \times \tau)$ it is often better to form two functions $f_1 : \sigma \to \rho$ and $f_2 : \sigma \to \tau$, (which are often defined simultaneously).
- Only, when delivering the **final program**, the use of products is often better, because the result is more compact.

4 (c) The Nondependent 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 : Student, then x.StudentNumber : Integer and x.Name : 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 $A, B : \mathrm{Set}$
  Then we can introduce the record type

$$\begin{array}{l}\mathrm{record}\ AB : \mathrm{Set}\ \mathrm{where} \\ \quad \mathrm{field} \\ \qquad a \ : \ A \\ \qquad b \ : \ B\end{array}$$

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

$$
\begin{aligned}
&n : \mathbb{N}\\
&n = \mathrm{Z}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{record}\ A : \mathrm{Set}\ \mathrm{where}\\
&\quad \mathrm{field}\qquad n : \mathbb{N}
\end{aligned}
$$

causes an error.

Name Clashes in the Record Type

- However

$$\begin{array}{l}\text{record } A : \text{Set where} \\ \quad \text{field} \quad n : \mathbb{N}\end{array}$$

$$\begin{array}{l}\text{record } A' : \text{Set where} \\ \quad \text{field} \quad n : \mathbb{N} \to \mathbb{N}\end{array}$$

  is accepted.

Longer Records

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

$$
\begin{array}{l}
\text{record } \mathit{ABCD} : \text{Set where} \\
\quad \text{field} \\
\qquad \begin{array}{lcl} a & : & A \\ b & : & B \\ c & : & C \\ d & : & D \end{array}
\end{array}
$$

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

$$\begin{array}{lcl} ab & : & AB \\ ab & = & \mathrm{record}\{a = a'; b = b'\} \end{array}$$

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

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

  is not possible.

The Product as a Record Type

- Using a let expression we can get around this problem:

$$
\begin{array}{lcl}
ab & : & AB \\
ab & = & \mathrm{let} \\
 & & \quad a' \;:\; A \\
 & & \quad a' \;=\; a \\
 & & \quad b' \;:\; B \\
 & & \quad b' \;=\; b \\
 & & \mathrm{in\ record}\{a = a'; b = b'\}
\end{array}
$$

- We recommend to avoid such definitions.

Projections

- If we have

$$\begin{array}{l}\text{record } AB : \text{Set where} \\ \quad \text{field} \\ \qquad a \;:\; A \\ \qquad b \;:\; B\end{array}$$

then Agda provides us with the following projection functions:

$$\begin{array}{lcl} AB.a & : & AB \to A \\ AB.b & : & AB \to B \end{array}$$

Projections

- If we define

$$\begin{array}{lcl} ab & : & AB \\ ab & = & \mathrm{record}\{a = a'; b = b'\} \end{array}$$

  then we obtain

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

Records with Dependencies

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

$$\begin{array}{l}\mathrm{record}\ \mathrm{rProd}\ (A\ B : \mathrm{Set}) : \mathrm{Set}\ \mathrm{where} \\ \quad \mathrm{field} \\ \qquad \begin{array}{lcl} \mathrm{first} & : & A \\ \mathrm{second} & : & B \end{array}\end{array}$$

  Here $(A\ B : \mathrm{Set})$ stands for the two parameters $(A : \mathrm{Set})\ (B : \mathrm{Set})$

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

$$\begin{array}{lcl} \mathrm{rProd.first}\ ab & : & A \\ \mathrm{rProd.second}\ ab & : & B \end{array}$$

Hidden Arguments

- When we use

$$\mathrm{rProd.first} : \mathrm{rProd}\ A\ B \to A$$

  it is not always clear, which sets A and B one is referring to.
- In fact A and B are **hidden arguments** of $\mathrm{rProd.first}$.
- In case one needs to make them explicit, this can be done as follows:

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

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

Hidden Arguments

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

$$\begin{array}{lcl} \mathrm{id} & : & \{A : \mathrm{Set}\} \to A \to A \\ \mathrm{id}\ a & = & a \end{array}$$

  defines the identity function, which for any set A and $a : A$ returns a.
- This function is used in the form

$$\mathrm{id}\ a$$

  without adding the parameter A.

Hidden Arguments

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

$$\mathrm{id}\ \{\ A\ \}\ a$$

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

$$\begin{array}{lcl} \mathrm{strange} & : & \{a : A\} \to A \\ \mathrm{strange}\ \{a\} & = & a \end{array}$$

$$\begin{array}{lcl} a & : & A \\ a & = & \mathrm{strange} \end{array}$$

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

Example

$$
\begin{array}{lcl}
\mathrm{strange} & : & \{a : A\} \to A \\
\mathrm{strange}\ \{a\} & = & a \\
a & : & A \\
a & = & \mathrm{strange}
\end{array}
$$

- It complains by
  - Marking the word $\mathrm{strange}$ 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 $\mathrm{strange}$ 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):

$$\begin{aligned}&\mathrm{data}\ \mathrm{dProd}\ (A\ B : \mathrm{Set}) : \mathrm{Set}\ \mathrm{where}\\ &\quad \mathrm{p} : A \to B \to \mathrm{dProd}\ A\ B\end{aligned}$$

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

$$\begin{aligned}
&\mathrm{postulate}\ A : \mathrm{Set}\\
&\mathrm{postulate}\ B : \mathrm{Set}\\
&AB \quad : \quad \mathrm{Set}\\
&AB \quad = \quad \mathrm{dProd}\ A\ B
\end{aligned}$$

Pattern Matching

$$\text{postulate } A : \text{Set}$$
$$\text{postulate } B : \text{Set}$$
$$AB : \text{Set}$$
$$AB = \text{dProd } A\ B$$

- Assume we want to define the first projection

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

s.t.

