4. The $\lambda$-Calc. with Prod. and Conj.

(4. The $\lambda$-Calculus with Products and Conjunction)

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

(a) The Typed $\lambda$-Calc. with Products

One can expand the set of $\lambda$-types and $\lambda$-terms as follows:

- Types are defined as before, but we have additionally:
  - If $\sigma$, $\tau$ are types, so is $\sigma \times \tau$.

Example (Products)

Assume we have some extra ground types

\[
\begin{align*}
\text{Name} & := \text{String} \\
\text{Gender} & := \{\text{female, male}\}
\end{align*}
\]

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 $\text{Term}$ of terms, representing functions

\[
\text{Int} \rightarrow \text{Int}.
\]

The set of terms $\text{Term}$ together with the function, they denote, is given as

\[
\text{Term} \times (\text{Int} \rightarrow \text{Int})
\]
Products

- The set of typed-\(\lambda\)-terms are defined as before but we have:
  - If \(s : \sigma, t : \tau\) then \(\langle s, t \rangle : \sigma \times \tau\):
    \[
    \frac{\Gamma \Rightarrow s : \sigma \quad \Gamma \Rightarrow t : \tau}{\Gamma \Rightarrow \langle s, t \rangle : \sigma \times \tau} \quad \text{(Pair)}
    \]
  - If \(s : \sigma \times \tau\), then \(\pi_0(s) : \sigma\) and \(\pi_1(s) : \tau\):
    \[
    \frac{\Gamma \Rightarrow s : \sigma \times \tau}{\Gamma \Rightarrow \pi_0(s) : \sigma} \quad \text{(Proj}_0\text{)}
    \]
    \[
    \frac{\Gamma \Rightarrow s : \sigma \times \tau}{\Gamma \Rightarrow \pi_1(s) : \tau} \quad \text{(Proj}_1\text{)}
    \]

- \(\beta\)-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 \(\beta\)-redexes of the term in question.

- In addition we have the \(\beta\)-redexes \((\lambda x.t)\ s\) of the \(\lambda\)-calculus with \(\to\).

- \(\beta\)-reduction for the typed \(\lambda\)-calculus with products includes both \(\beta\)-reduction for functions and \(\beta\)-reduction for pairs.

Example

- We show
  \[
  \lambda x.0 \times (0 \to 0 \to 0).\pi_0(x) \to 0 \to 0
  \]
  \[
  \lambda x.0 \times (0 \to 0 \to 0).\pi_0(\langle y, \lambda z.0.\lambda v.0 \rangle) \to 0 \to 0
  \]
  \[
  \lambda y.0 \times (0 \to 0 \to 0).\pi_0(\langle y, z, v \rangle) \to 0 \to 0
  \]
  \[
  \lambda y.0 \times (0 \to 0 \to 0).\pi_0(\langle y, z, v \rangle) \to 0 \to 0
  \]
  \[
  \lambda y.0 \times (0 \to 0 \to 0).\pi_0(\langle y, z, v \rangle) \to 0 \to 0
  \]

Example

- \((\lambda x.0 \times (0 \to 0 \to 0).\pi_0(x)) \to \langle y, z, v \rangle\)
- \(\to \beta \pi_0(\langle y, z, v \rangle)\)
- \(\to \beta y.0\)
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 $s_0 : \sigma_0, \ldots, s_n : \sigma_n$
$$\langle s_0, \ldots, s_n \rangle := \langle \cdots \langle \langle s_0, s_1 \rangle, s_2 \rangle, \cdots, s_n \rangle : \sigma_0 \times \cdots \times \sigma_n$$

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

One can easily define corresponding projections
$$\pi_i^n : (\sigma_0 \times \cdots \times \sigma_{n-1}) \rightarrow \sigma_i, \text{s.t.}$$
$$\pi_i^n(\langle s_0, \ldots, s_{n-1} \rangle) = \beta s_i .$$

For instance in case $n = 3$ we need
$$\pi_1^3(\langle s_0, s_1, s_2 \rangle) = \pi_1^3(\langle s_0, s_1, s_2 \rangle) = s_i$$

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

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

One can easily define corresponding projections
$$\pi_i^n : (\sigma_0 \times \cdots \times \sigma_{n-1}) \rightarrow \sigma_i, \text{s.t.}$$
$$\pi_i^n(\langle s_0, \ldots, s_{n-1} \rangle) = \beta s_i .$$

For instance in case $n = 3$ we need
$$\pi_1^3(\langle s_0, s_1, s_2 \rangle) = \pi_1^3(\langle s_0, s_1, s_2 \rangle) = s_i$$

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

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

One can easily define corresponding projections
$$\pi_i^n : (\sigma_0 \times \cdots \times \sigma_{n-1}) \rightarrow \sigma_i, \text{s.t.}$$
$$\pi_i^n(\langle s_0, \ldots, s_{n-1} \rangle) = \beta s_i .$$

For instance in case $n = 3$ we need
$$\pi_1^3(\langle s_0, s_1, s_2 \rangle) = \pi_1^3(\langle s_0, s_1, s_2 \rangle) = s_i$$

$\eta$-Expansion for Products

If we have a product $r : \sigma \times \tau$, then its projections are $\beta$-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 $\eta$-rule expresses that subterms $t : \sigma \times \tau$ can be $\eta$-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 \( t \rightarrow \langle \pi_0(t), \pi_1(t) \rangle \), we would get the following circular reduction:

\[
\begin{align*}
t & \rightarrow \langle \pi_0(t), \pi_1(t) \rangle \\
& \equiv \langle \pi_0(\langle r_0, r_1 \rangle), \pi_1(\langle r_0, r_1 \rangle) \rangle \\
\beta & \langle r_0, r_1 \rangle \\
& \equiv t
\end{align*}
\]

**η-Expansion for Products**

η-expansion for products is the rule which allows to replace in a typed \( \lambda \)-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 \( \lambda \)-calculus with products includes both η-expansion for functions and for pairs.

