

# CSC\_51051\_EP: Dependent types

---

Samuel Mimram

2024

École polytechnique

## Dependent types

First order logic is fine, but we would like much more.

## Dependent types

First order logic is fine, but we would like much more.

- we would like quantifications to be typed ( $\forall x \in A. \dots$  instead of  $\forall x. \dots$ ),

# Dependent types

First order logic is fine, but we would like much more.

- we would like quantifications to be typed ( $\forall x \in A. \dots$  instead of  $\forall x. \dots$ ),
- we would like to manipulate terms and proofs in the same way:

# Dependent types

First order logic is fine, but we would like much more.

- we would like quantifications to be typed ( $\forall x \in A. \dots$  instead of  $\forall x. \dots$ ),
- we would like to manipulate terms and proofs in the same way:
  - we can prove  $\Gamma \vdash t : \mathbb{N}$  or  $\Gamma \vdash t : x = y$  in the same way,

# Dependent types

First order logic is fine, but we would like much more.

- we would like quantifications to be typed ( $\forall x \in A. \dots$  instead of  $\forall x. \dots$ ),
- we would like to manipulate terms and proofs in the same way:
  - we can prove  $\Gamma \vdash t : \mathbb{N}$  or  $\Gamma \vdash t : x = y$  in the same way,
  - we unify the two  $\lambda$ -abstractions for terms and for proofs:  
 $\lambda n. t : \forall x \in \mathbb{N}. A$  vs  $\lambda x^A. t : A \rightarrow B$

# Dependent types

First order logic is fine, but we would like much more.

- we would like quantifications to be typed ( $\forall x \in A. \dots$  instead of  $\forall x. \dots$ ),
- we would like to manipulate terms and proofs in the same way:
  - we can prove  $\Gamma \vdash t : \mathbb{N}$  or  $\Gamma \vdash t : x = y$  in the same way,
  - we unify the two  $\lambda$ -abstractions for terms and for proofs:  
 $\lambda n. t : \forall x \in \mathbb{N}. A$  vs  $\lambda x^A. t : A \rightarrow B$
- this means that proofs can depend on terms and terms can depend on proofs,

# Dependent types

First order logic is fine, but we would like much more.

- we would like quantifications to be typed ( $\forall x \in A. \dots$  instead of  $\forall x. \dots$ ),
- we would like to manipulate terms and proofs in the same way:
  - we can prove  $\Gamma \vdash t : \mathbb{N}$  or  $\Gamma \vdash t : x = y$  in the same way,
  - we unify the two  $\lambda$ -abstractions for terms and for proofs:  
 $\lambda n. t : \forall x \in \mathbb{N}. A$  vs  $\lambda x^A. t : A \rightarrow B$
- this means that proofs can depend on terms and terms can depend on proofs,
- if we have a type **Type** of all types, propositions become terms of type **Type**,



# Dependent types

First order logic is fine, but we would like much more.

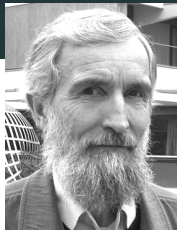
- we would like quantifications to be typed ( $\forall x \in A. \dots$  instead of  $\forall x. \dots$ ),
- we would like to manipulate terms and proofs in the same way:
  - we can prove  $\Gamma \vdash t : \mathbb{N}$  or  $\Gamma \vdash t : x = y$  in the same way,
  - we unify the two  $\lambda$ -abstractions for terms and for proofs:  
 $\lambda n. t : \forall x \in \mathbb{N}. A$  vs  $\lambda x^A. t : A \rightarrow B$
- this means that proofs can depend on terms and terms can depend on proofs,
- if we have a type **Type** of all types, propositions become terms of type **Type**,
- we also would like to generate inductive principles automatically,

# Dependent types

First order logic is fine, but we would like much more.

- we would like quantifications to be typed ( $\forall x \in A. \dots$  instead of  $\forall x. \dots$ ),
- we would like to manipulate terms and proofs in the same way:
  - we can prove  $\Gamma \vdash t : \mathbb{N}$  or  $\Gamma \vdash t : x = y$  in the same way,
  - we unify the two  $\lambda$ -abstractions for terms and for proofs:  
 $\lambda n. t : \forall x \in \mathbb{N}. A$  vs  $\lambda x^A. t : A \rightarrow B$
- this means that proofs can depend on terms and terms can depend on proofs,
- if we have a type **Type** of all types, propositions become terms of type **Type**,
- we also would like to generate inductive principles automatically,
- we would like to have a decent handling of equality.

# Dependent types



They were introduced by Martin-Löf 1972.

The arrow type  $A \rightarrow B$  is generalized into

$$\Pi(x : A).B$$

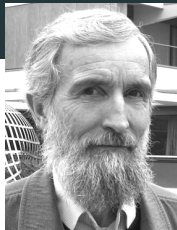
where  $x$  can occur in  $B$ : the type of the result can **depend** on the input.

Typical example:

$$\Pi(n : \mathbb{N}). \text{Vec } n$$

(here  $\text{Vec } n$  is the type of lists of length  $n$  with elements of a fixed type, say  $\mathbb{N}$ ).

# Dependent types



They were introduced by Martin-Löf 1972.

The arrow type  $A \rightarrow B$  is generalized into

$$\Pi(x : A).B$$

where  $x$  can occur in  $B$ : the type of the result can **depend** on the input.

Typical example:

$$\Pi(n : \mathbb{N}). \text{Vec } n$$

(here  $\text{Vec } n$  is the type of lists of length  $n$  with elements of a fixed type, say  $\mathbb{N}$ ).

And we add a type  $\text{Type}$  of all types.

The Agda notation for

$$\Pi(x : A).B$$

is

$$(x : A) \rightarrow B$$

# Agda notations

The Agda notation for

$\prod(x : A).B$

is

$(x : A) \rightarrow B$

The Agda notation for

Type

is

Set

Part I

# Dependent type theory

# Terms

There is no more distinction between terms and types: a type is a term of type **Type**.

An **expression** is a term of the form

It is either  $e, e' ::= x \mid e e' \mid \lambda x^e. e' \mid \Pi(x : e). e' \mid \text{Type}$

- $x$ : a variable,
- $e e'$ : an application,
- $\lambda x^e. e'$ : an abstraction,
- $\Pi(x : e). e'$ : a  $\Pi$ -type,
- **Type**: the type of types.



# Terms

There is no more distinction between terms and types: a type is a term of type **Type**.

An **expression** is a term of the form

It is either  $e, e' ::= x \mid e e' \mid \lambda x^e. e' \mid \Pi(x : e). e' \mid \text{Type}$

- $x$ : a variable,
- $e e'$ : an application,
- $\lambda x^e. e'$ : an abstraction,
- $\Pi(x : e). e'$ : a  $\Pi$ -type,
- **Type**: the type of types.

In the following, we also use  $A$  or  $t$  to denote expressions,  
but there is no syntactic distinction between terms and types!

# Terms

There is no more distinction between terms and types: a type is a term of type **Type**.

An **expression** is a term of the form

It is either  $e, e' ::= x \mid e e' \mid \lambda x^e. e' \mid \Pi(x : e). e' \mid \text{Type}$

- $x$ : a variable,
- $e e'$ : an application,
- $\lambda x^e. e'$ : an abstraction,
- $\Pi(x : e). e'$ : a  $\Pi$ -type,
- **Type**: the type of types.

In the following, we also use  $A$  or  $t$  to denote expressions,  
but there is no syntactic distinction between terms and types!  
(we could if we want but things are simpler this way)

## Terms: free variables

We write  $FV(e)$  for the free variables of an expression  $e$ .

## Terms: free variables

We write  $FV(e)$  for the free variables of an expression  $e$ .

$$FV(x) = \{x\}$$

$$FV(t\ u) = FV(t) \cup FV(u)$$

$$FV(\lambda x^A.t) = FV(A) \cup (FV(t) \setminus \{x\})$$

$$FV(\Pi(x : A).B) = FV(A) \cup (FV(B) \setminus \{x\})$$

$$FV(\text{Type}) = \emptyset$$

## Terms: free variables

We write  $FV(e)$  for the free variables of an expression  $e$ .

$$FV(x) = \{x\}$$

$$FV(t\ u) = FV(t) \cup FV(u)$$

$$FV(\lambda x^A.t) = FV(A) \cup (FV(t) \setminus \{x\})$$

$$FV(\Pi(x : A).B) = FV(A) \cup (FV(B) \setminus \{x\})$$

$$FV(\text{Type}) = \emptyset$$

Expressions are considered up to  $\alpha$ -**equivalence**: we can rename bound variables.

## Terms: substitution

We write  $e[u/x]$  for the expression  $e$  where  $x$  has been replaced by  $u$ .

## Terms: substitution

We write  $e[u/x]$  for the expression  $e$  where  $x$  has been replaced by  $u$ .

$$x[u/x] = u$$

$$y[u/x] = y$$

if  $x \neq y$

$$(t \ t')[u/x] = (t[u/x]) \ (t'[u/x])$$

$$(\lambda y^A. t)[u/x] = \lambda y^{A[u/x]}. t[u/x]$$

with  $y \notin \text{FV}(u) \cup \{x\}$

$$(\Pi(y : A). B)[u/x] = \Pi(y : A[u/x]). B[u/x]$$

with  $y \notin \text{FV}(u) \cup \{x\}$

$$\text{Type } [u/x] = \text{Type}$$

# Contexts

A context  $\Gamma$  is a list

$$x_1 : A_1, \dots, x_n : A_n$$

of variables and expressions.



# Contexts

A context  $\Gamma$  is a list

$$x_1 : A_1, \dots, x_n : A_n$$

of variables and expressions.

Its domain is

$$\text{dom}(\Gamma) = \{x_1, \dots, x_n\}$$

# Contexts

A context  $\Gamma$  is a list

$$x_1 : A_1, \dots, x_n : A_n$$

of variables and expressions.

Its domain is

$$\text{dom}(\Gamma) = \{x_1, \dots, x_n\}$$

The order really matters here: we can make sense of

$$n : \mathbb{N}, l : \text{Vec } n$$

but not of

$$l : \text{Vec } n, n : \mathbb{N}$$

# Judgments

The judgments are more complicated than before:

- a context is not necessarily well-formed:

# Judgments

The judgments are more complicated than before:

- a context is not necessarily well-formed:
  - $n : \mathbb{N}, l : \text{Vec } n$  is well-formed,

# Judgments

The judgments are more complicated than before:

- a context is not necessarily well-formed:
  - $n : \mathbb{N}, l : \text{Vec } n$  is well-formed,
  - $l : \text{Vec } n, \dots$  is not well-formed,

# Judgments

The judgments are more complicated than before:

- a context is not necessarily well-formed:
  - $n : \mathbb{N}, l : \text{Vec } n$  is well-formed,
  - $l : \text{Vec } n, \dots$  is not well-formed,
  - $n : \text{Bool}, l : \text{Vec } n$  is not well-formed,

# Judgments

The judgments are more complicated than before:

- a context is not necessarily well-formed:
  - $n : \mathbb{N}, l : \text{Vec } n$  is well-formed,
  - $l : \text{Vec } n, \dots$  is not well-formed,
  - $n : \text{Bool}, l : \text{Vec } n$  is not well-formed,
- we need to take reduction in account: we can apply a function  $f$  of type

$$\Pi(l : \text{Vec } 5).A$$

to an argument of type

$$\text{Vec } (3 + 2)$$

# Judgments

We have three forms of **judgments**:

- $\Gamma$  is a well-formed context:

$\Gamma \vdash$



# Judgments

We have three forms of **judgments**:

- $\Gamma$  is a well-formed context:

$$\Gamma \vdash$$

- $t$  has type  $A$  under the hypothesis  $\Gamma$ :

$$\Gamma \vdash t : A$$

# Judgments

We have three forms of **judgments**:

- $\Gamma$  is a well-formed context:

$$\Gamma \vdash$$

- $t$  has type  $A$  under the hypothesis  $\Gamma$ :

$$\Gamma \vdash t : A$$

- $t$  and  $u$  are equal terms of type  $A$  under the hypothesis  $\Gamma$ :

$$\Gamma \vdash t = u : A$$

NB: this is *definitional* equality.

# Judgments

We have three forms of **judgments**:

- $\Gamma$  is a well-formed context:

$$\Gamma \vdash$$

- $t$  has type  $A$  under the hypothesis  $\Gamma$ :

$$\Gamma \vdash t : A$$

- $t$  and  $u$  are equal terms of type  $A$  under the hypothesis  $\Gamma$ :

$$\Gamma \vdash t = u : A$$

NB: this is *definitional* equality.

They mutually need each other!

We also need to express that a type is **well-formed**:

- in the context  $n : \mathbb{N}$ , the type  $\text{Vec } n$  is well-formed,
- in the context  $n : \text{Bool}$  or the empty context, the type  $\text{Vec } n$  is not well-formed.