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

- This can be defined as follows:

$$\text{proj0} : AB \to A \ ,$$
$$\text{proj0}\ (\text{p}\ a\ b) = a$$

Pattern Matching

$\mathrm{postulate}\ A : \mathrm{Set}$
$\mathrm{postulate}\ B : \mathrm{Set}$
$AB : \mathrm{Set}$
$AB = \mathrm{dProd}\ A\ B$

- The second projection can be defined similarly:

$$\mathrm{proj1} : AB \to B \ ,$$
$$\mathrm{proj1}\ (\mathrm{p}\ a\ b) = b$$

- Note the parentheses around $(\mathrm{p}\ a\ b)$:
  $\mathrm{proj1}\ \mathrm{p}\ a\ b = b$
  would read: $\mathrm{proj1}$ applied to a variable p, a variable a and a variable b is equal to b.
  This causes an error, because $\mathrm{proj1}$ only allows one argument.

Deep Pattern Matching

$$\text{postulate } A : \text{Set}$$
$$\text{postulate } B : \text{Set}$$
$$AB : \text{Set}$$
$$AB = \text{dProd } A\ B$$

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

$$\text{p}\ (\text{p}\ a'\ b')\ b''$$

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

- We can define

$$f : \text{dProd}\ (\text{dProd}\ A\ B)\ B \to A$$
$$f\ (\text{p}\ (\text{p}\ a\ b)\ b') = a$$

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

$$f : \mathrm{dProd}\ (\mathrm{dProd}\ A\ B)\ B \to A$$
$$f\ (\mathrm{p}\ (\mathrm{p}\ a\ b)\ b) = a$$

causes an error.

Coverage Checker

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

$$f : \mathbb{N} \to \mathbb{N}$$
$$f\ \mathrm{Z} = \mathrm{Z}$$

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

Hidden Arguments in dProd

- p in

$$\begin{aligned}&\text{data dProd } (A\ B : \text{Set}) : \text{Set where}\\ &\quad \text{p} : A \to B \to \text{dProd } A\ B\end{aligned}$$

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

- In case one needs to make them explict, one can do so:

$$\begin{aligned}c &: \text{dProd } A\ B\\ c &= \text{p } \{A\}\ \{B\}\ a\ b\end{aligned}$$

Hidden Arguments in dProd

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

$$\begin{aligned} c &: \mathrm{dProd}\ A\ B \\ c &= \mathrm{p}\ \{A\}\ a\ b \end{aligned}$$

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

$$\begin{aligned} c &: \mathrm{dProd}\ A\ B \\ c &= \mathrm{p}\ \{\_\}\ \{B\}\ a\ b \end{aligned}$$

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

Decomposing Record Type

- Let

$$\begin{array}{lcl} D & : & \mathrm{Set} \\ D & = & \mathrm{rProd}\ (\mathrm{dProd}\ A\ B)\ C \end{array}$$

- Assume we want to define $f : D \to A$ 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

$$\begin{array}{l} f : D \to A \\ f\ d\ \mathrm{with}\ \mathrm{rProd.first}\ d \\ f\ d \mid \mathrm{p}\ a\ b = a \end{array}$$

We can abbreviate this by using:

$$\begin{array}{l} f : D \to A \\ f\ d\ \mathrm{with}\ \mathrm{rProd.first}\ d \\ \ldots \mid \mathrm{p}\ a\ b = a \end{array}$$

Decomposing Record Type

$f : D \to A$
$f\ d$ with $\mathrm{rProd.first}\ d$
$f\ d \mid \mathrm{p}\ a\ b = a$

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

Longer Example

- As an example we want to define in Agda, depending on
  - $A, B, C, D : \mathrm{Set}$,
  - $ab : A \times B$
  - $a\text{-}c : A \to C$,
  - $b\text{-}d : B \to D$

  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

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

Longer Example

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

$$\begin{aligned}&\mathrm{postulate}\ A : \mathrm{Set}\\&\mathrm{postulate}\ B : \mathrm{Set}\\&\mathrm{postulate}\ C : \mathrm{Set}\\&\mathrm{postulate}\ D : \mathrm{Set}\end{aligned}$$

Longer Example

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

$$
\begin{array}{l}
\text{record } AB \quad : \text{Set where} \\
\quad \text{field} \\
\qquad a : A \\
\qquad b : B \\
\text{record } CD \quad : \text{Set where} \\
\quad \text{field} \\
\qquad c : C \\
\qquad d : D
\end{array}
$$

Longer Example

- The goal to be solved is as follows:

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

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

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

    to $a : A$ and obtain an element

    $$c : C \ .$$

  - And we apply

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

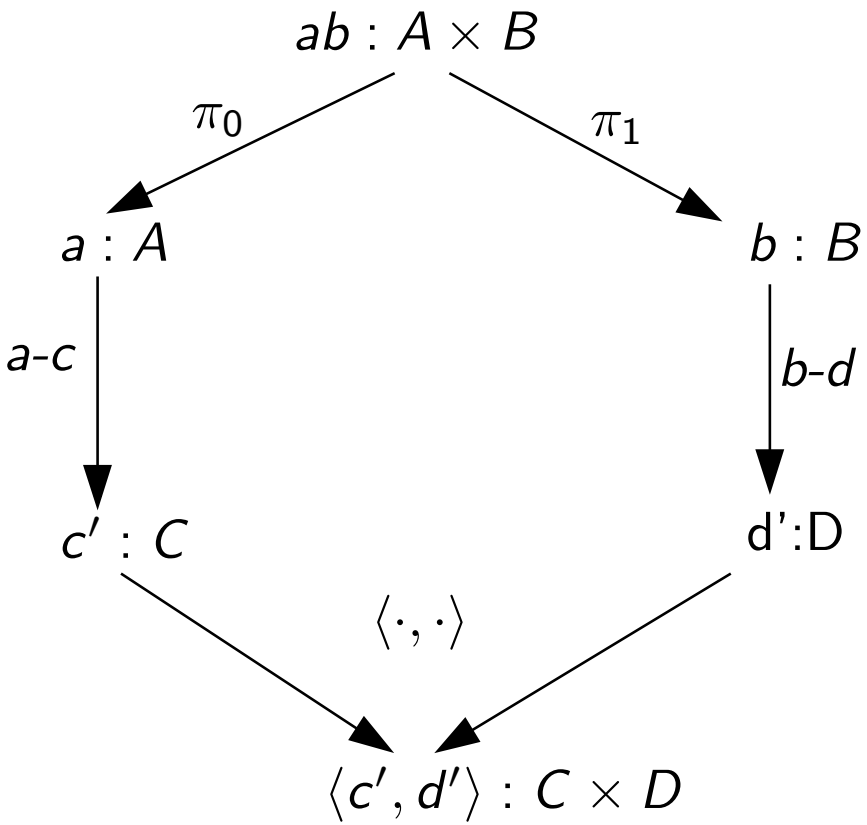
    to $b : B$ and obtain an element

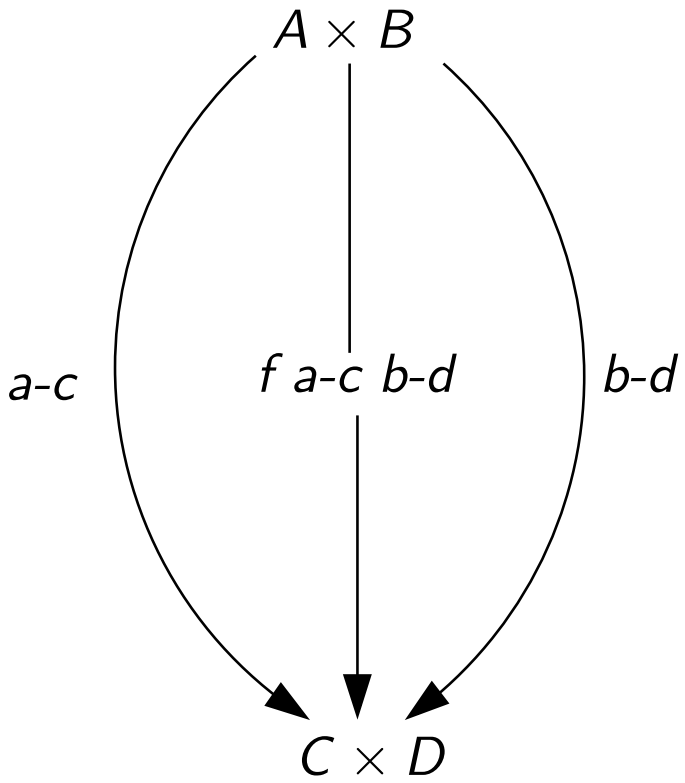
    $$d : D \ .$$

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

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

$$
\begin{array}{l}
f : AB \to (A \to C) \to (B \to D) \to CD \\
f\ ab\ a\text{-}c\ b\text{-}d = \mathrm{let}\ a' \ : \ A \\
\qquad\qquad\qquad\qquad\qquad\quad a' \ = \ AB.a\ ab \\
\qquad\qquad\qquad\qquad\qquad\quad b' \ : \ B \\
\qquad\qquad\qquad\qquad\qquad\quad b' \ = \ AB.b\ ab \\
\qquad\qquad\qquad\qquad\qquad\quad c' \ : \ C \\
\qquad\qquad\qquad\qquad\qquad\quad c' \ = \ a\text{-}c\ a' \\
\qquad\qquad\qquad\qquad\qquad\quad d' \ : \ D \\
\qquad\qquad\qquad\qquad\qquad\quad d' \ = \ b\text{-}d\ b' \\
\mathrm{in\ record}\{c = c'; d = d'\}
\end{array}
$$

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

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

- 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

$$\begin{aligned}&\text{data Person} : \text{Set where}\\ &\quad \text{person} : \text{Gender} \to \text{Name} \to \text{Person}\end{aligned}$$

Concrete Products

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

$$\begin{aligned} &\text{gender} : \text{Person} \to \text{Gender} \\ &\text{gender}\ (\text{person}\ g\ n) = g \end{aligned}$$

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:

$$
\begin{array}{l}
\mathrm{record}\ \_\wedge\mathrm{r}\_\ (A\ B : \mathrm{Set}) : \mathrm{Set}\ \mathrm{where} \\
\quad \mathrm{field} \\
\qquad \mathrm{and1} \ : \ A \\
\qquad \mathrm{and2} \ : \ B
\end{array}
$$

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

Conjunction in Agda

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

$$\mathrm{data}\ \_\wedge\mathrm{d}\_\ (A\ B : \mathrm{Set}) : \mathrm{Set}\ \mathrm{where}$$
$$\mathrm{and} : A \to B \to A \wedge\mathrm{d}\ B$$

- 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 an abbreviation by using **C-q** before adding any space-like character after "andd".