---

**Example**

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

\[
(\lambda f^{(o\times o)\rightarrow o}.\lambda x^{o\times o}.f \ x) \ g
\]

\[
\beta \rightarrow \lambda x^{o\times o}.g \ x
\]

\[
\eta \rightarrow \lambda x^{o\times o}.g \ \langle \pi_0(x), \pi_1(x) \rangle
\]

\( \lambda x^{o\times o}.g \ \langle \pi_0(x), \pi_1(x) \rangle \) is therefore the \( \beta.\eta \)-**normal form** of

\[
(\lambda f^{(o\times o)\rightarrow o}.\lambda x^{o\times o}.f \ x) \ g
\]
Theorem

- The typed $\lambda$-calculus with products, $\beta$-reduction and
  with (or without) $\eta$-expansion is confluent and strongly
  normalising.

- We can introduce products as well for the untyped
  $\lambda$-calculus. Then we obtain a confluent (but of course
  non normalising) reduction system.

$\eta$-Rule

- With the $\eta$-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
  $r =_{\beta} \langle \pi_0(r), \pi_1(r) \rangle$:

  $\langle \pi_0(r), \pi_1(r) \rangle \equiv \langle \pi_0(\langle r_0, r_1 \rangle), \pi_1(\langle r_0, r_1 \rangle) \rangle$
  $\beta \quad \langle r_0, r_1 \rangle$

  $\equiv \langle r_0, r_1 \rangle$

- Otherwise $r \rightarrow_{\eta} \langle \pi_0(r), \pi_1(r) \rangle$.

- 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 $\lambda$-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

$$f_1(x, y), \quad f_2(x, y).$$

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 \( \sigma_0, \ldots, \sigma_{n-1} \) and result type \( \rho \) 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

\[
f : \text{Int} \to \text{Float} \to \text{String} .
\]

  Then

\[
\lambda x.\text{Int} \cdot \lambda y.\text{Float} . f \langle x, y \rangle : \text{Int} \to \text{Float} \to \text{String}
\]

is the **Curried** form of \( f \).

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

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

\[
\lambda x.\text{Int} \cdot \lambda y.\text{Float} . f \langle x, y \rangle : \text{Int} \to \text{Float} \to \text{String}
\]

is the **Curried** form of \( f \).

Uncurrying

- We can obtain from the Curried form \( f_{\text{Curry}} \) of a function its Uncurried form \( f_{\text{Uncurry}} \) 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 \( \lambda \)-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. \( \text{Uncurry} f_{\text{Curry}} \rightarrow_{\beta} f_{\text{Uncurry}} \).

- This transformation is called **Uncurrying**.
Currying

We can obtain from the Uncurried form \( f_{\text{Uncurry}} \) of a function its Curried form \( f_{\text{Curry}} \) by

\[
f_{\text{Curry}} = \lambda x_0, \ldots, x_{n-1}. f_{\text{Uncurry}}(x_0, \ldots, x_{n-1})
\]

Again we can define

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

\[
\text{Curry} := \lambda f, x_0, \ldots, x_{n-1}. f(x_0, \ldots, x_{n-1})
\]

s.t. \( f_{\text{Uncurry}} \to_{\beta} f_{\text{Curry}} \).

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

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 \( _+ _3 \) 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

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

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.
(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 : 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:**

```java
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 : Set`
- Then we can introduce the record type

```
record AB : 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 \( A : \) Set where
  
  field \( n : \mathbb{N} \)

  causes an error.

### 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

\[
ab : AB
\]

\[
ab = \text{record}\{a = a'; b = b'\}
\]

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

\[
ab : AB \\
ab = \text{let} \\
a : A \\
a = a' \\
b : B \\
b = b' \\
in \text{record}\{a = a'; b = b}\}
\]

We recommend to avoid such definitions.

Projections

If we define

\[
ab : AB \\
ab = \text{record}\{a = a'; b = b}\}
\]

then we obtain

\[
AB.a ab = a' AB.b ab = b'
\]

Records with Dependencies

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

\[
\text{record} \text{rProd} \(A B : \text{Set}\) \text{where} \\
\text{field} \\
\text{first} : A \\
\text{second} : B
\]

The projections are denoted as follows:

If \(ab : \text{rProd} A B\), then

\[
\text{rProd.first} \ ab : A \\
\text{rProd.second} \ ab : B
\]
Hidden Arguments

When we use

\[ \text{rProd.first : rProd } A \ B \rightarrow A \]

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

In fact \( A \) and \( B \) are **hidden arguments** of \( \text{rProd.first} \).

In case one needs to make them explicit, this can be done as follows:

\[ \text{rProd.first} \ \{ A' \} \ \{ B' \} \ ab \]

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

Hidden Arguments

We can make any argument of a function hidden.

For instance

\[
\begin{align*}
\text{id} & : \{ A : \text{Set} \} \rightarrow A \rightarrow A \\
\text{id} \ a & = a
\end{align*}
\]

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

\[ \text{id} \ \{ A \} \ a \]

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

\[
\begin{align*}
\text{strange} & : \{a : A\} \rightarrow A \\
\text{strange} \{a\} & = a \\
a & : A \\
a & = \text{strange}
\end{align*}
\]

Agda doesn’t complain about the definition of \text{strange}.