Here, this is represented by judgments of the form

$$\Gamma \vdash A : \text{Type}$$

## Rules: contexts

The rules for well-formedness of **contexts** are

$$\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash A : \text{Type}}{\Gamma, x : A \vdash}$$

## Rules: contexts

The rules for well-formedness of **contexts** are

$$\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash A : \text{Type}}{\Gamma, x : A \vdash}$$

For instance,

$$n : \mathbb{N}, l : \text{Vec } n \vdash$$

## Rules: contexts

The rules for well-formedness of **contexts** are

$$\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash A : \text{Type}}{\Gamma, x : A \vdash}$$

For instance,

$$\frac{n : \mathbb{N} \vdash \text{Vec } n : \text{Type}}{n : \mathbb{N}, l : \text{Vec } n \vdash}$$

## Rules: contexts

The rules for well-formedness of **contexts** are

$$\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash A : \text{Type}}{\Gamma, x : A \vdash}$$

For instance,

$$\frac{\begin{array}{c} \vdots \\ \hline n : \mathbb{N} \vdash \text{Vec} : \Pi(\_ : \mathbb{N}). \text{Type} \end{array} \quad \frac{}{n : \mathbb{N} \vdash n : \mathbb{N}} \text{(ax)}}{\hline n : \mathbb{N} \vdash \text{Vec } n : \text{Type}} \text{(\Pi}_\text{E})$$
$$\hline n : \mathbb{N}, l : \text{Vec } n \vdash$$



## Rules: equality

The rules for **equality** ensure that we have an equivalence relation

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A}$$

$$\frac{\Gamma \vdash t = u : A}{\Gamma \vdash u = t : A}$$

$$\frac{\Gamma \vdash t = u : A \quad \Gamma \vdash u = v : A}{\Gamma \vdash t = v : A}$$

## Rules: equality

The rules for **equality** ensure that we have an equivalence relation

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A}$$

$$\frac{\Gamma \vdash t = u : A}{\Gamma \vdash u = t : A}$$

$$\frac{\Gamma \vdash t = u : A \quad \Gamma \vdash u = v : A}{\Gamma \vdash t = v : A}$$

and that we can convert terms and types

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A = B : \text{Type}}{\Gamma \vdash t : B}$$

$$\frac{\Gamma \vdash t = u : A \quad \Gamma \vdash A = B : \text{Type}}{\Gamma \vdash t = u : B}$$

## Rules: equality

The rules for **equality** ensure that we have an equivalence relation

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A}$$

$$\frac{\Gamma \vdash t = u : A}{\Gamma \vdash u = t : A}$$

$$\frac{\Gamma \vdash t = u : A \quad \Gamma \vdash u = v : A}{\Gamma \vdash t = v : A}$$

and that we can convert terms and types

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A = B : \text{Type}}{\Gamma \vdash t : B}$$

$$\frac{\Gamma \vdash t = u : A \quad \Gamma \vdash A = B : \text{Type}}{\Gamma \vdash t = u : B}$$

For instance,

$$f : \Pi(x : \text{Vec } 5).A, l : \text{Vec } (3 + 2) \vdash f l : A$$

## Rules: equality

The rules for **equality** ensure that we have an equivalence relation

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A}$$

$$\frac{\Gamma \vdash t = u : A}{\Gamma \vdash u = t : A}$$

$$\frac{\Gamma \vdash t = u : A \quad \Gamma \vdash u = v : A}{\Gamma \vdash t = v : A}$$

and that we can convert terms and types

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A = B : \text{Type}}{\Gamma \vdash t : B}$$

$$\frac{\Gamma \vdash t = u : A \quad \Gamma \vdash A = B : \text{Type}}{\Gamma \vdash t = u : B}$$

For instance,

$$\frac{\frac{}{\dots \vdash f : \Pi(x : \text{Vec } 5).A} \text{ (ax)} \quad \dots \vdash l : \text{Vec } 5}{f : \Pi(x : \text{Vec } 5).A, l : \text{Vec } (3 + 2) \vdash f \, l : A} \text{ (}\Pi_E\text{)}$$

## Rules: equality

The rules for **equality** ensure that we have an equivalence relation

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A}$$

$$\frac{\Gamma \vdash t = u : A}{\Gamma \vdash u = t : A}$$

$$\frac{\Gamma \vdash t = u : A \quad \Gamma \vdash u = v : A}{\Gamma \vdash t = v : A}$$

and that we can convert terms and types

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A = B : \text{Type}}{\Gamma \vdash t : B}$$

$$\frac{\Gamma \vdash t = u : A \quad \Gamma \vdash A = B : \text{Type}}{\Gamma \vdash t = u : B}$$

For instance,

$$\frac{\frac{\dots \vdash f : \Pi(x : \text{Vec } 5).A}{\dots \vdash f : \Pi(x : \text{Vec } 5).A} \text{ (ax)} \quad \frac{\frac{\dots \vdash l : \text{Vec } (3 + 2)}{\dots \vdash l : \text{Vec } 5} \text{ (ax)} \quad \dots \vdash \text{Vec } (3 + 2) = \text{Vec } 5}{\dots \vdash l : \text{Vec } 5} \quad (\Pi_E)}{f : \Pi(x : \text{Vec } 5).A, l : \text{Vec } (3 + 2) \vdash f \, l : A}$$

## Rules: equality

The rules for **equality** ensure that we have an equivalence relation

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A}$$

$$\frac{\Gamma \vdash t = u : A}{\Gamma \vdash u = t : A}$$

$$\frac{\Gamma \vdash t = u : A \quad \Gamma \vdash u = v : A}{\Gamma \vdash t = v : A}$$

and that we can convert terms and types

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A = B : \text{Type}}{\Gamma \vdash t : B}$$

$$\frac{\Gamma \vdash t = u : A \quad \Gamma \vdash A = B : \text{Type}}{\Gamma \vdash t = u : B}$$

For instance,

$$\frac{\frac{\dots \vdash f : \Pi(x : \text{Vec } 5).A}{\dots \vdash f : \Pi(x : \text{Vec } 5).A} \text{ (ax)} \quad \frac{\frac{\dots \vdash l : \text{Vec } (3 + 2)}{\dots \vdash l : \text{Vec } (3 + 2)} \text{ (ax)} \quad \frac{\dots \vdash 3 + 2 = 5 : \mathbb{N}}{\dots \vdash \text{Vec } (3 + 2) = \text{Vec } 5}}{\dots \vdash l : \text{Vec } 5} \quad \frac{\dots \vdash f : \Pi(x : \text{Vec } 5).A \quad \dots \vdash l : \text{Vec } 5}{f : \Pi(x : \text{Vec } 5).A, l : \text{Vec } (3 + 2) \vdash f \, l : A} (\Pi_E)$$

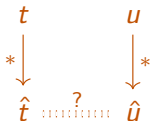
## Rules: equality

In practice, how do we test equality?

## Rules: equality

In practice, how do we test equality?

In order to decide whether  $t$  and  $u$  are equal, we rewrite them as much as we can according to a good orientation of rules:



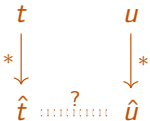
(which includes  $\beta$ -reduction).



## Rules: equality

In practice, how do we test equality?

In order to decide whether  $t$  and  $u$  are equal, we rewrite them as much as we can according to a good orientation of rules:



(which includes  $\beta$ -reduction).

This means that we must ensure that the corresponding reduction is

- terminating, and
- (locally) confluent.

## Rules: axiom

The **axiom** rule is

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A} \text{(ax)}$$

## Rules: axiom

The **axiom** rule is

$$\frac{\Gamma, x : A, \Delta \vdash}{\Gamma, x : A, \Delta \vdash x : A} (\text{ax})$$

## Rules: axiom

The **axiom** rule is

$$\frac{\Gamma, x : A, \Delta \vdash}{\Gamma, x : A, \Delta \vdash x : A} \text{(ax)}$$

whenever  $x \notin \text{dom}(\Delta)$  and  $\text{dom}(\Delta) \cap \text{FV}(A) = \emptyset$ .

## Rules: axiom

The **axiom** rule is

$$\frac{\Gamma, x : A, \Delta \vdash}{\Gamma, x : A, \Delta \vdash x : A} (\text{ax})$$

whenever  $x \notin \text{dom}(\Delta)$  and  $\text{dom}(\Delta) \cap \text{FV}(A) = \emptyset$ .

The side conditions avoid bad jokes such as

$$\frac{}{n : \mathbb{N}, l : \text{Vec } n, n : \text{Bool} \vdash l : \text{Vec } n} (\text{ax})$$

## Rules: $\Pi$ -types

The last thing we need to introduce is rules for  $\Pi$ -types.

As for all other type constructors, we will need to specify six families of rules:

## Rules: $\Pi$ -types

The last thing we need to introduce is rules for  $\Pi$ -types.

As for all other type constructors, we will need to specify six families of rules:

1. *formation*: when the type is well-formed,

## Rules: $\Pi$ -types

The last thing we need to introduce is rules for  $\Pi$ -types.

As for all other type constructors, we will need to specify six families of rules:

1. *formation*: when the type is well-formed,
2. *introduction*: introduce a term of this type,



## Rules: $\Pi$ -types

The last thing we need to introduce is rules for  $\Pi$ -types.

As for all other type constructors, we will need to specify six families of rules:

1. *formation*: when the type is well-formed,
2. *introduction*: introduce a term of this type,
3. *elimination*: use a term of this type,

## Rules: $\Pi$ -types

The last thing we need to introduce is rules for  $\Pi$ -types.

As for all other type constructors, we will need to specify six families of rules:

1. *formation*: when the type is well-formed,
2. *introduction*: introduce a term of this type,
3. *elimination*: use a term of this type,
4. *computation*: the deconstruction of a constructed term ( $\beta$ -equivalence),

## Rules: $\Pi$ -types

The last thing we need to introduce is rules for  $\Pi$ -types.

As for all other type constructors, we will need to specify six families of rules:

1. *formation*: when the type is well-formed,
2. *introduction*: introduce a term of this type,
3. *elimination*: use a term of this type,
4. *computation*: the deconstruction of a constructed term ( $\beta$ -equivalence),
5. *uniqueness*: reconstructing a deconstructed term does nothing ( $\eta$ -equivalence),

## Rules: $\Pi$ -types

The last thing we need to introduce is rules for  $\Pi$ -types.

As for all other type constructors, we will need to specify six families of rules:

1. *formation*: when the type is well-formed,
2. *introduction*: introduce a term of this type,
3. *elimination*: use a term of this type,
4. *computation*: the deconstruction of a constructed term ( $\beta$ -equivalence),
5. *uniqueness*: reconstructing a deconstructed term does nothing ( $\eta$ -equivalence),
6. *congruence*: compatibility with equality.

## Rules: $\Pi$ -types

Formation:

## Rules: $\Pi$ -types

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Pi(x : A).B : \text{Type}} (\Pi_F)$$

For instance,

$$\frac{\frac{}{\vdash \mathbb{N} : \text{Type}} \quad \frac{\vdots}{n : \mathbb{N} \vdash \text{Vec } n : \text{Type}}}{\vdash \Pi(n : \mathbb{N}). \text{Vec } n : \text{Type}} (\Pi_F)$$

## Rules: $\Pi$ -types

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Pi(x : A).B : \text{Type}} (\Pi_F)$$

But we cannot derive

$$\frac{\frac{}{\vdash \text{Bool} : \text{Type}} \quad \frac{\vdots}{n : \text{Bool} \vdash \text{Vec } n : \text{Type}}}{\vdash \Pi(n : \text{Bool}). \text{Vec } n : \text{Type}} (\Pi_F)$$

## Rules: $\Pi$ -types

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Pi(x : A).B : \text{Type}} (\Pi_F)$$

Introduction:



## Rules: $\Pi$ -types

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Pi(x : A).B : \text{Type}} (\Pi_F)$$

Introduction:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A. t : \Pi(x : A).B} (\Pi_I)$$

For instance,

$$\frac{n : \mathbb{N} \vdash l : \text{Vec } n}{\vdash \lambda n^{\mathbb{N}}. l : \Pi(n : \mathbb{N}). \text{Vec } n} (\Pi_I)$$

## Rules: $\Pi$ -types

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Pi(x : A).B : \text{Type}} (\Pi_F)$$

Introduction:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : \Pi(x : A).B} (\Pi_I)$$

Elimination:

## Rules: $\Pi$ -types

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Pi(x : A).B : \text{Type}} (\Pi_F)$$

Introduction:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A. t : \Pi(x : A).B} (\Pi_I)$$

Elimination:

$$\frac{\Gamma \vdash t : \Pi(x : A).B \quad \Gamma \vdash u : A}{\Gamma \vdash t u : B[u/x]} (\Pi_E)$$

For instance,

$$\frac{\frac{f : \Pi(n : \mathbb{N}). \text{Vec } n \vdash f : \Pi(n : \mathbb{N}). \text{Vec } n}{f : \Pi(n : \mathbb{N}). \text{Vec } n \vdash f 5 : \text{Vec } 5} (\text{ax}) \quad \dots \vdash 5 : \mathbb{N}}{f : \Pi(n : \mathbb{N}). \text{Vec } n \vdash f 5 : \text{Vec } 5} (\Pi_E)$$

## Rules: $\Pi$ -types

Computation:

## Rules: $\Pi$ -types

Computation:

$$\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x^A. t) u = t[u/x] : B[u/x]} (\Pi_C)$$

(this is  $\beta$ -equivalence!)

## Rules: $\Pi$ -types

Computation:

$$\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x^A. t) u = t[u/x] : B[u/x]} (\Pi_C)$$

(this is  $\beta$ -equivalence!)

Uniqueness:

## Rules: $\Pi$ -types

Computation:

$$\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x^A. t) u = t[u/x] : B[u/x]} (\Pi_C)$$

(this is  $\beta$ -equivalence!)