Customising Agda with Abbreviation Mode

- 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 Mode

- 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 Mode

- 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 Mode

- 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)))
```

- You need as well to run once `write-abbrev-file`.

Example

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

Example (Conjunction)

- We prove $A \wedge B \to B \wedge A$ (see **exampleproofproplogic6.agda**):

$$
\begin{array}{lcl}
\mathrm{Lemma} & : & \mathrm{Set} \\
\mathrm{Lemma} & = & A \wedge \mathrm{r}\ B \to B \wedge \mathrm{r}\ A \\
 & & \\
\mathrm{lemma} & : & \mathrm{Lemma} \\
\mathrm{lemma}\ ab & = & \mathrm{record}\{\mathrm{and1} = \_\wedge \mathrm{r}\_.\mathrm{and2}\ ab; \\
 & & \qquad\qquad \mathrm{and2} = \_\wedge \mathrm{r}\_.\mathrm{and1}\ ab\}
\end{array}
$$

Example (Conjunction)

$$
\begin{array}{lcl}
\mathrm{Lemma}' & : & \mathrm{Set} \\
\mathrm{Lemma}' & = & A \wedge \mathrm{d}\, B \to B \wedge \mathrm{d}\, A \\
 & & \\
\mathrm{lemma}' & : & \mathrm{Lemma}' \\
\mathrm{lemma}'\ (\mathrm{and}\ a\ b) & = & \mathrm{and}\ b\ a
\end{array}
$$

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 $(A \wedge\mathrm{r}\ B) \wedge\mathrm{r}\ C$ or $A \wedge\mathrm{r}\ (B \wedge\mathrm{r}\ C)$.
  - If one adds

    $$\mathrm{infixl}\ 30\ \_\wedge\mathrm{r}\_$$

    one can write

    $$A \wedge\mathrm{r}\ B \wedge\mathrm{r}\ C$$

    for

    $$(A \wedge\mathrm{r}\ B) \wedge\mathrm{r}\ C$$

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:

$$
\begin{array}{l}
\text{record And3r } (A\ B\ C : \text{Set}) \quad : \quad \text{Set where} \\
\quad \text{field} \\
\qquad \begin{array}{lcl} \text{and1} & : & A \\ \text{and2} & : & B \\ \text{and3} & : & C \end{array}
\end{array}
$$

Conjunction with more Conjuncts

- Definition of the ternary and using "data":

$$\begin{aligned}&\text{data And3d } (A\ B\ C : \text{Set}) : \text{Set where}\\ &\quad \text{and3d} : A \to B \to C \to \text{And3d } A\ B\ C\end{aligned}$$

See **exampleproofproplogic5.agda**

4 (e) The λ-Calculus 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

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

- 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 $x + 0 \longrightarrow_{\mathrm{Rule}} x$.
  - $\lambda x.x + 0$ 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 $x + 0 \longrightarrow_{\mathrm{Rule}} x$ 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 $s \longrightarrow_{\mathrm{Rule}} t$ 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

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

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

- Then replace this rule by

$$\lambda x_1, \ldots, x_n.s \longrightarrow_{\mathrm{Rule}'} \lambda x_1, \ldots, x_n.t \ .$$

Proof

- We write in the following $\vec{x}$ for $x_1, \ldots, x_n$.
- 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 $r =_\beta r' \longrightarrow_{\mathrm{Rule}'} u' =_\beta u$, 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 $\lambda\vec{x}.s \longrightarrow_{\mathrm{Rule}'} \lambda\vec{x}.t$, then $r \longrightarrow u$ in the previous system by using the rule $s \longrightarrow_{\mathrm{Rule}} t$
  - 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

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

by

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

- That

$$\mathrm{S}\ (0 + \mathrm{S}\ 0) \longrightarrow \mathrm{S}\ (\mathrm{S}\ (0 + 0)) \longrightarrow \mathrm{S}\ (\mathrm{S}\ 0)$$

becomes in the new system

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

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

$$\dfrac{\dfrac{\dfrac{x{:}\mathrm{nat} \Rightarrow \_+\_ : \mathrm{nat}\to\mathrm{nat}\to\mathrm{nat} \qquad x{:}\mathrm{nat} \Rightarrow x{:}\mathrm{nat}}{x{:}\mathrm{nat} \Rightarrow \_+\_\ x{:}\mathrm{nat}\to\mathrm{nat}}\,(\mathrm{Ap}) \qquad x{:}\mathrm{nat} \Rightarrow x{:}\mathrm{nat}}{x : \mathrm{nat} \Rightarrow \_ + \_\ x\ x : \mathrm{nat}}\,(\mathrm{Ap})}{(\lambda x^{\mathrm{nat}}.x + x) : \mathrm{nat} \to \mathrm{nat}}\,(\mathrm{Abs})$$

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

Example

Then we have

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

Extended Typed λ-Calculus

- Reduction rules should now be of the form $\Gamma \Rightarrow s \longrightarrow_{\mathrm{Rule}} t : \sigma$ (instead of $s \longrightarrow_{\mathrm{Rule}} t$) 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 $c \longrightarrow_{\mathrm{Rule}} t : \sigma$ where c is a constant and therefore t a closed term.

Extended Typed λ-Calculus

- Instantiations of a rule $\Gamma \Rightarrow s \longrightarrow_{\mathrm{Rule}} t : \sigma$ 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 $\_ + \_ : \mathrm{nat} \to \mathrm{nat} \to \mathrm{nat}$ (written infix, i.e. $r + s$ for $\_ + \_\ r\ s$),