However, when checking the definition of \(a\), it notices that it cannot figure out the hidden argument of \text{strange}.

The Product using “data”

The second version of the product uses the more general \texttt{data} construct for defining so called \texttt{algebraic types}.

With this construction we are leaving the so called \texttt{logical framework}.

\(\lambda\text{-terms and the record type form the logical framework}, \) the basic types of Agda and of Martin-Löf type theory.

The \texttt{data}-construct allows to introduce \texttt{user-defined types}.

Example

\[
\begin{align*}
\text{strange} & : \{a : A\} \rightarrow A \\
\text{strange} \{a\} & = a \\
a & : A \\
a & = \text{strange}
\end{align*}
\]

It complains by

- Marking the word \texttt{strange} in yellow.
- Displaying a \texttt{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 \texttt{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 “data”-product is introduced as follows (\texttt{dProd} stands for \texttt{data}-product):

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

Here

- \texttt{dProd }A \ B depends on two sets \(A, B\).
- \(p\) is the \texttt{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 \texttt{Prod'} are exactly the terms \(p\ a\ b\)
  where \(a : A\) and \(b : B\).
Pattern Matching

In order to decompose an element of $dProd \ A \ B$ in Agda, we can use **pattern matching**.

This is best explained by an example.

We postulate $A, B : Set$, and abbreviate $dProd \ A \ B$ as $AB$:

```
postulate A : Set
postulate B : Set
AB  : Set
AB  = dProd A B
```

The second projection can be defined similarly:

```
proj1 : AB -> B ,
proj1 (p a b) = b
```

Note the parentheses around $(p a b)$:

```
proj1 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

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

```
p (p a' b') b''
```

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

We can define

```
f : dProd \ (dProd A B) \ B -> 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

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

\[ f \ (p \ (a \ b) \ b) = a \]

causes an error.

Hidden Arguments in \text{dProd}

- \( p \) in

\[
\text{data} \ \text{dProd} \ (A : \text{Set}) : \text{Set} \ \\
\quad p : A \rightarrow B \rightarrow \text{dProd} \ A \ B
\]

has hidden arguments \( \{A : \text{Set}\} \) and \( \{B : \text{Set}\} \).
- In case one needs to make them explicit, one can do so:

\[
\begin{align*}
    c & : \text{dProd} \ A \ B \\
    c & = p \ \{A\} \ \{B\} \ a \ b
\end{align*}
\]

Coverage Checker

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

\[ f : \mathbb{N} \rightarrow \mathbb{N} \]

\[ f \ Z = Z \]

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

Hidden Arguments in \text{dProd}

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

\[
\begin{align*}
    c & : \text{dProd} \ A \ B \\
    c & = p \ \{A\} \ a \ b
\end{align*}
\]

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

\[
\begin{align*}
    c & : \text{dProd} \ A \ B \\
    c & = p \ \{_\} \ \{B\} \ a \ b
\end{align*}
\]

- 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

\[
f : D \rightarrow A \]
\[
f d \text{ with rProd.first } d \]
\[
f d \mid p \ a \ b = a
\]

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
  \( A, B, C, D : \text{Set} \),
  \( ab : A \times B \)
  \( a-c : A \rightarrow C \),
  \( b-d : B \rightarrow D \)
  an element
  \( f \ ab \ a-c \ b-d : C \times D \).
- This means that \( f \) is a function which takes arguments \( a-c, b-d \) and \( ab \) as above and returns an element of \( C \times D \).
- Therefore

\[
f : (A \times B) \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow D) \rightarrow (C \times D)
\]

A, B, C, D : 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:

```latex
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:

The idea for this function is as follows:

- We first project $ab : A \times B$ to elements $a : A$, $b : B$.
- Then we apply $a-c : A \rightarrow C$ to $a : A$ and obtain an element $c : C$.
- And we apply $b-d : B \rightarrow D$ to $b : B$ and obtain an element $d : D$.
- Finally we form the pair $\langle b, d \rangle$.

The function is as follows:

```latex
\begin{align*}
ab : & A \times B \\
\pi_0 : & A \\
\pi_1 : & B \\
a-c : & A \rightarrow C \\
b-d : & B \rightarrow D \\
\langle \cdot, \cdot \rangle : & A \times B \rightarrow C \times D
\end{align*}
```

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

\[ f : (A \rightarrow C)(B \rightarrow D) \rightarrow AB \rightarrow CD \]
\[ f \ a\ c\ b\ d\ a\ b\ = \ \text{let} \ a' : A \]
\[ a' = AB.a\ a\ b \]
\[ b' : B \]
\[ b' = AB.b\ a\ b \]
\[ c' : C \]
\[ c' = a\ c\ a \]
\[ d' : D \]
\[ d' = b\ d\ b \]
\[ \text{in record}\{c = c'; d = d'\} \]

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
    \[ \text{data Person : Set where} \]
    \[ \text{person : Gender} \rightarrow \text{Name} \rightarrow \text{Person} \]
    \[ \text{gender : Person} \rightarrow \text{Gender} \]
    \[ \text{gender (person} g \ n \text{) =} \ g \]
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:
    
    ```agda
    record _\land_ (A B : Set) : Set where
    field
    and1 : A
    and2 : B
    ```
    
  - The symbol $\land$ can be introduced by typing in $\backslash wedge$.

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

    ```agda
    data _\land_d_ (A B : Set) : Set where
    and : A → B → A \land_d B
    ```

- See `exampleproofprologic3.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 “annd” to expand to $\land$ we do the following:
  - We type in “annd”.
  - We use the emacs command `C-x ail`
  - We type in the mini buffer our intended expansion, namely $\land$ (typed in as “\wedge”).
  - Now whenever we type in a space-like character (blanks and some punctuations) followed by “annd” followed by a space-like character, then “annd” is replaced by $\land$.
  - 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).

Customising Agda with Abbreviation Mode

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

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 `foo`, type in `C-x ail`, and then type in the Mini-buffer `foo`, so that `foo` 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.

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 `exampleproofprologic6.agda`):