Uniqueness:

$$\frac{\Gamma \vdash t : \Pi(x : A). B}{\Gamma \vdash t = \lambda x^A. t x : \Pi(x : A). B} (\Pi_U)$$

(this is  $\eta$ -equivalence!)

## Rules: $\Pi$ -types

Congruence:



## Rules: $\Pi$ -types

Congruence:

$$\frac{\Gamma \vdash A = A' : \text{Type} \quad \Gamma, x : A \vdash B = B' : \text{Type}}{\Gamma \vdash \Pi(x : A).B = \Pi(x : A').B' : \text{Type}}$$

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type} \quad \Gamma \vdash t = t' : B}{\Gamma \vdash \lambda x^A.t = \lambda x^A.t' : \Pi(x : A).B}$$

In the following, we will usually omit congruence rules.

# Arrow types

If we write

$$A \rightarrow B$$

for

$$\Pi(x : A).B$$

for some variable  $x \notin FV(B)$ , we have

$$B[t/x] = B$$

and we recover the usual rules.

For instance, the elimination rule

$$\frac{\Gamma \vdash t : \Pi(x : A).B \quad \Gamma \vdash u : A}{\Gamma \vdash t u : B[u/x]} (\Pi_E)$$

For instance, the elimination rule

$$\frac{\Gamma \vdash t : \Pi(x : A).B \quad \Gamma \vdash u : A}{\Gamma \vdash t u : B[u/x]} (\Pi_E)$$

becomes

$$\frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash t u : B} (\rightarrow_E)$$

## Part II

# More types

## More type constructors

More type constructors can be added by adding constructions to expressions: we need

- a constructor for types (e.g.  $\Pi$ ),
- a constructor for terms (e.g.  $\lambda$ ),
- an eliminator (e.g. application)

and associated typing rules.

We will see that all the added features can be encoded with inductive types, but let's take it slowly.

# The empty type

Formation:

# The empty type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \perp : \text{Type}} (\perp_F)$$



# The empty type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \perp : \text{Type}} (\perp_F)$$

Introduction:

# The empty type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \perp : \text{Type}} (\perp_F)$$

Introduction: n/a

# The empty type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \perp : \text{Type}} (\perp_F)$$

Introduction: n/a

Elimination:

# The empty type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \perp : \text{Type}} (\perp_F)$$

Introduction: n/a

Elimination:

$$\frac{\Gamma \vdash t : \perp \quad \Gamma, x : \perp \vdash A : \text{Type}}{\Gamma \vdash \text{bot}(t, x \mapsto A) : A[t/x]} (\top_E)$$

# The empty type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \perp : \text{Type}} (\perp_F)$$

Introduction: n/a

Elimination:

$$\frac{\Gamma \vdash t : \perp \quad \Gamma, x : \perp \vdash A : \text{Type}}{\Gamma \vdash \text{bot}(t, x \mapsto A) : A[t/x]} (\top_E)$$

Computation:

# The empty type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \perp : \text{Type}} (\perp_F)$$

Introduction: n/a

Elimination:

$$\frac{\Gamma \vdash t : \perp \quad \Gamma, x : \perp \vdash A : \text{Type}}{\Gamma \vdash \text{bot}(t, x \mapsto A) : A[t/x]} (\top_E)$$

Computation: n/a

# The empty type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \perp : \text{Type}} (\perp_F)$$

Introduction: n/a

Elimination:

$$\frac{\Gamma \vdash t : \perp \quad \Gamma, x : \perp \vdash A : \text{Type}}{\Gamma \vdash \text{bot}(t, x \mapsto A) : A[t/x]} (\top_E)$$

Computation: n/a

Uniqueness:

# The empty type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \perp : \text{Type}} (\perp_F)$$

Introduction: n/a

Elimination:

$$\frac{\Gamma \vdash t : \perp \quad \Gamma, x : \perp \vdash A : \text{Type}}{\Gamma \vdash \text{bot}(t, x \mapsto A) : A[t/x]} (\top_E)$$

Computation: n/a

Uniqueness: n/a



# The empty type

In Agda, elimination and computation correspond to

# The empty type

In Agda, elimination and computation correspond to

```
 $\perp$ -elim : (A :  $\perp$  → Set) → (x :  $\perp$ ) → A x  
 $\perp$ -elim A ()
```

# The unit type

Formation:

In OCaml

# The unit type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \top : \text{Type}} (\top_F)$$

In OCaml

- `⊤` is `unit`

# The unit type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \top : \text{Type}} (\top_F)$$

Introduction:

In OCaml

- `⊤` is `unit`

# The unit type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \top : \text{Type}} (\top_F)$$

Introduction:

$$\frac{\Gamma \vdash}{\Gamma \vdash \star : \top} (\top_I)$$

In OCaml

- $\top$  is `unit`
- $\star$  is `()`

# The unit type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \top : \text{Type}} (\top_F)$$

Introduction:

$$\frac{\Gamma \vdash}{\Gamma \vdash \star : \top} (\top_I)$$

Elimination:

In OCaml

- $\top$  is `unit`
- $\star$  is `()`

# The unit type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \top : \text{Type}} (\top_F)$$

Introduction:

$$\frac{\Gamma \vdash}{\Gamma \vdash \star : \top} (\top_I)$$

Elimination:

$$\frac{\Gamma \vdash t : \top \quad \Gamma, x : \top \vdash A : \text{Type} \quad \Gamma \vdash u : A[\star/x]}{\Gamma \vdash \text{top}(t, x \mapsto A, u) : A[t/x]} (\top_E)$$

In OCaml

- $\top$  is `unit`
- $\star$  is `()`
- `top(t, x ↦ A, u)` is `match t with () -> u`



# The unit type

Computation:

# The unit type

Computation:

$$\frac{\Gamma, x : \top \vdash A : \text{Type} \quad \Gamma \vdash u : A[\star/x]}{\Gamma \vdash \text{top}(\star, x \mapsto A, u) = u : A[\star/x]} (\top_c)$$

In OCaml,

```
match () with () -> u = u
```

# The unit type

Computation:

$$\frac{\Gamma, x : \top \vdash A : \text{Type} \quad \Gamma \vdash u : A[\star/x]}{\Gamma \vdash \text{top}(\star, x \mapsto A, u) = u : A[\star/x]} \quad (\top_c)$$

Uniqueness:

In OCaml,

```
match () with () -> u = u
```

# The unit type

Computation:

$$\frac{\Gamma, x : \top \vdash A : \text{Type} \quad \Gamma \vdash u : A[\star/x]}{\Gamma \vdash \text{top}(\star, x \mapsto A, u) = u : A[\star/x]} \quad (\top_c)$$

Uniqueness:

$$\frac{\Gamma \vdash t : \top}{\Gamma \vdash t = \star : \top} \quad (\top_u)$$

In OCaml,

```
match () with () -> u   =   u
                t   =   ()   for t of type unit
```

# The unit type

In Agda, elimination and computation correspond to

# The unit type

In Agda, elimination and computation correspond to

```
 $\top$ -elim : (A :  $\top \rightarrow \text{Set}$ )  $\rightarrow$  A tt  $\rightarrow$  (t :  $\top$ )  $\rightarrow$  A t
```

```
 $\top$ -elim A a tt = a
```

# Booleans

Formation:

In OCaml:

# Booleans

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Bool} : \text{Type}} (\text{Bool}_F)$$

In OCaml:



# Booleans

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Bool} : \text{Type}} \text{ (Bool}_F\text{)}$$

Introduction:

In OCaml:

# Booleans

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Bool} : \text{Type}} (\text{Bool}_F)$$

Introduction:

$$\frac{\Gamma \vdash}{\Gamma \vdash 1 : \text{Bool}} (\text{Bool}_I^1)$$

$$\frac{\Gamma \vdash}{\Gamma \vdash 0 : \text{Bool}} (\text{Bool}_I^0)$$

In OCaml:

- 1 is `true` and 0 is `false`

# Booleans

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Bool} : \text{Type}} (\text{Bool}_F)$$

Introduction:

$$\frac{\Gamma \vdash}{\Gamma \vdash 1 : \text{Bool}} (\text{Bool}_I^1)$$

$$\frac{\Gamma \vdash}{\Gamma \vdash 0 : \text{Bool}} (\text{Bool}_I^0)$$

Elimination:

In OCaml:

- 1 is `true` and 0 is `false`

# Booleans

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Bool} : \text{Type}} (\text{Bool}_F)$$

Introduction:

$$\frac{\Gamma \vdash}{\Gamma \vdash 1 : \text{Bool}} (\text{Bool}_I^1)$$

$$\frac{\Gamma \vdash}{\Gamma \vdash 0 : \text{Bool}} (\text{Bool}_I^0)$$

Elimination:

$$\frac{\Gamma \vdash t : \text{Bool} \quad \Gamma, x : \text{Bool} \vdash A : \text{Type} \quad \Gamma \vdash u : A[1/x] \quad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \text{ite}(t, x \mapsto A, u, v) : A[t/x]} (\text{Bool}_E)$$

In OCaml:

- 1 is **true** and 0 is **false**
- $\text{ite}(t, x \mapsto A, u, v)$  is if  $t$  then  $u$  else  $v$

Computation:

# Booleans

Computation:

$$\frac{\Gamma, x : \text{Bool} \vdash A : \text{Type} \quad \Gamma \vdash u : A[1/x] \quad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \text{ite}(1, x \mapsto A, u, v) = u : A[1/x]} \text{ (Bool}_C^1\text{)}$$

$$\frac{\Gamma, x : \text{Bool} \vdash A : \text{Type} \quad \Gamma \vdash u : A[1/x] \quad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \text{ite}(0, x \mapsto A, u, v) = v : A[0/x]} \text{ (Bool}_C^0\text{)}$$

In OCaml,

```
if true then u else v = u  
if false then u else v = v
```

# Booleans

Computation:

$$\frac{\Gamma, x : \text{Bool} \vdash A : \text{Type} \quad \Gamma \vdash u : A[1/x] \quad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \text{ite}(1, x \mapsto A, u, v) = u : A[1/x]} \text{ (Bool}_C^1\text{)}$$

$$\frac{\Gamma, x : \text{Bool} \vdash A : \text{Type} \quad \Gamma \vdash u : A[1/x] \quad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \text{ite}(0, x \mapsto A, u, v) = v : A[0/x]} \text{ (Bool}_C^0\text{)}$$

Uniqueness:

In OCaml,

```
if true then u else v = u
if false then u else v = v
```

# Booleans

Computation:

$$\frac{\Gamma, x : \text{Bool} \vdash A : \text{Type} \quad \Gamma \vdash u : A[1/x] \quad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \text{ite}(1, x \mapsto A, u, v) = u : A[1/x]} \text{ (Bool}_C^1\text{)}$$

$$\frac{\Gamma, x : \text{Bool} \vdash A : \text{Type} \quad \Gamma \vdash u : A[1/x] \quad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \text{ite}(0, x \mapsto A, u, v) = v : A[0/x]} \text{ (Bool}_C^0\text{)}$$

Uniqueness:

$$\frac{\Gamma \vdash t : \text{Bool}}{\Gamma \vdash \text{ite}(t, x \mapsto \text{Bool}, 1, 0) = t : \text{Bool}} \text{ (Bool}_U\text{)}$$

In OCaml,

```
if true then u else v = u      if t then true else false = t
if false then u else v = v
```



In Agda, elimination and computation correspond to

In Agda, elimination and computation correspond to

```
Bool-elim : (A : Bool → Set) → A true → A false → (b : Bool) → A b
Bool-elim A t f true  = t
Bool-elim A t f false = f
```

# Natural numbers

Formation:

# Natural numbers

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Nat} : \text{Type}} (\text{Nat}_F)$$

# Natural numbers

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Nat} : \text{Type}} (\text{Nat}_F)$$

Introduction:

# Natural numbers

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Nat} : \text{Type}} (\text{Nat}_F)$$

Introduction:

$$\frac{\Gamma \vdash}{\Gamma \vdash Z : \text{Nat}} (\text{Nat}_I^Z)$$

$$\frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash S(t) : \text{Nat}} (\text{Nat}_I^S)$$

In OCaml,  $Z$  is  $0$  and  $S(n)$  is  $n+1$ .