  - and $\mathrm{double} : \mathrm{nat} \to \mathrm{nat}$.
  - and the reduction rule $\mathrm{double} \longrightarrow (\lambda x^{\mathrm{nat}}.x + x) : \mathrm{nat} \to \mathrm{nat}$.

Example

Then we have

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

4 (f) Finite Sets and Decidable Formulae

- 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

$$\mathrm{Bool}$$

  to the set of ground types.
- We add constants

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

- Here tt stands for true, ff for false.

$\mathrm{Case}^{\sigma}_{\mathrm{Bool}}$

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

$$\mathit{case}_{\mathrm{tt}} : \sigma \qquad \mathit{case}_{\mathrm{ff}} : \sigma$$

  - Then we want to have that

$$\mathrm{Case}^{\sigma}_{\mathrm{Bool}}\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}} : \mathrm{Bool} \to \sigma$$

  - And we want that

$$\begin{array}{lcl} \mathrm{Case}^{\sigma}_{\mathrm{Bool}}\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ \mathrm{tt} & = & \mathit{case}_{\mathrm{tt}} \\ \mathrm{Case}^{\sigma}_{\mathrm{Bool}}\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ \mathrm{ff} & = & \mathit{case}_{\mathrm{ff}} \end{array}$$

If then else

$$\begin{array}{lcl} \mathrm{Case}^{\sigma}_{\mathrm{Bool}}\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ \mathrm{tt} & = & \mathit{case}_{\mathrm{tt}} \\ \mathrm{Case}^{\sigma}_{\mathrm{Bool}}\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ \mathrm{ff} & = & \mathit{case}_{\mathrm{ff}} \end{array}$$

- $\mathrm{Case}^{\sigma}_{\mathrm{Bool}}\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ b$ corresponds to

$$\text{If } b \text{ then } \mathit{case}_{\mathrm{tt}} \text{ else } \mathit{case}_{\mathrm{ff}} \quad :$$

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

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

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

$$\mathrm{Case}^{\sigma}_{\mathrm{Bool}} : \sigma \to \sigma \to \mathrm{Bool} \to \sigma$$

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

$$\mathrm{Case}^{\sigma}_{\mathrm{Bool}}\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}} : \mathrm{Bool} \to \sigma$$

- The equalities are achieved by adding reductions

$$\begin{array}{lcl} \mathrm{Case}^{\sigma}_{\mathrm{Bool}}\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ \mathrm{tt} & \longrightarrow & \mathit{case}_{\mathrm{tt}} \\ \mathrm{Case}^{\sigma}_{\mathrm{Bool}}\ \mathit{case}_{\mathrm{tt}}\ \mathit{case}_{\mathrm{ff}}\ \mathrm{ff} & \longrightarrow & \mathit{case}_{\mathrm{ff}} \end{array}$$

Example: Boolean Conjunction

- We define Boolean valued conjunction

$$\_\wedge_{\mathrm{Bool}}\_ : \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool} \ .$$

- We write
  - $\_\wedge_{\mathrm{Bool}}\_$ for function symbol,
  - $\wedge_{\mathrm{Bool}}$ for the symbol, written infix,

  so $b \wedge_{\mathrm{Bool}} c$ stands for $\_\wedge_{\mathrm{Bool}}\_\ b\ c$.
- 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 $A \wedge B$.
  - $b \wedge_{\mathrm{Bool}} c$ will form the Boolean value corresponding to the conjunction of b and c.

Truth Table for $\wedge_{\mathrm{Bool}}$

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

$\wedge_{\mathrm{Bool}}$	ff	tt
ff	ff	ff
tt	ff	tt

- So we have

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

Example: $\wedge_{\mathrm{Bool}}$

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

Example: $\wedge_{\mathrm{Bool}}$

$\_\wedge_{\mathrm{Bool}}\_ : \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool}.$

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

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

  for some term t.

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

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

  for some e, f.

Example: $\wedge_{\mathrm{Bool}}$

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

- For conjunction we have:
  - We have seen that
$$\mathrm{tt} \wedge_{\mathrm{Bool}} c = c$$
    - So the if-case e above is c.
  - Furthermore
$$\mathrm{ff} \wedge_{\mathrm{Bool}} c = \mathrm{ff}$$
    - So the else-case f above is ff.

Example: $\wedge_{\mathrm{Bool}}$

- In total we define therefore

$$\begin{array}{lcl} \_\wedge_{\mathrm{Bool}}\_ & = & \lambda(b, c : \mathrm{Bool}).\mathrm{Case}^{\mathrm{Bool}}_{\mathrm{Bool}}\ c\ \mathrm{ff}\ b \\ & : & \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool} \end{array}$$

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

$$\mathrm{A}_k^n : \mathrm{Fin}_n$$

  - Informally we will have have

$$\mathrm{Fin}_n = \{\mathrm{A}_0^n, \mathrm{A}_1^n, \ldots, \mathrm{A}_{n-1}^n\}$$

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

$$\begin{array}{lcl} \mathrm{Fin}_2 & = & \{\mathrm{A}_0^2, \mathrm{A}_1^2\} \\ \mathrm{Fin}_1 & = & \{\mathrm{A}_0^1\} \\ \mathrm{Fin}_0 & = & \emptyset \end{array}$$