  ```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}
  ```
Conjunction with more Conjuncts

If one has a conjunction with more than two conjuncts, e.g. $A \land B \land C$, one can always express it using the binary $\land$:

- As $(A \lor B) \land C$ or $A \land (B \lor C)$.

- If one adds
  
  ```
  infixl 30 _\lor_
  ```

  one can write

  ```
  A \lor B \land C
  ```

  for

  ```
  (A \lor B) \land C
  ```

Conjunction with more Conjuncts

- Especially when using the record version of $\land$ 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 : Set) : Set where
  field
  and1 : A
  and2 : B
  and3 : C
  ```

Conjunction with more Conjuncts

- Definition of the ternary and using “data”:

  ```
  data And3d (A B C : Set) : Set where
  and3d : A -> B -> C -> And3d A B C
  ```

  See `exampleproofprologic5.agda`
One can combine the $\lambda$-calculus with term writing. This means that we have apart from the rules of the typed or untyped $\lambda$-calculus additional rules like $x + 0 \rightarrow x$. Then we obtain for instance

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

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

---

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

Then $s \rightarrow t$, if

- $s$ β-reduces (or η-expands, if one allows the η-rule) to $t$
- or there exists an instantiation $s' \rightarrow t'$ 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 $s$.

Assume for instance the rule

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

Then we have

$$(\lambda f.\lambda x.f(f x)) \text{double}$$

$$\rightarrow \lambda x. \text{double} (\text{double} x)$$

$$\rightarrow \lambda x. \text{double} ((\lambda x. x + x) x)$$

$$\rightarrow \lambda x. \text{double} (x + x)$$

$$\rightarrow \lambda x. (\lambda x. x + x) (x + x)$$

$$\rightarrow \lambda x. (x + x) + (x + x)$$
**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 \( t'[x := s'] \).

**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 \( \lambda \)-calculus by one in which for all rules \( s \rightarrow_{\text{Rule}} t \) we have that \( s, t \) are closed.
- This can be done in such a way that equality (modulo the rewriting rules, \( \beta \) and possibly \( \eta \)) in both systems coincide:
- Assume a rule 
  \[ s \rightarrow_{\text{Rule}} t \]
  and let \( x_1, \ldots, x_n \) be the free variables in \( s \).
- Then replace this rule by
  \[ \lambda x_1, \ldots, x_n, s \rightarrow_{\text{Rule'}} \lambda x_1, \ldots, x_n, t \]
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 \( r =_\beta r' \xrightarrow{\text{Rule}'} u' =_\beta u \), so the reduction can be simulated in the second system.

Example

We can replace the rewriting rules

\[
\begin{align*}
x + 0 & \rightarrow x \\
x + S \ y & \rightarrow S (x + y)
\end{align*}
\]

by

\[
\begin{align*}
\lambda x. x + 0 & \rightarrow \lambda x. x \\
\lambda x, y. x + S \ y & \rightarrow \lambda x, y. S (x + y)
\end{align*}
\]

That

\[
S (0 + S 0) =_\beta S (S (0 + 0)) =_\beta S (S 0)
\]

becomes in the new system

\[
S (0 + S 0) =_\beta S ((\lambda x, y. x + S \ y) 0 0) =_\beta S (S (0 + 0)) =_\beta S (S ((\lambda x. x + 0) 0)) =_\beta S (S 0)
\]

Extended Typed \( \lambda \)-Calculus

Finally, we can combine the typed \( \lambda \)-calculus (with or without products, with or without \( \eta \)-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 $\lambda$-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 $\lambda$-terms is then introduced by the same rules as before, but we have as additional rule:
  - If $c$ is a constant of type $\sigma$, 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 $\lambda x : \text{nat} . x + x : \text{nat} \rightarrow \text{nat}$:

\[
\begin{align*}
\text{Ap} & : \Gamma \Rightarrow x : \text{nat} \Rightarrow x : \text{nat} \rightarrow \text{nat} \\
\text{Ap} & : \Gamma \Rightarrow x : \text{nat} \Rightarrow \_ + \_ x : \text{nat} \\
\text{Ap} & : \Gamma \Rightarrow x : \text{nat} \Rightarrow x : \text{nat} \\
\text{Ap} & : \Gamma \Rightarrow x : \text{nat} \Rightarrow \_ + \_ x : \text{nat} \\
\text{Ap} & : \Gamma \Rightarrow x : \text{nat} \Rightarrow (\lambda x : \text{nat} . x + x) : \text{nat} \rightarrow \text{nat}
\end{align*}
\]