# Natural numbers

Elimination:

# Natural numbers

Elimination:

$$\frac{\Gamma \vdash t : \text{Nat} \quad \Gamma, x : \text{Nat} \vdash A : \text{Type} \quad \Gamma \vdash u : A[Z/x] \quad \Gamma, x : \text{Nat}, y : A \vdash v : A[S(x)/x]}{\Gamma \vdash \text{rec}(t, x \mapsto A, u, xy \mapsto v) : A[t/x]} \text{ (Nat}_E\text{)}$$

In OCaml,  $\text{rec}(t, x \mapsto A, u, xy \mapsto v)$  is

```
let rec natrec t u v =  
  if t = 0 then u  
  else v (t-1) (natrec (t-1) u v)
```



For instance, the factorial function can be implemented with

```
let fact n =  
  natrec n 1 (fun n r -> (n+1) * r)
```

# Natural numbers

Computation:

# Natural numbers

Computation:

$$\frac{\Gamma, x : \text{Nat} \vdash A : \text{Type} \quad \Gamma \vdash u : A[Z/x] \quad \Gamma, x : \text{Nat}, y : A \vdash v : A[S(x)/x]}{\Gamma \vdash \text{rec}(Z, x \mapsto A, u, xy \mapsto v) = u : A[Z/x]} \quad (\text{Nat}_C^Z)$$

In OCaml,

```
natrec 0 u v = u
```

# Natural numbers

Computation:

$$\frac{\Gamma, x : \text{Nat} \vdash A : \text{Type} \quad \Gamma \vdash u : A[Z/x] \quad \Gamma, x : \text{Nat}, y : A \vdash v : A[S(x)/x]}{\Gamma \vdash \text{rec}(Z, x \mapsto A, u, xy \mapsto v) = u : A[Z/x]} \quad (\text{Nat}_C^Z)$$

$$\frac{\Gamma \vdash t : \text{Nat} \quad \Gamma, x : \text{Nat} \vdash A : \text{Type} \quad \Gamma \vdash u : A[Z/x] \quad \Gamma, x : \text{Nat}, y : A \vdash v : A[S(x)/x]}{\Gamma \vdash \text{rec}(S(t), x \mapsto A, u, xy \mapsto v) = v[t/x, \text{rec}(t, x \mapsto A, u, xy \mapsto v)/y] : A[S(t)/x]} \quad (\text{Nat}_C^S)$$

In OCaml,

$$\begin{aligned} \text{natrec } 0 \ u \ v &= u \\ \text{natrec } (n+1) \ u \ v &= v \ n \ (\text{natrec } n \ u \ v) \end{aligned}$$

# Natural numbers

In Agda,

# Natural numbers

In Agda,

```
 $\mathbb{N}$ -elim : (A :  $\mathbb{N} \rightarrow \text{Set}$ )  $\rightarrow$   
          A zero  $\rightarrow$  ((n :  $\mathbb{N}$ )  $\rightarrow$  A n  $\rightarrow$  A (suc n))  $\rightarrow$  (n :  $\mathbb{N}$ )  $\rightarrow$  A n  
 $\mathbb{N}$ -elim A z s zero      = z  
 $\mathbb{N}$ -elim A z s (suc n) = s n ( $\mathbb{N}$ -elim A z s n)
```

# Natural numbers

In Agda,

```
ℕ-elim : (A : ℕ → Set) →  
          A zero → ((n : ℕ) → A n → A (suc n)) → (n : ℕ) → A n  
ℕ-elim A z s zero      = z  
ℕ-elim A z s (suc n) = s n (ℕ-elim A z s n)
```

The generation of a vector of length  $n$  :

```
fill : {A : Set} (a : A) (n : ℕ) → Vec A n  
fill a zero      = []  
fill a (suc n) = a :: fill a n
```

# Natural numbers

In Agda,

```
 $\mathbb{N}$ -elim : (A :  $\mathbb{N} \rightarrow \text{Set}$ )  $\rightarrow$   
           A zero  $\rightarrow$  ((n :  $\mathbb{N}$ )  $\rightarrow$  A n  $\rightarrow$  A (suc n))  $\rightarrow$  (n :  $\mathbb{N}$ )  $\rightarrow$  A n  
 $\mathbb{N}$ -elim A z s zero      = z  
 $\mathbb{N}$ -elim A z s (suc n) = s n ( $\mathbb{N}$ -elim A z s n)
```

The generation of a vector of length  $n$  :

```
fill : {A : Set} (a : A) (n :  $\mathbb{N}$ )  $\rightarrow$  Vec A n  
fill {A} a n =  $\mathbb{N}$ -elim ( $\lambda$  n  $\rightarrow$  Vec A n) [] ( $\lambda$  n l  $\rightarrow$  a :: l) n
```



# Natural numbers

Uniqueness:

# Natural numbers

Uniqueness:

$$\frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash \text{rec}(t, x \mapsto \text{Nat}, Z, xy \mapsto S(y)) = t : \text{Nat}} \text{ (Nat}_U\text{)}$$

In OCaml,

```
natrec n 0 (fun x y -> y + 1) = n
```

i.e. we can program the identity on natural numbers as

```
let rec id n =  
  if n = 0 then 0  
  else (id (n-1)) + 1
```

# Products

Formation:

# Products

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash B : \text{Type}}{\Gamma \vdash A \times B : \text{Type}} (\times_F)$$

# Products

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash B : \text{Type}}{\Gamma \vdash A \times B : \text{Type}} (\times_F)$$

Introduction:

# Products

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash B : \text{Type}}{\Gamma \vdash A \times B : \text{Type}} (\times_F)$$

Introduction:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B} (\times_I)$$

# Products

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash B : \text{Type}}{\Gamma \vdash A \times B : \text{Type}} (\times_F)$$

Introduction:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B} (\times_I)$$

Elimination:

# Products

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash B : \text{Type}}{\Gamma \vdash A \times B : \text{Type}} (\times_F)$$

Introduction:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B} (\times_I)$$

Elimination:

$$\frac{\Gamma \vdash t : A \times B \quad \Gamma, z : A \times B \vdash C : \text{Type} \quad \Gamma, x : A, y : B \vdash u : C[\langle x, y \rangle / z]}{\Gamma \vdash \text{unpair}(t, z \mapsto C, \langle x, y \rangle \mapsto u) : C[t/z]} (\times_E)$$

In OCaml,

$$\text{unpair}(t, z \mapsto C, \langle x, y \rangle \mapsto u) = \text{match } t \text{ with } (x, y) \rightarrow u$$



Computation:

Computation:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B \quad \Gamma, z : A \times B \vdash C : \text{Type} \quad \Gamma, x : A, y : B \vdash v : C[\langle x, y \rangle / z]}{\Gamma \vdash \text{unpair}(\langle t, u \rangle, z \mapsto C, \langle x, y \rangle \mapsto v) = v[t/x, u/y] : C[\langle t, u \rangle / z]} (\times_c)$$

In OCaml,

```
match (t,u) with (x,y) -> v x y = v t u
```

# Products

Computation:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B \quad \Gamma, z : A \times B \vdash C : \text{Type} \quad \Gamma, x : A, y : B \vdash v : C[\langle x, y \rangle / z]}{\Gamma \vdash \text{unpair}(\langle t, u \rangle, z \mapsto C, \langle x, y \rangle \mapsto v) = v[t/x, u/y] : C[\langle t, u \rangle / z]} (\times_c)$$

Uniqueness:

In OCaml,

```
match (t,u) with (x,y) -> v x y = v t u
```

# Products

Computation:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B \quad \Gamma, z : A \times B \vdash C : \text{Type} \quad \Gamma, x : A, y : B \vdash v : C[\langle x, y \rangle / z]}{\Gamma \vdash \text{unpair}(\langle t, u \rangle, z \mapsto C, \langle x, y \rangle \mapsto v) = v[t/x, u/y] : C[\langle t, u \rangle / z]} (\times_c)$$

Uniqueness:

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \text{unpair}(t, z \mapsto A \times B, \langle x, y \rangle \mapsto \langle x, y \rangle) = t : A \times B} (\times_u)$$

In OCaml,

```
match (t,u) with (x,y) -> v x y = v t u
match t with (x,y) -> (x,y) = t
```

## Dependent sums

In a similar way, we can consider dependent sums types

$$\Sigma(x : A).B$$

whose terms are pairs  $\langle t, u \rangle$  with

$$t : A$$

and

$$u : B[t/x]$$

## Dependent sums

In a similar way, we can consider dependent sums types

$$\Sigma(x : A).B$$

whose terms are pairs  $\langle t, u \rangle$  with

$$t : A \quad \text{and} \quad u : B[t/x]$$

From a logical point of view, this can be read as

$$\exists x \in A. B$$

## Dependent sums

In a similar way, we can consider dependent sums types

$$\Sigma(x : A).B$$

whose terms are pairs  $\langle t, u \rangle$  with

$$t : A \quad \text{and} \quad u : B[t/x]$$

From a logical point of view, this can be read as

$$\exists x \in A. B$$

For instance, the type  $V_A$  of all vectors of type  $A$  can be defined as

$$V_A = \Sigma(n : \text{Nat}). \text{Vec } A \ n$$

## Dependent sums

A product is a particular case of sum of a constant finite sequence:

$$m \times n = \sum_{1 \leq i \leq m} n$$

where we can more generally consider

$$\sum_{1 \leq i \leq m} n_i$$

for a finite sequence  $(n_i)_{1 \leq i \leq m}$ .



## Dependent sums

A product is a particular case of sum of a constant finite sequence:

$$m \times n = \sum_{1 \leq i \leq m} n$$

where we can more generally consider

$$\sum_{1 \leq i \leq m} n_i$$

for a finite sequence  $(n_i)_{1 \leq i \leq m}$ .

Similarly, in type theory, we have

$$A \times B = \Sigma(x : A).B$$

when  $x \notin \text{FV}(B)$ .

## Dependent sums

Formation:

## Dependent sums

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Sigma(x : A).B : \text{Type}} \quad (\Sigma_F)$$

## Dependent sums

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Sigma(x : A).B : \text{Type}} \quad (\Sigma_F)$$

Introduction:

## Dependent sums

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Sigma(x : A).B : \text{Type}} \quad (\Sigma_F)$$

Introduction:

$$\frac{\Gamma, x : A \vdash B : \text{Type} \quad \Gamma \vdash t : A \quad \Gamma \vdash u : B[t/x]}{\Gamma \vdash \langle t, u \rangle : \Sigma(x : A).B} \quad (\Sigma_I)$$

## Dependent sums

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Sigma(x : A).B : \text{Type}} \quad (\Sigma_F)$$

Introduction:

$$\frac{\Gamma, x : A \vdash B : \text{Type} \quad \Gamma \vdash t : A \quad \Gamma \vdash u : B[t/x]}{\Gamma \vdash \langle t, u \rangle : \Sigma(x : A).B} \quad (\Sigma_I)$$

Elimination:

## Dependent sums

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Sigma(x : A).B : \text{Type}} \quad (\Sigma_F)$$

Introduction:

$$\frac{\Gamma, x : A \vdash B : \text{Type} \quad \Gamma \vdash t : A \quad \Gamma \vdash u : B[t/x]}{\Gamma \vdash \langle t, u \rangle : \Sigma(x : A).B} \quad (\Sigma_I)$$

Elimination:

$$\frac{\Gamma \vdash t : \Sigma(x : A).B \quad \Gamma, z : \Sigma(x : A).B \vdash C : \text{Type} \quad \Gamma, x : A, y : B \vdash u : C[\langle x, y \rangle / z]}{\Gamma \vdash \text{unpair}(t, z \mapsto C, \langle x, y \rangle \mapsto u) : C[t/z]} \quad (\Pi_E)$$

## Dependent sums

Computation:



Computation:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B[t/x] \quad \Gamma, z : \Sigma(x : A).B \vdash C : \text{Type} \quad \Gamma, x : A, y : B \vdash v : C[\langle x, y \rangle / z]}{\Gamma \vdash \text{unpair}(\langle t, u \rangle, z \mapsto C, \langle x, y \rangle \mapsto v) = v[t/x, u/y] : C[\langle t, u \rangle / z]} (\Pi_C)$$

Computation:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B[t/x] \quad \Gamma, z : \Sigma(x : A).B \vdash C : \text{Type} \quad \Gamma, x : A, y : B \vdash v : C[\langle x, y \rangle / z]}{\Gamma \vdash \text{unpair}(\langle t, u \rangle, z \mapsto C, \langle x, y \rangle \mapsto v) = v[t/x, u/y] : C[\langle t, u \rangle / z]} (\Pi_C)$$

Uniqueness:

## Dependent sums

Computation:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B[t/x] \quad \Gamma, z : \Sigma(x : A).B \vdash C : \text{Type} \quad \Gamma, x : A, y : B \vdash v : C[\langle x, y \rangle / z]}{\Gamma \vdash \text{unpair}(\langle t, u \rangle, z \mapsto C, \langle x, y \rangle \mapsto v) = v[t/x, u/y] : C[\langle t, u \rangle / z]} (\Pi_C)$$

Uniqueness:

$$\frac{\Gamma \vdash t : \Sigma(x : A).B}{\Gamma \vdash \text{unpair}(t, z \mapsto \Sigma(x : A).B, \langle x, y \rangle \mapsto \langle x, y \rangle) = t : \Sigma(x : A).B} (\Pi_U)$$

One can also add **inductive types** to the theory (as in Agda).

In this case, the above constructions become definable (as in Agda).

We shall see this later on.

## Lemma

*If  $\Gamma \vdash t : A$  is derivable then both  $\Gamma \vdash$  and  $\Gamma \vdash A : \text{Type}$  are derivable.*

## Proof.

By induction on the proof.



NB: this actually *almost* true, see next slides.

## Part III

# Typing Type

## A slight problem

Previous theorem is not exactly true, we can derive

$$\frac{}{\Gamma \vdash \text{Nat} : \text{Type}}$$

but not

$$\frac{}{\Gamma \vdash \text{Type} : \text{Type}}$$

# Type constructors

We want to manipulate terms of type `Type`.

Type constructors typically have type

$$\text{Type} \rightarrow \text{Type} \rightarrow \dots \rightarrow \text{Type}$$

For instance,

$$\neg : \text{Type} \rightarrow \text{Type}$$

In Agda,

$$\neg : (A : \text{Set}) \rightarrow \text{Set}$$
$$\neg A = A \rightarrow \perp$$



# Type constructors

We want to manipulate terms of type `Type`.