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 $case_i : \sigma$ for $i = 0, \ldots, n-1$.
- Then we want

$$\mathrm{Case}_n^\sigma \; case_0 \; \cdots \; case_{n-1} : \mathrm{Fin}_n \to \sigma$$

- And we want

$$\mathrm{Case}_n^\sigma \; case_0 \; \cdots \; case_{n-1} \; \mathrm{A}_i^n = case_i$$

Constants Case_n^σ

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

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

  - Then from $case_i : \sigma$ we obtain

$$\mathrm{Case}_n^\sigma\ case_0\ \cdots\ case_{n-1} : \mathrm{Fin}_n \to \sigma$$

- Furthermore we add the reduction rules

$$\mathrm{Case}_n^\sigma\ case_0\ \cdots\ case_{n-1}\ \mathrm{A}_i^n \longrightarrow case_i$$

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

$$\mathrm{Eq}_{\mathrm{n,Bool}} : \mathrm{Fin}_n \to \mathrm{Fin}_n \to \mathrm{Bool}$$

    s.t.

$$\begin{aligned} \mathrm{Eq}_{\mathrm{n,Bool}}\ \mathrm{A}_n^i\ \mathrm{A}_n^i &= \mathrm{tt} \\ \mathrm{Eq}_{\mathrm{n,Bool}}\ \mathrm{A}_n^i\ \mathrm{A}_n^j &= \mathrm{ff} \quad \text{for } i \neq j \end{aligned}$$

    - $\mathrm{Eq}_{\mathrm{n,Bool}}$ 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

$$\mathrm{Fin}_n$$

with $n = 2$:

$$\begin{array}{rcll} \mathrm{Bool} & := & \mathrm{Fin}_2 & \\ \mathrm{tt} & := & \mathrm{A}_0^2 & : \ \mathrm{Bool} \\ \mathrm{ff} & := & \mathrm{A}_1^2 & : \ \mathrm{Bool} \\ \mathrm{Case}^{\sigma}_{\mathrm{Bool}} & := & \mathrm{Case}^{\sigma}_2 & : \ \sigma \to \sigma \to \mathrm{Bool} \to \sigma \end{array}$$

Rules for $\top$

- $\top$ (pronounced "top") is the special case

$$\mathrm{Fin}_n$$

for $n = 1$
(we write true for A_0^1):

  - So we have a type $\top := \mathrm{Fin}_1$,
  - $\mathrm{true} := \mathrm{A}_0^1 : \top$, (pronounced as "top", typed in in Agda as \top)
  - $\mathrm{Case}_\top^\sigma := \mathrm{Case}_1^\sigma : \sigma \to \top \to \sigma$.
  - $\mathrm{case}_\top^\sigma\ a\ \mathrm{true} \longrightarrow a$.

$\top$ 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

$$\mathrm{true} \ ,$$

    since it is the proof of the always true formula.
  - Example: we have

$$\lambda x^{A}.\mathrm{true} : A \to \top$$

Rules for $\bot$

$\bot$ (pronounced "bottom", typed in in Agda as \bot) is the special case

$$\mathrm{Fin}_n$$

for $n = 0$:

- $\bot := \mathrm{Fin}_0$.
- $\bot$ 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

$$\mathrm{Case}^{\sigma}_{\bot} : \bot \to \sigma$$

  - We have no reduction rules.

$\bot$

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

$\bot$

- $\mathrm{Case}^{\sigma}_{\bot}$ expresses: **from an element f of $\bot$ we obtain an element of any set**.
  - Correct, since there is no element of $\bot$.
  - Considered as a formula, $\mathrm{Case}^{\sigma}_{\bot}$ means: from a proof of $\bot$ we obtain a proof of every other formula.
  - I.e. it means $\bot$ 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 $\bot \to A$:

$$\mathrm{Case}^{A}_{\bot} : \bot \to A \ .$$

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

4 (g) Finite Sets and Decidable Formulae 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

$$\begin{array}{lcl} \neg\mathrm{Bool} & : & \mathrm{Bool} \to \mathrm{Bool} \\ \neg\mathrm{Bool}\ \mathrm{tt} & = & \mathrm{ff} \\ \neg\mathrm{Bool}\ \mathrm{ff} & = & \mathrm{tt} \end{array}$$

- The above is already the Agda code defining $\neg\mathrm{Bool}$.
  examplenegbool.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

$$\begin{array}{lcl} \text{is}-\text{red} & : & \text{Colour} \to \text{Bool} \\ \text{is}-\text{red red} & = & \text{tt} \\ \text{is}-\text{red \_} & = & \text{ff} \end{array}$$

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

$$\text{is}-\text{red red}$$

matches both lines

$$\begin{array}{lcl} \text{is}-\text{red red} & = & \text{tt} \\ \text{is}-\text{red \_} & = & \text{ff} \end{array}$$

Finite Sets in Agda

$$
\begin{array}{lcl}
\text{is-red} & : & \text{Colour} \to \text{Bool} \\
\text{is-red red} & = & \text{tt} \\
\text{is-red \_} & = & \text{ff}
\end{array}
$$

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

$$\text{is-red red} = \text{tt}$$

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

$$\text{is-red blue} = \text{is-red green} = \text{ff}$$

$\top$ in Agda

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

$$\begin{array}{l}\text{data } \top : \text{Set where}\\ \quad \text{true} : \top\end{array}$$

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

$$\begin{array}{lcl} g & : & \top \to \text{Bool} \\ g\ \text{true} & = & \text{tt} \end{array}$$

$\top$ in Agda

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

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

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