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

Example

Then we have

\[
\begin{align*}
(\lambda f : \text{nat} \rightarrow \text{nat} . \lambda x : \text{nat} . f (f x)) \text{ double} & \\
\rightarrow & \lambda x : \text{nat} . \text{ double} (\text{ double } x) \\
\rightarrow & \lambda x : \text{nat} . \text{ double} ((\lambda x : \text{nat} . x + x) x) \\
\rightarrow & \lambda x : \text{nat} . \text{ double} (x + x) \\
\rightarrow & \lambda x : \text{nat} . (\lambda x : \text{nat} . x + x) (x + x) \\
\rightarrow & \lambda x : \text{nat} . (x + x) + (x + x)
\end{align*}
\]

Extended Typed $\lambda$-Calculus

- Reduction rules should now be of the form $\Gamma \Rightarrow s \xrightarrow{\text{Rule } t : \sigma} (\text{instead of } s \xrightarrow{\text{Rule } t})$ 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 $\Gamma$.
- Best guaranteed by demanding that all variables in $\Gamma$ occur free in $s$.
- One usually omits $\Gamma, \sigma$, if it is clear from the context.
- Very often, the reduction rules will be of the form $c \xrightarrow{\text{Rule } t : \sigma}$ where $c$ is a constant and therefore $t$ a closed term.
Extended Typed $\lambda$-Calculus

- Instantiations of a rule $\Gamma \Rightarrow s \rightarrow_{\text{Rule } t : \sigma}$ are now obtained by replacing variables $x$ of type $\tau$ by terms $r : \tau$ (possibly depending on some context $\Delta$).
- 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 $r \rightarrow r' : \sigma$ by the right hand side $r'$.

Example

Assume

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

(f) Finite Sets and Decidable Formulae

We want to add types containing finitely many elements to the $\lambda$-calculus.

We treat first the special case $\text{Bool}$ (finite set with 2 elements) and then generalise this to general finite sets.

Example

Then we have

\[
(\lambda f^{\text{nat} \to \text{nat}} . \lambda x^{\text{nat}}. f (f x)) \text{ double} \\
\rightarrow \lambda x^{\text{nat}}. \text{ double} (\text{ double } x) \\
\rightarrow \lambda x^{\text{nat}}. \text{ double} (((\lambda x.x + x) x) \\
\rightarrow \lambda x^{\text{nat}}. (\lambda x.x + x) (x + x) \\
\rightarrow \lambda x^{\text{nat}}. (x + x) + (x + x)
\]
The Type of Booleans

- We add a new type \( \text{Bool} \) to the set of ground types.
- We add constants \( \texttt{tt} : \text{Bool} \), \( \texttt{ff} : \text{Bool} \).
- Here \( \texttt{tt} \) stands for true, \( \texttt{ff} \) for false.

Case\( ^\sigma \text{Bool} \)

- Furthermore we add the principle of case distinction to the \( \lambda \)-calculus extended by \( \text{Bool} \):
  - Assume we have a type \( \sigma \) and
    \[
    \text{case}_{\text{tt}} : \sigma \quad \text{case}_{\text{ff}} : \sigma
    \]
  - Then we want to have that
    \[
    \text{Case}_{\text{Bool}} \text{case}_{\text{tt}} \text{case}_{\text{ff}} : \text{Bool} \rightarrow \sigma
    \]
  - And we want that
    \[
    \text{Case}_{\text{Bool}} \text{case}_{\text{tt}} \text{case}_{\text{ff}} \text{tt} = \text{case}_{\text{tt}}
    \]
    \[
    \text{Case}_{\text{Bool}} \text{case}_{\text{tt}} \text{case}_{\text{ff}} \text{ff} = \text{case}_{\text{ff}}
    \]

If then else

- Case\( ^\sigma \text{Bool} \) \text{case}_{\text{tt}} \text{case}_{\text{ff}} \text{tt} = \text{case}_{\text{tt}}
- Case\( ^\sigma \text{Bool} \) \text{case}_{\text{tt}} \text{case}_{\text{ff}} \text{ff} = \text{case}_{\text{ff}}
- Case\( ^\sigma \text{Bool} \) \text{case}_{\text{tt}} \text{case}_{\text{ff}} \) \text{b} corresponds to
  - If \( b \) then \text{case}_{\text{tt}} else \text{case}_{\text{ff}} :
    - In case \( b = \texttt{tt} \), the if-then-else-term should be equal to \text{case}_{\text{tt}} , as it is the case for Case\( ^\sigma \text{Bool} \) \text{case}_{\text{tt}} \text{case}_{\text{ff}} \) \text{b}.
    - In case \( b = \texttt{ff} \), the if-then-else-term should be equal to \text{case}_{\text{ff}} , as it is the case for Case\( ^\sigma \text{Bool} \) \text{case}_{\text{tt}} \text{case}_{\text{ff}} \) \text{b}.

Type of Case\( ^\sigma \text{Bool} \)

- We don’t need to have a complex rule for forming Case\( ^\sigma \text{Bool} \) \text{case}_{\text{tt}} \text{case}_{\text{ff}} .
- All we need to do is add a constant Case\( ^\sigma \text{Bool} \) of type
  \[
  \text{Case}_{\text{Bool}} : \sigma \rightarrow \sigma \rightarrow \text{Bool} \rightarrow \sigma
  \]
- Then it follows that, whenever \text{case}_{\text{tt}} : \sigma \) and \text{case}_{\text{ff}} : \sigma ,
  then
  \[
  \text{Case}_{\text{Bool}} \text{case}_{\text{tt}} \text{case}_{\text{ff}} : \text{Bool} \rightarrow \sigma
  \]
- The equalities are achieved by adding reductions
  \[
  \text{Case}_{\text{Bool}} \text{case}_{\text{tt}} \text{case}_{\text{ff}} \text{tt} \rightarrow \text{case}_{\text{tt}}
  \]
  \[
  \text{Case}_{\text{Bool}} \text{case}_{\text{tt}} \text{case}_{\text{ff}} \text{ff} \rightarrow \text{case}_{\text{ff}}
  \]
Example: Boolean Conjunction

- We define Boolean valued conjunction
  \[ _\land_{\text{Bool}} : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} . \]

- We write
  \[ _\land_{\text{Bool}} \] for function symbol,
  \[ \land_{\text{Bool}} \] for the symbol, written infix,
  so \( b \land_{\text{Bool}} c \) stands for \( _\land_{\text{Bool}} b c \).