A polymorphic identity function has type

$$\text{id} : \Pi(A : \text{Type}). A \rightarrow A$$

In Agda:

```
id : (A : Set) → A → A
```

```
id A a = a
```

$$\vdash \lambda A^{\text{Type}}. \lambda x^A. x : \Pi(A : \text{Type}). A \rightarrow A$$

## Typing polymorphic identity

$$\frac{A : \text{Type} \vdash \lambda x^A. x : A \rightarrow A}{\vdash \lambda A^{\text{Type}}. \lambda x^A. x : \Pi(A : \text{Type}). A \rightarrow A} \quad (\Pi_I)$$

## Typing polymorphic identity

$$\frac{\frac{A : \text{Type}, x : A \vdash x : A}{A : \text{Type} \vdash \lambda x^A. x : A \rightarrow A} (\rightarrow_I)}{\vdash \lambda A^{\text{Type}}. \lambda x^A. x : \Pi(A : \text{Type}). A \rightarrow A} (\Pi_I)$$

## Typing polymorphic identity

$$\frac{\frac{\frac{A : \text{Type}, x : A \vdash}{A : \text{Type}, x : A \vdash x : A} (\text{ax})}{A : \text{Type} \vdash \lambda x^A. x : A \rightarrow A} (\rightarrow_I)}{\vdash \lambda A^{\text{Type}}. \lambda x^A. x : \Pi(A : \text{Type}). A \rightarrow A} (\Pi_I)$$

## Typing polymorphic identity

$$\frac{\frac{\frac{A : \text{Type} \vdash A : \text{Type}}{A : \text{Type}, x : A \vdash} \quad (\text{ax})}{A : \text{Type}, x : A \vdash x : A} \quad (\rightarrow_I)}{A : \text{Type} \vdash \lambda x^A. x : A \rightarrow A} \quad (\Pi_I)$$
$$\vdash \lambda A^{\text{Type}}. \lambda x^A. x : \Pi(A : \text{Type}). A \rightarrow A$$

## Typing polymorphic identity

$$\frac{\frac{\frac{A : \text{Type} \vdash}{A : \text{Type} \vdash A : \text{Type}} \text{ (ax)}}{A : \text{Type}, x : A \vdash x : A} \text{ (ax)}}{A : \text{Type} \vdash \lambda x^A. x : A \rightarrow A} \text{ (}\rightarrow\text{I)} \\ \vdash \lambda A^{\text{Type}}. \lambda x^A. x : \Pi(A : \text{Type}). A \rightarrow A \text{ (}\Pi\text{I)}$$

## Typing polymorphic identity

$$\frac{\frac{\frac{\frac{\vdash \text{Type} : \text{Type}}{A : \text{Type} \vdash} \text{ (ax)}}{A : \text{Type} \vdash A : \text{Type}} \text{ (ax)}}{A : \text{Type}, x : A \vdash} \text{ (ax)}}{A : \text{Type}, x : A \vdash x : A} \text{ (ax)} \quad \frac{}{A : \text{Type} \vdash \lambda x^A. x : A \rightarrow A} \text{ (}\rightarrow\text{I)} \quad \frac{}{\vdash \lambda A^{\text{Type}}. \lambda x^A. x : \Pi(A : \text{Type}). A \rightarrow A} \text{ (}\Pi\text{I)}$$



## Typing polymorphic identity

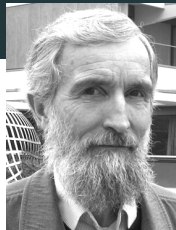
$$\frac{\frac{\frac{???}{\vdash \text{Type} : \text{Type}}}{A : \text{Type} \vdash} \quad (\text{ax})}{A : \text{Type} \vdash A : \text{Type}} \quad (\text{ax})$$
$$\frac{\frac{A : \text{Type}, x : A \vdash} \quad (\text{ax})}{A : \text{Type}, x : A \vdash x : A} \quad (\text{ax})}{A : \text{Type} \vdash \lambda x^A. x : A \rightarrow A} \quad (\rightarrow_I)$$
$$\frac{}{\vdash \lambda A^{\text{Type}}. \lambda x^A. x : \Pi(A : \text{Type}). A \rightarrow A} \quad (\Pi_I)$$

# The obvious guess

The obvious guess consists in adding the rule

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Type} : \text{Type}}$$

and this was present in Martin-Löf's original type system (1971).



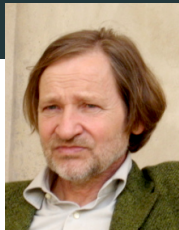
# The obvious guess

The obvious guess consists in adding the rule

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Type} : \text{Type}}$$

and this was present in Martin-Löf's original type system (1971).

However, Girard showed that this was inconsistent, by analogy with the fact that a theory of sets with a set of all sets is inconsistent.



# The obvious guess

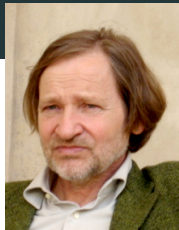
The obvious guess consists in adding the rule

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Type} : \text{Type}}$$

and this was present in Martin-Löf's original type system (1971).

However, Girard showed that this was inconsistent, by analogy with the fact that a theory of sets with a set of all sets is inconsistent.

Technically, his paradox was based on the one of Burali-Forti (there is no *set* of all ordinals), we will see a simpler variant in the case where we have inductive types.



# The obvious guess

The obvious guess consists in adding the rule

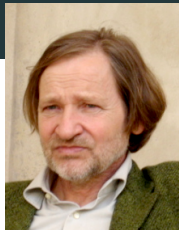
$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Type} : \text{Type}}$$

and this was present in Martin-Löf's original type system (1971).

However, Girard showed that this was inconsistent, by analogy with the fact that a theory of sets with a set of all sets is inconsistent.

Technically, his paradox was based on the one of Burali-Forti (there is no *set* of all ordinals), we will see a simpler variant in the case where we have inductive types.

Let's see how to encode set theory in type theory!



# Finite collections in OCaml

We want to implement finite collections of elements of type `'a` in OCaml.

We can define:

```
type 'a fincol = 'a array
```

# Finite collections in OCaml

We want to implement finite collections of elements of type `'a` in OCaml.

We can define:

```
type 'a fincol = 'a array
```

Note that

`[|1;2;3|]`      and      `[|1;3;2;2;1|]`

represent the same set!

# Finite collections in OCaml

We want to implement finite collections of elements of type `'a` in OCaml.

We can define:

```
type 'a fincol = Fincol of 'a array
```

Note that

`[|1;2;3|]`      and      `[|1;3;2;2;1|]`

represent the same set!



## Finite collections in OCaml

We can test whether `x` belong to some set `Fincol a` with

# Finite collections in OCaml

We can test whether `x` belong to some set `Fincol a` with

```
let mem (x : 'a) (Fincol a : 'a fincol) =  
  let ans = ref false in  
  for i = 0 to Array.length a - 1 do  
    if x = a.(i) then ans := true  
  done;  
  !ans
```

# Finite collections in OCaml

We can test whether `x` belong to some set `Fincol a` with

```
let mem (x : 'a) (Fincol a : 'a fincol) =  
  let ans = ref false in  
  for i = 0 to Array.length a - 1 do  
    if x = a.(i) then ans := true  
  done;  
  !ans
```

Or even, using the standard library,

```
let mem (x : 'a) (Fincol a : 'a fincol) =  
  Array.exists (fun y -> x = y) a
```

Then, inclusion of sets can be computed with

Then, inclusion of sets can be computed with

```
let included (Fincol a : 'a fincol) (b : 'a fincol) =  
  Array.for_all (fun x -> mem x b) a
```

# Finite collections in OCaml

Then, inclusion of sets can be computed with

```
let included (Fincol a : 'a fincol) (b : 'a fincol) =  
  Array.for_all (fun x -> mem x b) a
```

and finally, equality of sets:

# Finite collections in OCaml

Then, inclusion of sets can be computed with

```
let included (Fincol a : 'a fincol) (b : 'a fincol) =  
  Array.for_all (fun x -> mem x b) a
```

and finally, equality of sets:

```
let eq (a : 'a fincol) (b : 'a fincol) =  
  included a b && included b a
```

# Finite collections in OCaml

Then, inclusion of sets can be computed with

```
let included (Fincol a : 'a fincol) (b : 'a fincol) =  
  Array.for_all (fun x -> mem x b) a
```

and finally, equality of sets:

```
let eq (a : 'a fincol) (b : 'a fincol) =  
  included a b && included b a
```

NB: this is the reasonable notion of equality, we never want to use `=`.



In particular, we can implement finite sets as finite collections of finite sets:

In particular, we can implement finite sets as finite collections of finite sets:

```
type finset = finset fincol
```

In particular, we can implement finite sets as finite collections of finite sets:

```
type finset = finset array
```

OCaml complains:

```
The type abbreviation finset is cyclic.
```

In particular, we can implement finite sets as finite collections of finite sets:

```
type finset = Finset of finset array
```

We can implement equality as previously:

```
let eq a b =  
    included a b && included b a
```

We can implement equality as previously:

```
let included (Finset a) b =  
  Array.for_all (fun x -> mem x b) a  
  
let eq a b =  
  included a b && included b a
```

We can implement equality as previously:

```
let mem x (Finset a) =  
    Array.exists (fun y -> eq x y) a  
  
let included (Finset a) b =  
    Array.for_all (fun x -> mem x b) a  
  
let eq a b =  
    included a b && included b a
```

We can implement equality as previously:

```
let rec mem x (Finset a) =  
    Array.exists (fun y -> eq x y) a  
  
and included (Finset a) b =  
    Array.for_all (fun x -> mem x b) a  
  
and eq a b =  
    included a b && included b a
```



These ideas can be immediately adapted to type theory and we define finite sets as

These ideas can be immediately adapted to type theory and we define finite sets as

```
data finset (A : Set) : Set where  
  Finset : {n : ℕ} → (Fin n → A) → finset A
```

## Encoding set theory in type theory

These ideas can be immediately adapted to type theory and we define finite sets as

```
data finset (A : Set) : Set where  
  Finset : {n : ℕ} → (Fin n → A) → finset A
```

But now there is no reason to limit ourselves to finite sets!

# Encoding set theory in type theory

These ideas can be immediately adapted to type theory and we define finite sets as

```
data finset (A : Set) : Set where  
  Finset : {I : Set} → (I → A) → finset A
```

But now there is no reason to limit ourselves to finite sets!

## Encoding set theory in type theory

Aczel defines the following encoding of set theory into type theory:

# Encoding set theory in type theory

Aczel defines the following encoding of set theory into type theory:

```
data U : Set1 where  
  set : {I : Set} → (I → U) → U
```

# Encoding set theory in type theory

Aczel defines the following encoding of set theory into type theory:

```
data U : Set1 where  
  set : {I : Set} → (I → U) → U
```

Instead of digging into set theory, we can try to encode Russell's paradox.

# Encoding set theory in type theory

Aczel defines the following encoding of set theory into type theory:

```
data U : Set1 where  
  set : {I : Set} → (I → U) → U
```

Instead of digging into set theory, we can try to encode Russell's paradox.

It turns out that if we suppose that the rule

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Type} : \text{Type}}$$

holds, the proof goes through!



# Encoding set theory in type theory

Aczel defines the following encoding of set theory into type theory:

```
data U : Set where  
  set : {I : Set} → (I → U) → U
```

Instead of digging into set theory, we can try to encode Russell's paradox.

It turns out that if we suppose that the rule

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Type} : \text{Type}}$$

holds, the proof goes through!