$$\begin{array}{lcl} \mathrm{true}' & : & \top' \\ \mathrm{true}' & = & \mathrm{record}\{\ \} \end{array}$$

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

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

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

$\bot$ 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:

$$g : \bot \to \mathrm{Bool}$$
$$g\ ()$$

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

¬ in Agda

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

$$\begin{array}{lcl} \neg & : & \mathrm{Set} \to \mathrm{Set} \\ \neg\, A & = & A \to \bot \end{array}$$

Example for the Use of $\bot$

- Assume the **type of trees**:

$$\begin{array}{l}\mathrm{data\ Tree : Set\ where}\\ \quad \begin{array}{lcl}\mathrm{oak} & : & \mathrm{Tree}\\ \mathrm{pine} & : & \mathrm{Tree}\\ \mathrm{spruce} & : & \mathrm{Tree}\end{array}\end{array}$$

- We can now define

$$\begin{array}{lcl}\mathrm{IsConifer} & : & \mathrm{Tree} \to \mathrm{Set}\\ \mathrm{IsConifer\ oak} & = & \bot\\ \mathrm{IsConifer\ \_} & = & \top\end{array}$$

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

Example for the Use of ⊥

- 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":

$$
\begin{array}{lll}
f : (t : \mathrm{Tree}) \to \mathrm{IsConifer}\ t \to A \\
f \quad \mathrm{pine} \quad \_ \quad = \quad \{!\ \ !\} \\
f \quad \mathrm{spruce} \quad \_ \quad = \quad \{!\ \ !\} \\
f \quad \mathrm{oak} \quad ()
\end{array}
$$

- 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 $\bot$

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

$$\begin{array}{l}\mathrm{data\ Stack}\ (A : \mathrm{Set}) : \mathrm{Set\ where}\\ \quad \begin{array}{lcl}\mathrm{empty} & : & \mathrm{Stack}\ A\\ \mathrm{push} & : & A \to \mathrm{Stack}\ A \to \mathrm{Stack}\ A\end{array}\end{array}$$

- We can then introduce a predicate $\mathrm{NonEmpty}$ expressing that the stack is nonempty:

$$\begin{array}{lcl}\mathrm{NonEmpty} : \{A : \mathrm{Set}\} \to \mathrm{Stack}\ A \to \mathrm{Set}\\ \mathrm{NonEmpty\ empty} & = & \bot\\ \mathrm{NonEmpty\ (push\ \_\ \_)} & = & \top\end{array}$$

Example 2 for the Use of ⊥

- Now we can define

$$
\begin{array}{lll}
\mathrm{top} : \{A : \mathrm{Set}\} \to (s : \mathrm{Stack}\ A) \to \mathrm{NonEmpty}\ s \to A \\
\mathrm{top} \quad \mathrm{empty} \quad\quad () \\
\mathrm{top} \quad (\mathrm{push}\ a\ \_) \quad \_ \quad = \quad a
\end{array}
$$

  (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

$$\begin{array}{lcl} \mathrm{Atom} & : & \mathrm{Bool} \to \mathrm{Set} \\ \mathrm{Atom\ tt} & = & \top \\ \mathrm{Atom\ ff} & = & \bot \end{array}$$

Atomic Formulae

$$\begin{array}{lcl} \mathrm{Atom} & : & \mathrm{Bool} \to \mathrm{Set} \\ \mathrm{Atom\ tt} & = & \top \\ \mathrm{Atom\ ff} & = & \bot \end{array}$$

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

$$\begin{array}{lcl} \mathrm{Eq_{n,Bool}} & : & \mathrm{Fin}_n \to \mathrm{Fin}_n \to \mathrm{Bool} \\ \mathrm{Eq_{n,Bool}}\ \mathrm{A}^i_n\ \mathrm{A}^i_n & = & \mathrm{tt} \\ \mathrm{Eq_{n,Bool}}\ \_\ \_ & = & \mathrm{ff} \end{array}$$

Example

- For instance in case of the set

$$
\begin{array}{l}
\mathrm{data\ Colour : Set\ where} \\
\quad \begin{array}{lcl} \mathrm{blue} & : & \mathrm{Colour} \\ \mathrm{red} & : & \mathrm{Colour} \\ \mathrm{green} & : & \mathrm{Colour} \end{array}
\end{array}
$$

we define

$$
\begin{array}{l}
\mathrm{eqColourBool : Colour \to Colour \to Bool} \\
\begin{array}{llll} \mathrm{eqColourBool} & \mathrm{blue} & \mathrm{blue} & = \ \mathrm{tt} \\ \mathrm{eqColourBool} & \mathrm{red} & \mathrm{red} & = \ \mathrm{tt} \\ \mathrm{eqColourBool} & \mathrm{green} & \mathrm{green} & = \ \mathrm{tt} \\ \mathrm{eqColourBool} & \_ & \_ & = \ \mathrm{ff} \end{array}
\end{array}
$$

Example

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

$$\begin{array}{lcl} \_==\_ & : & \mathrm{Colour} \to \mathrm{Colour} \to \mathrm{Set} \\ c == c' & = & \mathrm{Atom}\ (\mathrm{eqColourBool}\ c\ c') \end{array}$$

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

Example

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

$$
\begin{array}{lllll}
\_=='\_ : \mathrm{Colour} \to \mathrm{Colour} \to \mathrm{Set} & & & & \\
\mathrm{blue} & ==' & \mathrm{blue} & = & \top \\
\mathrm{red} & ==' & \mathrm{red} & = & \top \\
\mathrm{green} & ==' & \mathrm{green} & = & \top \\
\_ & ==' & \_ & = & \bot
\end{array}
$$

exampleColourEquality.agda

Example 2

- Remember the definition of Boolean valued negation in Agda:

$$\begin{array}{lcl} \neg\mathrm{Bool} : \mathrm{Bool} \to \mathrm{Bool} \\ \neg\mathrm{Bool} \quad \mathrm{tt} \quad = \quad \mathrm{ff} \\ \neg\mathrm{Bool} \quad \mathrm{ff} \quad = \quad \mathrm{tt} \end{array}$$

- We show

$$\mathrm{Atom}\ (\neg\mathrm{Bool}\ b) \to \neg\ (\mathrm{Atom}\ b)$$

- Remember that we defined

$$\begin{array}{lcl} \neg \quad : \quad \mathrm{Set} \to \mathrm{Set} \\ \neg\ A \quad = \quad A \to \bot \end{array}$$