- 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 \( A \land B \).
  \( b \land_{\text{Bool}} c \) will form the Boolean value corresponding to the conjunction of \( b \) and \( c \).

Truth Table for \( \land_{\text{Bool}} \)

- \( \land_{\text{Bool}} \) has the following truth table:

\[
\begin{array}{c|cc}
\land_{\text{Bool}} & ff & tt \\
\hline
ff & ff & ff \\
tt & ff & tt
\end{array}
\]

- So we have
  \[ ff \land_{\text{Bool}} b = ff \]
  \[ tt \land_{\text{Bool}} b = b \]

Example: \( \land_{\text{Bool}} \)

- 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 \( (\text{Atom} b) \land (\text{Atom} c) \) is equivalent to \( \text{Atom}(b \land_{\text{Bool}} c) \).

- This means that \( b \land_{\text{Bool}} c \) is true iff \( b \) is true and \( c \) is true.
Example: $\land_{\text{Bool}}$

\[ \land_{\text{Bool}} = \lambda(b, c : \text{Bool}).\text{Case}_{\text{Bool}} e f b \]

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

In total we define therefore
\[ \land_{\text{Bool}} = \lambda(b, c : \text{Bool}).\text{Case}_{\text{Bool}} c \text{ ff } b \]
\[ : \text{ Bool } \rightarrow \text{ Bool } \rightarrow \text{ Bool } \]

We verify the correctness of this definition:
- \[ \text{tt} \land_{\text{Bool}} c = \land_{\text{Bool}} \text{ tt } c = \text{Case}_{\text{Bool}} c \text{ ff } \text{tt} = c. \]
  as desired.
- \[ \text{ff} \land_{\text{Bool}} c = \land_{\text{Bool}} \text{ ff } c = \text{Case}_{\text{Bool}} c \text{ ff } \text{ff} = \text{ff}. \]
  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 $\text{Fin}_n$.
  - 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 $\text{Fin}_n = \{A_0^n, A_1^n, \ldots, A_{n-1}^n\}$

especially in the cases $n = 2, 1, 0$ we have
\[
\begin{align*}
\text{Fin}_2 & = \{A_0^2, A_1^2\} \\
\text{Fin}_1 & = \{A_0^1\} \\
\text{Fin}_0 & = \emptyset
\end{align*}
\]

We have not made use of dependent types yet. $n$, $k$ are external natural numbers.
- So we have for each $n$ added one type $\text{Fin}_n$ to the calculus.
- We have for each $n$ and $k < n$ added one constant $A_k^n$ to the calculus.
**Rules for \( \text{Fin}_n \)**

- We add the principle of case distinction on \( \text{Fin}_n \):
  - Assume \( n \in \mathbb{N}, \) a type \( \sigma, \) and \( \text{case}_i : \sigma \) for \( i = 0, \ldots, n - 1. \)
  - Then we want
    \[
    \text{Case}_n^\sigma \text{ case}_0 \cdots \text{ case}_{n-1} : \text{Fin}_n \rightarrow \sigma
    \]
  - And we want
    \[
    \text{Case}_n^\sigma \text{ case}_0 \cdots \text{ case}_{n-1} A_i^n = \text{case}_i
    \]

**Equality on \( \text{Fin}_n \)**

- We can now define the Boolean valued function which determines for two elements of \( \text{Fin}_n \), whether they are equal:
  - Define
    \[
    \text{Eq}_{n, \text{Bool}} : \text{Fin}_n \rightarrow \text{Fin}_n \rightarrow \text{Bool}
    \]
    s.t.
    \[
    \text{Eq}_{n, \text{Bool}} A_i^n A_i^n = \text{tt}
    \]
    \[
    \text{Eq}_{n, \text{Bool}} A_i^n A_j^n = \text{ff} \quad \text{for } i \neq j
    \]
  - \( \text{Eq}_{n, \text{Bool}} \) can be defined easily (for fixed \( n \)) by case distinction on its two arguments.

**Constants \( \text{Case}_n^\sigma \)**

- As for \( \text{Bool}, \) this can be achieved by having constants
  \[
  \text{Case}_n^\sigma : \sigma \rightarrow \cdots \rightarrow \sigma \rightarrow \text{Fin}_n \rightarrow \sigma
  \]
  \[
  \text{Then from } \text{case}_i : \sigma \text{ we obtain}
  \]
  \[
  \text{Case}_n^\sigma \text{ case}_0 \cdots \text{ case}_{n-1} : \text{Fin}_n \rightarrow \sigma
  \]
  \[
  \text{Furthermore we add the reduction rules}
  \]
  \[
  \text{Case}_n^\sigma \text{ case}_0 \cdots \text{ case}_{n-1} A_i^n \longrightarrow \text{case}_i
  \]