## Russell's paradox in type theory

Let's suppose `Type : Type` (i.e. `Set` of type `Set`):

```
{-# OPTIONS --type-in-type #-}
```

# Russell's paradox in type theory

Let's suppose `Type : Type` (i.e. `Set` of type `Set`):

```
{-# OPTIONS --type-in-type #-}
```

We define “sets” as

# Russell's paradox in type theory

Let's suppose `Type : Type` (i.e. `Set` of type `Set`):

```
{-# OPTIONS --type-in-type #-}
```

We define “sets” as

```
data U : Set where  
  set : {I : Set} → (I → U) → U
```

# Russell's paradox in type theory

Let's suppose `Type : Type` (i.e. `Set` of type `Set`):

```
{-# OPTIONS --type-in-type #-}
```

We define “sets” as

```
data U : Set where  
  set : {I : Set} → (I → U) → U
```

We define the membership relation as

# Russell's paradox in type theory

Let's suppose `Type : Type` (i.e. `Set` of type `Set`):

```
{-# OPTIONS --type-in-type #-}
```

We define “sets” as

```
data U : Set where
  set : {I : Set} → (I → U) → U
```

We define the membership relation as

```
_∈_ : (A B : U) → Set
A ∈ set {I} f =  $\sum I (\lambda i \rightarrow f\ i \equiv A)$ 
```

## Russell's paradox in type theory

A set is *regular* when it does not contains itself:

# Russell's paradox in type theory

A set is *regular* when it does not contains itself:

`regular : U → Set`

`regular A = ¬ (A ∈ A)`



# Russell's paradox in type theory

A set is *regular* when it does not contains itself:

`regular : U → Set`

`regular A = ¬ (A ∈ A)`

Russell's set `R` is the set of all regular sets

# Russell's paradox in type theory

A set is *regular* when it does not contains itself:

`regular : U → Set`

`regular A = ¬ (A ∈ A)`

Russell's set `R` is the set of all regular sets

`R : U`

`R = set {Σ U (λ A → regular A)} proj₁`

# Russell's paradox in type theory

$\mathbb{R}$  is not regular

# Russell's paradox in type theory

$R$  is not regular

$R\text{-nonreg} : \neg (\text{regular } R)$

$R\text{-nonreg } \text{reg} = \text{reg } ((R, \text{reg}), \text{refl})$

# Russell's paradox in type theory

$R$  is not regular

$R\text{-nonreg} : \neg (\text{regular } R)$

$R\text{-nonreg } \text{reg} = \text{reg } ((R, \text{reg}), \text{refl})$

$R$  is regular

# Russell's paradox in type theory

$R$  is not regular

$R\text{-nonreg} : \neg (\text{regular } R)$

$R\text{-nonreg } \text{reg} = \text{reg } ((R , \text{reg}) , \text{refl})$

$R$  is regular

$R\text{-reg} : \text{regular } R$

$R\text{-reg } ((\text{.}(\text{set } \text{proj}_1) , \text{reg}) , \text{refl}) = R\text{-nonreg } \text{reg}$

# Russell's paradox in type theory

$R$  is not regular

$R\text{-nonreg} : \neg (\text{regular } R)$

$R\text{-nonreg } \text{reg} = \text{reg } ((R , \text{reg}) , \text{refl})$

$R$  is regular

$R\text{-reg} : \text{regular } R$

$R\text{-reg } ((\text{.}(\text{set } \text{proj}_1) , \text{reg}) , \text{refl}) = R\text{-nonreg } \text{reg}$

The world falls apart

# Russell's paradox in type theory

$R$  is not regular

$R\text{-nonreg} : \neg (\text{regular } R)$

$R\text{-nonreg } \text{reg} = \text{reg } ((R , \text{reg}) , \text{refl})$

$R$  is regular

$R\text{-reg} : \text{regular } R$

$R\text{-reg } ((\text{set } \text{proj}_1) , \text{reg}) , \text{refl}) = R\text{-nonreg } \text{reg}$

The world falls apart

$\text{absurd} : \perp$

$\text{absurd} = R\text{-nonreg } R\text{-reg}$



## Russell's paradox in type theory

The morale of the story is the same as previously:  
the type of all types is “too big” to be a type.

## Russell's paradox in type theory

The morale of the story is the same as previously:  
the type of all types is “too big” to be a type.

However, we need to give it a type (everything does).

## Russell's paradox in type theory

The morale of the story is the same as previously:  
the type of all types is “too big” to be a type.

However, we need to give it a type (everything does).

Let's say that

Type : TYPE

where TYPE is the type of “big types”.

## Russell's paradox in type theory

The morale of the story is the same as previously:  
the type of all types is “too big” to be a type.

However, we need to give it a type (everything does).

Let's say that

Type : TYPE

where TYPE is the type of “big types”.

However, the next question is...

## Russell's paradox in type theory

The morale of the story is the same as previously:  
the type of all types is “too big” to be a type.

However, we need to give it a type (everything does).

Let's say that

Type : TYPE

where TYPE is the type of “big types”.

However, the next question is... what is the type of TYPE?

We thus introduce types

$\text{Type}_i$

indexed by  $i \in \mathbb{N}$ , and called **universes**, where

- $\text{Type}_0 = \text{Type}$  is the type of usual/small types,
- $\text{Type}_1$  is the type of “big types”,
- $\text{Type}_2$  is the type of “very big types”,

and so on.

We add the rule, for  $i \in \mathbb{N}$

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Type}_i : \text{Type}_{i+1}}$$

We add the rule, for  $i \in \mathbb{N}$

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Type}_i : \text{Type}_{i+1}}$$

as well as **cumulativity**, for  $i \in \mathbb{N}$ ,

$$\frac{\Gamma \vdash A : \text{Type}_i}{\Gamma \vdash A : \text{Type}_{i+1}}$$



We also adapt all the rules involving **Type**, e.g. for every  $i \in \mathbb{N}$ ,

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x : A \vdash B : \text{Type}_i}{\Gamma \vdash \Pi(x : A).B : \text{Type}_i} (\Pi_F)$$

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x : A \vdash B : \text{Type}_i}{\Gamma \vdash \Sigma(x : A).B : \text{Type}_i} (\Sigma_F)$$

and so on.

The rule for  $\Sigma$ -types is

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x : A \vdash B : \text{Type}_i}{\Gamma \vdash \Sigma(x : A).B : \text{Type}_i} (\Sigma_F)$$

For instance, we can define the type of all vectors of type  $A$  by

$\text{VEC} : (A : \text{Set}) \rightarrow \text{Set}$

$\text{VEC } A = \Sigma \mathbb{N} (\lambda n \rightarrow \text{Vec } A \ n)$

The rule for  $\Sigma$ -types is

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x : A \vdash B : \text{Type}_i}{\Gamma \vdash \Sigma(x : A).B : \text{Type}_i} (\Sigma_F)$$

However, if we try to define the type of all lists

`LIST : Set`

`LIST =  $\Sigma$  Set ( $\lambda A \rightarrow \text{List } A$ )`

The rule for  $\Sigma$ -types is

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x : A \vdash B : \text{Type}_i}{\Gamma \vdash \Sigma(x : A).B : \text{Type}_i} (\Sigma_F)$$

However, if we try to define the type of all lists

`LIST : Set`

`LIST =  $\Sigma$  Set ( $\lambda A \rightarrow \text{List } A$ )`

we get the error

`Set1 != Set`

when checking that the expression  `$\Sigma$  Set ( $\lambda A \rightarrow \text{List } A$ )` has type `Set`

The rule for  $\Sigma$ -types is

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x : A \vdash B : \text{Type}_i}{\Gamma \vdash \Sigma(x : A).B : \text{Type}_i} (\Sigma_F)$$

However, if we try to define the type of all lists

`LIST : Set1`

`LIST =  $\Sigma$  Set ( $\lambda A \rightarrow \text{List } A$ )`

Part IV

# Termination

All the programs in Agda should be **terminating**, otherwise it gets inconsistent:

All the programs in Agda should be **terminating**, otherwise it gets inconsistent:

```
{-# TERMINATING #-}  
anything : {A : Set} → ℕ → A  
anything n = anything (suc n)
```



All the programs in Agda should be **terminating**, otherwise it gets inconsistent:

```
{-# TERMINATING #-}  
anything : {A : Set} → A  
anything = anything
```

# Termination

All the programs in Agda should be **terminating**, otherwise it gets inconsistent:

```
{-# TERMINATING #-}  
anything : {A : Set} → A  
anything = anything
```

From which we deduce:

```
absurd : 0 ≡ 1  
absurd = anything
```

In this case, we should ask Agda to automatically check that the programs we write are terminating?



In this case, we should ask Agda to automatically check that the programs we write are terminating?

But in 1936, Turing showed that the **halting problem** is undecidable:

I: a program  $P$ ,

O: whether or not the program  $P$  will eventually stop.

# Termination

In order to always ensure termination, Agda (and most provers) use a syntactic criterion to ensure termination.

It only accepts programs which are “obviously” terminating, based on their form.

This means that some programs which are halting (and thus innocuous) are rejected.

But we will see that we can manage to get usual programs accepted.

# Termination

In Agda, the only possible source of non-termination is recursive functions:

```
anything : {A : Set} → ℕ → A  
anything n = anything (suc n)
```

# Termination

In Agda, the only possible source of non-termination is recursive functions:

```
anything : {A : Set} → ℕ → A  
anything n = anything (suc n)
```

If we try, we get the error

```
Termination checking failed for the following functions: anything  
Problematic calls: anything (suc n)
```

# Termination

In Agda, the only possible source of non-termination is recursive functions:

```
anything : {A : Set} → ℕ → A  
anything n = anything (suc n)
```

If we try, we get the error

```
Termination checking failed for the following functions: anything  
Problematic calls: anything (suc n)
```

It's something like a nightclub bouncer.



## Syntactic termination

In order to ensure termination, Agda imposes that recursive calls should be performed on strict subterms of the argument:

```
id : ℕ → ℕ
id zero      = zero
id (suc n) = suc (id n)
```

is accepted.

# Syntactic termination

In order to ensure termination, Agda imposes that recursive calls should be performed on strict subterms of the argument:

```
id :  $\mathbb{N} \rightarrow \mathbb{N}$   
id zero      = zero  
id (suc n) = id (suc n)
```

is rejected.

## Syntactic termination

In order to ensure termination, Agda imposes that recursive calls should be performed on strict subterms of the argument:

```
id : ℕ → ℕ
id zero          = zero
id (suc zero)    = suc zero
id (suc (suc n)) = suc (id (suc n))
```

is accepted.

## Syntactic termination

In order to ensure termination, Agda imposes that recursive calls should be performed on strict subterms of the argument:

```
div2 :  $\mathbb{N} \rightarrow \mathbb{N}$   
div2 zero          = zero  
div2 (suc zero)    = zero  
div2 (suc (suc n)) = suc (div2 n)
```

is accepted.

## Syntactic termination is over-restrictive

Suppose that we want to write the function `bits` which computes the number of bits necessary to write a natural number.

$$1_b = 1 \qquad 2_b = 10 \qquad 3_b = 11 \qquad 4_b = 100 \quad \dots$$

## Syntactic termination is over-restrictive

Suppose that we want to write the function `bits` which computes the number of bits necessary to write a natural number.

$$\begin{array}{ccccccccc} 1_b = 1 & & 2_b = 10 & & 3_b = 11 & & 4_b = 100 & & \dots \\ \text{bits}(1) = 1 & & \text{bits}(2) = 2 & & \text{bits}(3) = 2 & & \text{bits}(4) = 3 & & \dots \end{array}$$

## Syntactic termination is over-restrictive

Suppose that we want to write the function `bits` which computes the number of bits necessary to write a natural number.

$0_b =$	$1_b = 1$	$2_b = 10$	$3_b = 11$	$4_b = 100$	...
<code>bits(0) = 0</code>	<code>bits(1) = 1</code>	<code>bits(2) = 2</code>	<code>bits(3) = 2</code>	<code>bits(4) = 3</code>	...

## Syntactic termination is over-restrictive

Suppose that we want to write the function `bits` which computes the number of bits necessary to write a natural number.

$0_b =$	$1_b = 1$	$2_b = 10$	$3_b = 11$	$4_b = 100$	...
<code>bits(0) = 0</code>	<code>bits(1) = 1</code>	<code>bits(2) = 2</code>	<code>bits(3) = 2</code>	<code>bits(4) = 3</code>	...

The mathematical definition is

$$\text{bits}(n) = 1 + \lfloor \log_2(n) \rfloor$$



## Syntactic termination is over-restrictive

Suppose that we want to write the function `bits` which computes the number of bits necessary to write a natural number.

$0_b =$	$1_b = 1$	$2_b = 10$	$3_b = 11$	$4_b = 100$	...
<code>bits(0) = 0</code>	<code>bits(1) = 1</code>	<code>bits(2) = 2</code>	<code>bits(3) = 2</code>	<code>bits(4) = 3</code>	...

The mathematical definition is

$$\text{bits}(n) = 1 + \lfloor \log_2(n) \rfloor$$

In OCaml, we would write

## Syntactic termination is over-restrictive

Suppose that we want to write the function `bits` which computes the number of bits necessary to write a natural number.

$0_b =$	$1_b = 1$	$2_b = 10$	$3_b = 11$	$4_b = 100$	...
<code>bits(0) = 0</code>	<code>bits(1) = 1</code>	<code>bits(2) = 2</code>	<code>bits(3) = 2</code>	<code>bits(4) = 3</code>	...

The mathematical definition is

$$\text{bits}(n) = 1 + \lfloor \log_2(n) \rfloor$$

In OCaml, we would write

```
let rec bits n =  
  if n = 0 then 0 else 1 + bits (n / 2)
```

## Syntactic termination is over-restrictive

If we translate this into Agda

## Syntactic termination is over-restrictive

If we translate this into Agda

```
bits : ℕ → ℕ
bits zero    = zero
bits (suc n) = suc (bits (div2 (suc n)))
```

## Syntactic termination is over-restrictive

If we translate this into Agda

```
bits : ℕ → ℕ
bits zero    = zero
bits (suc n) = suc (bits (div2 (suc n)))
```

the definition gets rejected.

## Syntactic termination is over-restrictive

If we translate this into Agda

```
bits : ℕ → ℕ
bits zero      = zero
bits (suc n) = suc (bits (div2 (suc n)))
```

the definition gets rejected.

Agda is not smart enough to find out that `div2 (suc n)` will return something which is *syntactically* smaller than `suc n`.

## Syntactic termination is over-restrictive

If we translate this into Agda

```
bits : ℕ → ℕ
bits zero    = zero
bits (suc n) = suc (bits (div2 (suc n)))
```

the definition gets rejected.

Agda is not smart enough to find out that `div2 (suc n)` will return something which is *syntactically* smaller than `suc n`.

Fortunately, there are ways to get over this.

## The fuel technique

The **fuel** technique consists in adding an extra argument (typically a natural number) which will be syntactically decreasing.

This is some kind of fuel which gets consumed at each recursive call: the function will terminate because it will eventually run out of fuel.