Example 2

- So our lemma is

$$\begin{array}{lcl} \text{Lemma} & : & \text{Set} \\ \text{Lemma} & = & (b : \text{Bool}) \to \text{Atom}\ (\neg\text{Bool}\ b) \to \neg\ (\text{Atom}\ b) \end{array}$$

- Since $\neg A = A \to \bot$ this is equivalent to

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

- We need to show

$$\begin{array}{lcl} \text{lemma} & : & \text{Lemma} \\ \text{lemma}\ b\ p\ q & = & \{!\ \ !\} \end{array}$$

Example 2

$$
\begin{array}{lcl}
\text{Lemma} & : & \text{Set} \\
\text{Lemma} & = & (b : \text{Bool}) \to \text{Atom}\ (\neg\text{Bool}\ b) \to \text{Atom}\ b \to \bot \\
\text{lemma} & : & \text{Lemma} \\
\text{lemma}\ b\ p\ q & = & \{!\ \ !\}
\end{array}
$$

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

$$
\begin{array}{lcl}
\text{lemma} & : & \text{Lemma} \\
\text{lemma tt}\ p\ q & = & \{!\ \ !\} \\
\text{lemma ff}\ p\ q & = & \{!\ \ !\}
\end{array}
$$

Example 2

$$
\begin{array}{lcl}
\text{Lemma} & : & \text{Set} \\
\text{Lemma} & = & (b : \text{Bool}) \to \text{Atom}\ (\neg\text{Bool}\ b) \to \text{Atom}\ b \to \bot \\
\text{lemma} & : & \text{Lemma} \\
\text{lemma}\ \text{tt}\ p\ q & = & \{!\ \ !\} \\
\text{lemma}\ \text{ff}\ p\ q & = & \{!\ \ !\}
\end{array}
$$

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

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

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

Example 2

$$
\begin{array}{lcl}
\text{Lemma} & : & \text{Set} \\
\text{Lemma} & = & (b : \text{Bool}) \to \text{Atom}\ (\neg\text{Bool}\ b) \to \text{Atom}\ b \to \bot \\
\text{lemma} & : & \text{Lemma} \\
\text{lemma}\ \text{tt}\ ()\ q & & \\
\text{lemma}\ \text{ff}\ p\ q & = & \{!\ \ !\}
\end{array}
$$

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

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

- 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

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

  and show that $\mathrm{Atom}\ (b\ \rightarrow\mathrm{Bool}\ b')$ implies $\mathrm{Atom}\ b \rightarrow \mathrm{Atom}\ b'$.
  - The other direction can be shown as well.

Example 3

- Classically $A \to B$ is true iff A is false or B is true.
  - Something false implies everything, so $A \to B$ is true if A is false.
  - If A is true, then $A \to B$ is true if B is true.
- So we have $A \to B$ 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

$$
\begin{array}{lll}
\_{\to}\mathrm{Bool}\_ : \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool} & & \\
\mathrm{tt} \quad {\to}\mathrm{Bool} \quad \mathrm{ff} & = & \mathrm{ff} \\
\_ \quad\quad {\to}\mathrm{Bool} \quad \_ & = & \mathrm{tt}
\end{array}
$$

Example 3

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

$$\begin{aligned} &\mathrm{Lemma} \;:\; \mathrm{Set} \\ &\mathrm{Lemma} \;=\; (b\ b' : \mathrm{Bool}) \to \mathrm{Atom}\ (b \to_{\mathrm{Bool}} b') \to \mathrm{Atom}\ b \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \mathrm{Atom}\ b' \end{aligned}$$

$$\begin{aligned} &\mathrm{lemma} \;:\; \mathrm{Lemma} \\ &\mathrm{lemma}\ b\ b'\ b{\to}b'\ \mathit{btrue} = \{!\ \ !\} \end{aligned}$$

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

Example 3

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

$$\text{lemma}\ b\ b'\ \ b{\to}b'\ \ \mathit{btrue} = \{!\ \ !\}$$

- If $b = \text{ff}$, then

$$\mathit{btrue} : \text{Atom}\ b = \bot$$

  which matches the empty pattern.

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

$$b{\to}b' : \text{Atom}\ (b\ \to\text{Bool}\ \ b') = \text{Atom}\ \text{ff} = \bot$$

  which again matches the empty pattern.

Example 3

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

$$\text{lemma}\ b\ b'\ \ b{\to}b'\ \ \mathit{btrue} = \{!\ \ !\}$$

- 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′ : Bool) → Atom (b →Bool b′)→ Atom b
                                            → 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
      $$\mathrm{Eq}_{\mathbb{N},\mathrm{Bool}} : \mathbb{N} \to \mathbb{N} \to \mathrm{Bool}$$
      which decides it.
    - The equality on **functions N $\to$ 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

$$\mathrm{RealWorldState} : \mathrm{Set}$$

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

$$\mathrm{ControlState} : \mathrm{Set}$$

  - e.g. the set of states a railway controler 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

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

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

Decidable Predicates (Cont.)

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

Decidable Predicates (Cont.)

- So if we have a proof

$$p : (a : \mathrm{ControlState}) \to \mathrm{Safe}\ (\mathrm{control}{\to}\mathrm{realWorld}\ s)$$

  we have shown that **the system is safe** w.r.t. the safety property expressed by $\mathrm{safe}_{\mathrm{Bool}}$.