**Special Case \( \text{Bool} \)**

- \( \text{Bool} \) can now be treated as the special case
  \[
  \text{Fin}_n
  \]
  \[
  \text{with } n = 2:
  \]
  \[
  \text{Bool} := \text{Fin}_2
  \]
  \[
  \text{tt} := A_0^2 : \text{Bool}
  \]
  \[
  \text{ff} := A_1^2 : \text{Bool}
  \]
  \[
  \text{Case}_{\text{Bool}} := \text{Case}_2^\sigma : \sigma \rightarrow \sigma \rightarrow \text{Bool} \rightarrow \sigma
  \]
**Rules for ⊤**

⊤ (pronounced “top”) is the special case $\text{Fin}_n$

for $n = 1$

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

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

**⊤ 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.
- ⊤ has exactly one proof, and corresponds therefore to the always **always true formula**.
- That's why we call the element $\text{true}$,

  $$\text{true}$$

since it is the proof of the always true formula.

- Example: we have

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

**Rules for ⊥**

⊥ (pronounced “bottom”) is the special case $\text{Fin}_n$

for $n = 0$:

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

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

- We have no reduction rules.

- ⊥ has no elements.
- It is the formula, which is **always false**, since it has no proofs.
  - Often called **falsum** or **absurdity**.
Case \( \sigma \perp \) 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, Case \( \sigma \perp \) 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 \( \perp \rightarrow A \):

\[ \text{Case}^A_{\perp} : \perp \rightarrow A \]

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 \( A \rightarrow \perp \).

Pattern Matching

We can use pattern matching in order to make case distinction on an argument of type \( \text{Bool} \):

Assume we want to define

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

\[ \neg \text{Bool} \ tt = \ ff \]

\[ \neg \text{Bool} \ ff = \ tt \]

The above is already the Agda code defining \( \neg \text{Bool} \).

examenegbool.agda

We introduce \( \text{Bool} \) by listing its constructors

\[
\begin{align*}
\text{data} & \quad \text{Bool} : \text{Set} \; \text{where} \\
\text{tt} & \quad : \text{Bool} \\
\text{ff} & \quad : \text{Bool}
\end{align*}
\]
**Finite Sets in Agda**

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

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

**Case distinction on finite sets in Agda can be done using pattern matching.**

In the “Colour” example above for instance, we can define

```agda
  is-red : Colour → Bool
  is-red red = tt
  is-red _ = ff
```

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

  ```agda
  is-red _ = ff
  ```

  **matches** `is-red red`.

**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
  ```agda
  is-red red = tt
  ```

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

**⊤ in Agda**

- The definition of `⊤` in Agda is **straightforward**:

  ```agda
data ⊤ : Set where
    true : ⊤
  ```

- We can define a function having an argument in `⊤` by using pattern matching:

  ```agda
  g : ⊤ → Bool
  q true = tt
  ```
\[ \top \text{ in Agda} \]

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

\[
\text{record } \top' : \text{Set} \text{ where}
\]

- Then the element \( \text{true} \) of \( \top \) is defined as follows

\[
\text{true}' : \top'
\]
\[
\text{true}' = \text{record}\{\}
\]

- Agda has a built-in \( \eta \)-rule, which says that every \( x : \top \) is equal to \text{record}\{\}.

\text{exampletrue.agda}

\[ \bot \text{ in Agda} \]

- \( \bot \) can be defined as the “data”-set with no constructors:

\[
\text{data } \bot : \text{Set} \text{ where}
\]

- If we want to define

\[
g : \bot \rightarrow \text{Bool}
\]

by pattern matching, we see that there is no element in \( \bot \), so there is no constructor case matching \( g \cdot x \).

\text{examplefalse.agda}

\[ \neg \text{ in Agda} \]

- Above we have shown why we can define \( \neg A \) as \( A \rightarrow \bot \).

- Therefore negation can be defined in Agda as follows:

\[
\neg : \text{Set} \rightarrow \text{Set}
\]
\[
\neg A = A \rightarrow \bot
\]
Example for the Use of ⊥

Assume the type of trees:

```agda
data Tree : Set where
  oak    : Tree
  pine   : Tree
  spruce : Tree
```

We can now define

```agda
IsConifer : Tree → Set
IsConifer oak = ⊥
IsConifer _ = ⊤
```

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

Example for the Use of ⊥

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 ⊥

Assume the type Stack of stacks of elements of \( \mathbb{N} \) given by

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

We can then introduce a predicate \( \text{NonEmpty} \) expressing that the stack is nonempty:

```agda
NonEmpty : {A : Set} → Stack A → Set
NonEmpty empty     = ⊥
NonEmpty (push _ _) = ⊤
```
Example 2 for the Use of $\bot$

Now we can define

$$\text{top} : \{A : \text{Set}\} \rightarrow (s : \text{Stack } A) \rightarrow \text{NonEmpty } s \rightarrow \text{Set}$$

$$\text{top empty} \quad ()$$

$$\text{top (push } a \_ ) \quad _\_ = a$$

(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 $\lambda$-calculus, and move to dependent types.

The operation which converts Boolean values into atomic formulae is

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

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

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

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 $\text{Fin}_n$, which for fixed $n$ can be defined in Agda.