Note that in order to make a recursive call, we have to ensure that there is some fuel left, i.e. there is a strict subterm of the fuel argument. We have to add a second new argument which maintains an invariant ensuring this.



## The fuel technique

```
bits : (n : ℕ) → (fuel : ℕ) → ℕ
bits zero    f      = zero
bits (suc n) zero    = ?
bits (suc n) (suc f) = suc (bits (div2 (suc n)) f)
```

## The fuel technique

```
bits : (n : ℕ) → (fuel : ℕ) → (n ≤ fuel) → ℕ
bits zero    f      p = zero
bits (suc n) zero    ()
bits (suc n) (suc f) p = suc (bits (div2 (suc n)) f ?)
```

## The fuel technique

```
bits : (n : ℕ) → (fuel : ℕ) → (n ≤ fuel) → ℕ
bits zero    f      p = zero
bits (suc n) zero    ()
bits (suc n) (suc f) p = suc (bits (div2 (suc n)) f ?)
```

The last goal requires proving, under  $\text{suc } n \leq \text{suc } f$ , that  $\text{div2 } (\text{suc } n) \leq f$ .

## The fuel technique

```
bits : (n : ℕ) → (fuel : ℕ) → (n ≤ fuel) → ℕ
bits zero    f      p = zero
bits (suc n) zero    ()
bits (suc n) (suc f) p = suc (bits (div2 (suc n)) f ?)
```

The last goal requires proving, under  $\text{suc } n \leq \text{suc } f$ , that  $\text{div2 } (\text{suc } n) \leq f$ .  
Namely,

$$n + 1 \leq f + 1$$

## The fuel technique

```
bits : (n : ℕ) → (fuel : ℕ) → (n ≤ fuel) → ℕ
bits zero    f      p = zero
bits (suc n) zero    ()
bits (suc n) (suc f) p = suc (bits (div2 (suc n)) f ?)
```

The last goal requires proving, under  $\text{suc } n \leq \text{suc } f$ , that  $\text{div2 } (\text{suc } n) \leq f$ .  
Namely,

$$(n + 1)/2 \leq (f + 1)/2$$

## The fuel technique

```
bits : (n : ℕ) → (fuel : ℕ) → (n ≤ fuel) → ℕ
bits zero    f      p = zero
bits (suc n) zero    ()
bits (suc n) (suc f) p = suc (bits (div2 (suc n)) f ?)
```

The last goal requires proving, under  $\text{suc } n \leq \text{suc } f$ , that  $\text{div2 } (\text{suc } n) \leq f$ .  
Namely,

$$(n + 1)/2 \leq (f + 1)/2 \leq f$$

## The fuel technique

```
bits : (n : ℕ) → (fuel : ℕ) → (n ≤ fuel) → ℕ
bits zero    f      p = zero
bits (suc n) zero    ()
bits (suc n) (suc f) p = suc (bits (div2 (suc n)) f ?)
```

The last goal requires proving, under  $\text{suc } n \leq \text{suc } f$ , that  $\text{div2 } (\text{suc } n) \leq f$ .  
Namely,

$$(n + 1)/2 \leq (f + 1)/2 \leq f$$

This thus follows from two easy lemmas:

- division is increasing:  $\{m \ n : \mathbb{N}\} \rightarrow m \leq n \rightarrow \text{div2 } m \leq \text{div2 } n$

## The fuel technique

```
bits : (n : ℕ) → (fuel : ℕ) → (n ≤ fuel) → ℕ
bits zero    f      p = zero
bits (suc n) zero    ()
bits (suc n) (suc f) p = suc (bits (div2 (suc n)) f ?)
```

The last goal requires proving, under  $\text{suc } n \leq \text{suc } f$ , that  $\text{div2 } (\text{suc } n) \leq f$ .  
Namely,

$$(n + 1)/2 \leq (f + 1)/2 \leq f$$

This thus follows from two easy lemmas:

- division is increasing:  $\{m \ n : \mathbb{N}\} \rightarrow m \leq n \rightarrow \text{div2 } m \leq \text{div2 } n$
- a technical result:  $(n : \mathbb{N}) \rightarrow \text{div2 } (\text{suc } n) \leq n$



## Proof of the technical lemma

We can show (left as exercise):

$$\leq\text{-div2} : \{m\ n : \mathbb{N}\} \rightarrow m \leq n \rightarrow \text{div2 } m \leq \text{div2 } n$$

and

$$\leq\text{-suc} : \{n : \mathbb{N}\} \rightarrow n \leq \text{suc } n$$

## Proof of the technical lemma

We can show (left as exercise):

$$\leq\text{-div2} : \{m\ n : \mathbb{N}\} \rightarrow m \leq n \rightarrow \text{div2 } m \leq \text{div2 } n$$

and

$$\leq\text{-suc} : \{n : \mathbb{N}\} \rightarrow n \leq \text{suc } n$$

Thus,

$$\text{lem} : (n : \mathbb{N}) \rightarrow \text{div2 } (\text{suc } n) \leq n$$

$$\text{lem zero} = \text{z}\leq\text{n}$$

$$\text{lem } (\text{suc } n) = \leq\text{-trans } (\leq\text{-div2 } \leq\text{-suc}) (\text{s}\leq\text{s } (\text{lem } n))$$

In maths, the second case is

$$(n+2)/2 \leq (n+3)/2 = (n+1)/2 + 1 \leq n+1$$

# Well-founded induction

A more general technique is **well-founded induction**.

This is an adaptation of a classical mathematical technique.

# Recurrence

## Theorem (Recurrence principle)

Given a property  $P(n)$ , if

- $P(0)$  holds, and
- for every  $n \in \mathbb{N}$ ,  $P(n)$  implies  $P(n + 1)$ ,

then  $P(n)$  holds for every  $n \in \mathbb{N}$ .

# Recurrence

## Theorem (Recurrence principle)

Given a property  $P(n)$ , if

- $P(0)$  holds, and
- for every  $n \in \mathbb{N}$ ,  $P(n)$  implies  $P(n + 1)$ ,

then  $P(n)$  holds for every  $n \in \mathbb{N}$ .

## Theorem (Strong recurrence principle)

Given a property  $P(n)$ , if

- for every  $n \in \mathbb{N}$ , if  $P(m)$  holds for every  $m < n$ , then  $P(n)$ ,

then  $P(n)$  holds for every  $n \in \mathbb{N}$ .

# Recurrence

## Theorem (Recurrence principle)

Given a property  $P(n)$ , if

- $P(0)$  holds, and
- for every  $n \in \mathbb{N}$ ,  $P(n)$  implies  $P(n + 1)$ ,

then  $P(n)$  holds for every  $n \in \mathbb{N}$ .

## Theorem (Strong recurrence principle)

Given a property  $P(n)$ , if

- for every  $n \in \mathbb{N}$ , if  $P(m)$  holds for every  $m < n$ , then  $P(n)$ ,

then  $P(n)$  holds for every  $n \in \mathbb{N}$ .

Note that we have to show  $P(0)$  as a particular case.

## Theorem (Induction on proofs)

Suppose given a predicate  $P(\pi)$  on proofs  $\pi$ . Suppose that for every rule

$$\pi = \frac{\frac{\pi_1}{\Gamma_1 \vdash A_1} \quad \cdots \quad \frac{\pi_n}{\Gamma_n \vdash A_n}}{\Gamma \vdash A}$$

if  $P(\pi_i)$  holds for every  $i$  then  $P(\pi)$  also holds. Then  $P(\pi)$  holds for every proof  $\pi$ .

Let's give a general theory of induction.



## Well-founded relations

On a set  $A$ , a relation  $R \subseteq A \times A$  is **well-founded** if there is no infinite sequence of elements  $x_i$  of  $A$  such that  $(x_{i+1}, x_i) \in R$ .

We forbid  $\dots R x_3 R x_2 R x_1 R x_0$

## Well-founded relations

On a set  $A$ , a relation  $R \subseteq A \times A$  is **well-founded** if there is no infinite sequence of elements  $x_i$  of  $A$  such that  $(x_{i+1}, x_i) \in R$ .

We forbid  $\dots R x_3 R x_2 R x_1 R x_0$

The relation  $<$  on  $\mathbb{N}$  is

## Well-founded relations

On a set  $A$ , a relation  $R \subseteq A \times A$  is **well-founded** if there is no infinite sequence of elements  $x_i$  of  $A$  such that  $(x_{i+1}, x_i) \in R$ .

We forbid  $\dots R x_3 R x_2 R x_1 R x_0$

The relation  $<$  on  $\mathbb{N}$  is well-founded: there is no infinite sequence

$$\dots < n_3 < n_2 < n_1 < n_0$$

## Well-founded relations

On a set  $A$ , a relation  $R \subseteq A \times A$  is **well-founded** if there is no infinite sequence of elements  $x_i$  of  $A$  such that  $(x_{i+1}, x_i) \in R$ .

We forbid  $\dots R x_3 R x_2 R x_1 R x_0$

The relation  $<$  on  $\mathbb{N}$  is well-founded: there is no infinite sequence

$$\dots < n_3 < n_2 < n_1 < n_0$$

The relation  $<$  on  $\mathbb{Q}$  is

## Well-founded relations

On a set  $A$ , a relation  $R \subseteq A \times A$  is **well-founded** if there is no infinite sequence of elements  $x_i$  of  $A$  such that  $(x_{i+1}, x_i) \in R$ .

We forbid  $\dots R x_3 R x_2 R x_1 R x_0$

The relation  $<$  on  $\mathbb{N}$  is well-founded: there is no infinite sequence

$$\dots < n_3 < n_2 < n_1 < n_0$$

The relation  $<$  on  $\mathbb{Q}$  is not well-founded:

$$\dots < \frac{1}{4} < \frac{1}{3} < \frac{1}{2} < 1$$

## Well-founded relations

On a set  $A$ , a relation  $R \subseteq A \times A$  is **well-founded** if there is no infinite sequence of elements  $x_i$  of  $A$  such that  $(x_{i+1}, x_i) \in R$ .

We forbid  $\dots R x_3 R x_2 R x_1 R x_0$

The relation  $<$  on  $\mathbb{N}$  is well-founded: there is no infinite sequence

$$\dots < n_3 < n_2 < n_1 < n_0$$

The relation  $<$  on  $\mathbb{Q}$  is not well-founded:

$$\dots < \frac{1}{4} < \frac{1}{3} < \frac{1}{2} < 1$$

The relation  $S = \{(n, n+1) \mid n \in \mathbb{N}\}$  on  $\mathbb{N}$  is

## Well-founded relations

On a set  $A$ , a relation  $R \subseteq A \times A$  is **well-founded** if there is no infinite sequence of elements  $x_i$  of  $A$  such that  $(x_{i+1}, x_i) \in R$ .

We forbid  $\dots R x_3 R x_2 R x_1 R x_0$

The relation  $<$  on  $\mathbb{N}$  is well-founded: there is no infinite sequence

$$\dots < n_3 < n_2 < n_1 < n_0$$

The relation  $<$  on  $\mathbb{Q}$  is not well-founded:

$$\dots < \frac{1}{4} < \frac{1}{3} < \frac{1}{2} < 1$$

The relation  $S = \{(n, n+1) \mid n \in \mathbb{N}\}$  on  $\mathbb{N}$  is well-founded:

$$\dots S (n-3) S (n-2) S (n-1) S n$$

## Well-founded relations

On a set  $A$ , a relation  $R \subseteq A \times A$  is **well-founded** if there is no infinite sequence of elements  $x_i$  of  $A$  such that  $(x_{i+1}, x_i) \in R$ .

We forbid  $\dots R x_3 R x_2 R x_1 R x_0$

The relation  $<$  on  $\mathbb{N}$  is well-founded: there is no infinite sequence

$$\dots < n_3 < n_2 < n_1 < n_0$$

The relation  $<$  on  $\mathbb{Q}$  is not well-founded:

$$\dots < \frac{1}{4} < \frac{1}{3} < \frac{1}{2} < 1$$

The relation  $S = \{(n, n+1) \mid n \in \mathbb{N}\}$  on  $\mathbb{N}$  is well-founded:

$$\dots S (n-3) S (n-2) S (n-1) S n$$

The relation  $\in$  on sets is



## Well-founded relations

On a set  $A$ , a relation  $R \subseteq A \times A$  is **well-founded** if there is no infinite sequence of elements  $x_i$  of  $A$  such that  $(x_{i+1}, x_i) \in R$ .

We forbid  $\dots R x_3 R x_2 R x_1 R x_0$

The relation  $<$  on  $\mathbb{N}$  is well-founded: there is no infinite sequence

$$\dots < n_3 < n_2 < n_1 < n_0$$

The relation  $<$  on  $\mathbb{Q}$  is not well-founded:

$$\dots < \frac{1}{4} < \frac{1}{3} < \frac{1}{2} < 1$$

The relation  $S = \{(n, n+1) \mid n \in \mathbb{N}\}$  on  $\mathbb{N}$  is well-founded:

$$\dots S (n-3) S (n-2) S (n-1) S n$$

The relation  $\in$  on sets is well-founded: this is the axiom of foundation.

## Immediately below

Given a relation  $R$  on  $A$ , and  $x \in A$  we write

$$\downarrow x = \{y \in A \mid y R x\}$$

for the set of elements **immediately below**  $x$ .