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

$$\text{Eq}_{n, \text{Bool}} A^i_n A^i_n = \text{tt}$$

$$\text{Eq}_{n, \text{Bool}} _\_ _\_ = \text{ff}$$
**Example**

For instance in case of the set

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

we define

```agda
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:

```agda
_==_ : Colour → Colour → Set
\_ == \_ = Atom (eqColourBool \_ \_)
```

_\_ == \_\_ is the formula expressing that \_ and \_\_ are the same colour.

**Example 2**

Remember the definition of Boolean valued negation in Agda:

```agda
¬Bool : Bool → Bool
¬Bool tt = ff
¬Bool ff = tt
```

We show

```agda
Atom (¬Bool b) → ¬ (Atom b)
```

Remember that we defined

```agda
¬ : Set → Set
¬ A = A → ⊥
```
Example 2

So our lemma is

\[
\text{Lemma} : \text{Set} \\
\text{Lemma} = (b : \text{Bool}) \to \text{Atom}(\neg \text{Bool } b) \to \neg (\text{Atom } b)
\]

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

\[
\text{lemma} : \text{Lemma} \\
\text{lemma } b \ p \ q = \{! !\}
\]

Example 2

\[
\text{Lemma} : \text{Set} \\
\text{Lemma} = (b : \text{Bool}) \to \text{Atom}(\neg \text{Bool } b) \to \text{Atom } b \to \bot
\]

\[
\text{lemma } : \text{Lemma} \\
\text{lemma } tt \ p \ q = \{! !\} \\
\text{lemma } ff \ p \ q = \{! !\}
\]

In the first equation we have \(b = tt\), therefore

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

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

Example 2

\[
\text{Lemma} : \text{Set} \\
\text{Lemma} = (b : \text{Bool}) \to \text{Atom}(\neg \text{Bool } b) \to \text{Atom } b \to \bot
\]

\[
\text{lemma } : \text{Lemma} \\
\text{lemma } tt () \ q = \{! !\} \\
\text{lemma } ff \ p \ q = \{! !\}
\]

In the second case we have \(b = ff\), so

\[
q : \text{Atom } b = \text{Atom } ff = \bot
\]

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 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.

Jump over Example 3 (→Bool)

Example 3

We introduce Boolean valued implication

_→Bool_ : Bool → Bool → Bool

and show that Atom (b →Bool b') implies
Atom b → Atom b'.

The other direction can be shown as well.
Decidable Predicates

- In general, $\text{Atom}$ 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

$$\text{EqN,Bool} : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$$

which decides it.

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

**Example 3**

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

**Lemma**

- $b = \text{ff}$, then
  $$btrue : \text{Atom } b = \bot$$

  which matches the empty pattern.
- $b = \text{tt}$, $b' = \text{ff}$, then
  $$b \to b' : \text{Atom } (b \to \text{Bool } b') = \text{Atom } \text{ff} = \bot$$

  which again matches the empty pattern.

**Example 3**

 lemma : (b b' : Bool) → Atom (b →Bool b')→ Atom b
  → Atom b'

 lemma b b' b→b' btrue = {! !(}

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

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

  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**

**Lemma**

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

$$\text{Lemma : Lemma}$$

$$\text{lemma } \text{ff } _ _ ()$$

$$\text{lemma } \text{tt } \text{ff } () _ _$$

$$\text{lemma } \text{tt } \text{tt } _ _ _ _ = \text{true}$$

**Example 3**

**Lemma**

$$\text{Lemma : Set}$$

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

$$\text{lemma : Lemma}$$

$$\text{lemma } \text{ff } _ _ ()$$

$$\text{lemma } \text{tt } \text{ff } () _ _$$

$$\text{lemma } \text{tt } \text{tt } _ _ _ _ = \text{true}$$

In general, $\text{Atom}$ 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

$$\text{EqN,Bool} : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$$

which decides it.

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

In general, $\text{Atom}$ 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

$$\text{EqN,Bool} : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$$

which decides it.

- The equality on **functions** $\mathbb{N} \to \mathbb{N}$ 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

\[ \text{RealWorldState} : \text{Set} \]

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

a set of control states

\[ \text{ControlState} : \text{Set} \]

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

Let now

\[ \text{Safe} : \text{RealWorldState} \to \text{Set} \]

\[ \text{Safe} \ s = \text{Atom}(\text{safeBool} \ s) \]

If \( \text{safeBool} \ s \) is \textbf{true} (e.g. \( s \) is safe), \( \text{Safe} \ s \) is \textbf{inhabited}, i.e. provable.

If \( \text{safeBool} \ s \) is \textbf{false} (e.g. \( s \) is unsafe), \( \text{Safe} \ s \) is \textbf{not inhabited}.

Decidable Predicates (Cont.)

and a function

\[ \text{control} \to \text{realWorld} : \text{ControlState} \to \text{RealWorldState} \]

mapping control states to external states they represent.

Furthermore, assume we have defined in Agda a function

\[ \text{safeBool} : \text{RealWorldState} \to \text{Bool} \]

The intended meaning is that

\[ \text{safeBool} \ s \]

means: \textit{real world state } \( s \textit{ is safe.} \)

The existence of a

\[ p : (s : \text{ControlState}) \to \text{Safe} (\text{control} \to \text{realWorld} \ s) \]

means:

\begin{itemize}
  \item For every \( s : \text{ControlState} \) we have that if \( s' := \text{control} \to \text{realWorld} \ s \) is the corresponding real world state, then \( \text{Safe} \ s' \) is \textbf{inhabited},
  \item i.e. \( \text{Safe} \ s' \) is \textbf{true},
  \item i.e. \( s' \) is \textbf{safe}.
\end{itemize}
So if we have a proof

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

we have shown that \textbf{the system is safe} w.r.t. the safety property expressed by \( \text{safe}_{\text{Bool}} \).