## Immediately below

Given a relation  $R$  on  $A$ , and  $x \in A$  we write

$$\downarrow x = \{y \in A \mid y R x\}$$

for the set of elements **immediately below**  $x$ .

For  $<$  on  $\mathbb{N}$ ,

$$\downarrow n =$$

## Immediately below

Given a relation  $R$  on  $A$ , and  $x \in A$  we write

$$\downarrow x = \{y \in A \mid y R x\}$$

for the set of elements **immediately below**  $x$ .

For  $<$  on  $\mathbb{N}$ ,

$$\downarrow n = \{m \in \mathbb{N} \mid m < n\}$$

## Immediately below

Given a relation  $R$  on  $A$ , and  $x \in A$  we write

$$\downarrow x = \{y \in A \mid y R x\}$$

for the set of elements **immediately below**  $x$ .

For  $<$  on  $\mathbb{N}$ ,

$$\downarrow n = \{m \in \mathbb{N} \mid m < n\}$$

For  $S$  on  $\mathbb{N}$

$$\downarrow n =$$

## Immediately below

Given a relation  $R$  on  $A$ , and  $x \in A$  we write

$$\downarrow x = \{y \in A \mid y R x\}$$

for the set of elements **immediately below**  $x$ .

For  $<$  on  $\mathbb{N}$ ,

$$\downarrow n = \{m \in \mathbb{N} \mid m < n\}$$

For  $S$  on  $\mathbb{N}$

$$\downarrow n = \begin{cases} \emptyset & \text{if } n = 0 \\ \{n - 1\} & \text{otherwise} \end{cases}$$

# Well-founded relation

Alternatively, some people say that a relation  $R$  on  $A$  is **well-founded** when every non-empty subset  $X \subseteq A$  admits a minimal element:

$$\forall X \subseteq A. (X \neq \emptyset \Rightarrow \exists x \in X. \downarrow x \cap X = \emptyset)$$

The two definitions agree if we assume the axiom of (dependent) choice.

## Well-founded induction

### Theorem (Well-founded induction)

Suppose given a well-founded relation  $R$  on  $A$  and a property  $P(x)$  on  $A$ .

If  $P$  satisfies, for every  $x \in A$ ,

if  $P(y)$  for every  $y \in \downarrow x$  then  $P(x)$

then  $P(x)$  holds for every  $x \in A$ .



# Well-founded induction

## Theorem (Well-founded induction)

Suppose given a well-founded relation  $R$  on  $A$  and a property  $P(x)$  on  $A$ .

If  $P$  satisfies, for every  $x \in A$ ,

if  $P(y)$  for every  $y \in \downarrow x$  then  $P(x)$

then  $P(x)$  holds for every  $x \in A$ .

Note that

- for  $S$  on  $\mathbb{N}$ , we recover the recurrence principle,
- for  $<$  on  $\mathbb{N}$ , we recover the strong recurrence principle.

# Well-founded induction

## Theorem (Well-founded induction)

Suppose given a well-founded relation  $R$  on  $A$  and a property  $P(x)$  on  $A$ .

If  $P$  satisfies, for every  $x \in A$ ,

if  $P(y)$  for every  $y \in \downarrow x$  then  $P(x)$

then  $P(x)$  holds for every  $x \in A$ .

## Proof.

By contradiction, suppose that  $P(x_0)$  does not hold for some  $x_0 \in A$  then

# Well-founded induction

## Theorem (Well-founded induction)

Suppose given a well-founded relation  $R$  on  $A$  and a property  $P(x)$  on  $A$ .

If  $P$  satisfies, for every  $x \in A$ ,

if  $P(y)$  for every  $y \in \downarrow x$  then  $P(x)$

then  $P(x)$  holds for every  $x \in A$ .

## Proof.

By contradiction, suppose that  $P(x_0)$  does not hold for some  $x_0 \in A$  then

- there exists  $x_1 \in A$  with  $x_1 R x_0$  such that  $P(x_1)$  does not hold,

# Well-founded induction

## Theorem (Well-founded induction)

Suppose given a well-founded relation  $R$  on  $A$  and a property  $P(x)$  on  $A$ .

If  $P$  satisfies, for every  $x \in A$ ,

if  $P(y)$  for every  $y \in \downarrow x$  then  $P(x)$

then  $P(x)$  holds for every  $x \in A$ .

## Proof.

By contradiction, suppose that  $P(x_0)$  does not hold for some  $x_0 \in A$  then

- there exists  $x_1 \in A$  with  $x_1 R x_0$  such that  $P(x_1)$  does not hold,
- there exists  $x_2 \in A$  with  $x_2 R x_1$  such that  $P(x_2)$  does not hold,

# Well-founded induction

## Theorem (Well-founded induction)

Suppose given a well-founded relation  $R$  on  $A$  and a property  $P(x)$  on  $A$ .

If  $P$  satisfies, for every  $x \in A$ ,

if  $P(y)$  for every  $y \in \downarrow x$  then  $P(x)$

then  $P(x)$  holds for every  $x \in A$ .

## Proof.

By contradiction, suppose that  $P(x_0)$  does not hold for some  $x_0 \in A$  then

- there exists  $x_1 \in A$  with  $x_1 R x_0$  such that  $P(x_1)$  does not hold,
- there exists  $x_2 \in A$  with  $x_2 R x_1$  such that  $P(x_2)$  does not hold,
- ...

# Well-founded induction

## Theorem (Well-founded induction)

Suppose given a well-founded relation  $R$  on  $A$  and a property  $P(x)$  on  $A$ .

If  $P$  satisfies, for every  $x \in A$ ,

if  $P(y)$  for every  $y \in \downarrow x$  then  $P(x)$

then  $P(x)$  holds for every  $x \in A$ .

## Proof.

By contradiction, suppose that  $P(x_0)$  does not hold for some  $x_0 \in A$  then

- there exists  $x_1 \in A$  with  $x_1 R x_0$  such that  $P(x_1)$  does not hold,
- there exists  $x_2 \in A$  with  $x_2 R x_1$  such that  $P(x_2)$  does not hold,
- ...

We have constructed an infinite decreasing sequence, contradiction.



A variant can also be used to construct functions.

### Theorem (Well-founded recursion)

Suppose given sets  $A$  and  $B$ , a well-founded relation  $R$  on  $A$  and function  $r$  which to every  $x \in A$  and function  $g : \downarrow x \rightarrow B$  associates an element of  $B$ . Then there is a unique function  $f : A \rightarrow B$  such that, for every  $x \in A$ ,

$$f(x) = r(x, f|_{\downarrow x})$$

## Well-founded recursion

For instance, we can define binary trees by

```
data BTree (A : Set) : Set where  
  Leaf : BTree A  
  Node : A → BTree A → BTree A → BTree A
```



## Well-founded recursion

For instance, we can define binary trees by

```
data BTree (A : Set) : Set where
  Leaf : BTree A
  Node : A → BTree A → BTree A → BTree A
```

We consider the relation such that, for every tree  $t = \text{Node}(a, t_1, t_2)$ , we have  $t_1 < t$  and  $t_2 < t$ , which is well-founded (why?).

## Well-founded recursion

For instance, we can define binary trees by

```
data BTree (A : Set) : Set where
  Leaf : BTree A
  Node : A → BTree A → BTree A → BTree A
```

We consider the relation such that, for every tree  $t = \text{Node}(a, t_1, t_2)$ , we have  $t_1 < t$  and  $t_2 < t$ , which is well-founded (why?).

We can define the height function by well-founded recursion:

## Well-founded recursion

For instance, we can define binary trees by

```
data BTree (A : Set) : Set where
  Leaf : BTree A
  Node : A → BTree A → BTree A → BTree A
```

We consider the relation such that, for every tree  $t = \text{Node}(a, t_1, t_2)$ , we have  $t_1 < t$  and  $t_2 < t$ , which is well-founded (why?).

We can define the height function by well-founded recursion:

```
height : {A : Set} → BTree A → ℕ
height Leaf = 0
height (Node x t t') = suc (max (height t) (height t'))
```

## Well-founded recursion

More generally, given a fixed signature, we can define the subterm order by

$$t < u$$

when  $t$  is a strict subterm of  $u$ .

This relation is always well-founded.

## Well-founded recursion

More generally, given a fixed signature, we can define the subterm order by

$$t < u$$

when  $t$  is a strict subterm of  $u$ .

This relation is always well-founded.

Agda ensures that, when defining  $f(u)$ , the recursive calls are performed on  $f(t)$  with  $t < u$ . This enforces that

- all functions are terminating,
- all functions are well-defined (by well-founded recursion).

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

## Lemma

*The set  $\text{Acc}(A)$  is well-defined.*



# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

## Lemma

*The set  $\text{Acc}(A)$  is well-defined.*

## Proof.

A subset  $X$  of  $A$  is **closed** when  $\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in X) \Rightarrow x \in X$ .

We define

$$\text{Acc}(A) = \bigcap_{X \text{ closed}} X$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

With  $R$  being the relation such that  $x R (x + 1)$  for every  $x$ ,

$$\text{Acc}(\mathbb{N}) = \{ \quad \quad \}$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

With  $R$  being the relation such that  $x R (x + 1)$  for every  $x$ ,

$$\text{Acc}(\mathbb{N}) = \{0 \quad \quad \quad \}$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

With  $R$  being the relation such that  $x R (x + 1)$  for every  $x$ ,

$$\text{Acc}(\mathbb{N}) = \{0, 1 \quad \}$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

With  $R$  being the relation such that  $x R (x + 1)$  for every  $x$ ,

$$\text{Acc}(\mathbb{N}) = \{0, 1, 2 \quad \}$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

With  $R$  being the relation such that  $x R (x + 1)$  for every  $x$ ,

$$\text{Acc}(\mathbb{N}) = \{0, 1, 2, \dots\}$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

With  $R$  being the relation such that  $x R (x + 1)$  for every  $x$ ,

$$\text{Acc}(\mathbb{N}) = \mathbb{N}$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

With  $R$  being the relation such that  $x R (x + 1)$  for every  $x$ ,

$$\text{Acc}(\mathbb{N}) = \mathbb{N} \qquad \text{Acc}(\mathbb{Z}) =$$



# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

With  $R$  being the relation such that  $x R (x + 1)$  for every  $x$ ,

$$\text{Acc}(\mathbb{N}) = \mathbb{N} \qquad \text{Acc}(\mathbb{Z}) = \emptyset$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

With  $R$  being the relation such that  $x R (x + 1)$  for every  $x$ ,

$$\text{Acc}(\mathbb{N}) = \mathbb{N} \qquad \text{Acc}(\mathbb{Z}) = \emptyset \qquad \text{Acc}(\mathbb{R}) =$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

With  $R$  being the relation such that  $x R (x + 1)$  for every  $x$ ,

$$\text{Acc}(\mathbb{N}) = \mathbb{N}$$

$$\text{Acc}(\mathbb{Z}) = \emptyset$$

$$\text{Acc}(\mathbb{R}) = \emptyset$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

With  $R$  being the relation such that  $x R (x + 1)$  for every  $x$ ,

$$\text{Acc}(\mathbb{N}) = \mathbb{N} \qquad \text{Acc}(\mathbb{Z}) = \emptyset \qquad \text{Acc}(\mathbb{R}) = \emptyset \qquad \text{Acc}(\mathbb{R} \setminus \{-1\}) =$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

With  $R$  being the relation such that  $x R (x + 1)$  for every  $x$ ,

$$\text{Acc}(\mathbb{N}) = \mathbb{N}$$

$$\text{Acc}(\mathbb{Z}) = \emptyset$$

$$\text{Acc}(\mathbb{R}) = \emptyset$$

$$\text{Acc}(\mathbb{R} \setminus \{-1\}) = \mathbb{N}$$

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

## Lemma

*A relation  $R$  on  $A$  is well-founded iff*

# Accessibility

Given a set  $A$  equipped with a relation  $R$  (not necessarily well-founded), the set of **accessible** elements  $\text{Acc}(A)$  is the smallest set on which well-founded induction would work.

Formally, the set  $\text{Acc}(A)$  is the smallest subset of  $A$  such that

$$\forall x \in A. (\forall y \in A. y R x \Rightarrow y \in \text{Acc}(A)) \Rightarrow x \in \text{Acc}(A)$$

## Lemma

*A relation  $R$  on  $A$  is well-founded iff  $A = \text{Acc}(A)$ .*

## Well-founded relations

In Agda, we can define the type of relations on  $A$  by



# Well-founded relations

In Agda, we can define the type of relations on  $A$  by

```
Rel : Set → Set1
```

```
Rel A = A → A → Set
```

# Well-founded relations

In Agda, we can define the type of relations on  $A$  by

```
Rel : Set → Set1
```

```
Rel A = A → A → Set
```

We can then define the predicate of being accessible for an element by

# Well-founded relations

In Agda, we can define the type of relations on  $A$  by

```
Rel : Set → Set1
```

```
Rel A = A → A → Set
```

We can then define the predicate of being accessible for an element by

```
data Acc {A : Set} (_<_ : Rel A) (x : A) : Set where
```

```
  acc : ((y : A) → y < x → Acc _<_ y) → Acc _<_ x
```

# Well-founded relations

In Agda, we can define the type of relations on  $A$  by

```
Rel : Set → Set1
```

```
Rel A = A → A → Set
```

We can then define the predicate of being accessible for an element by

```
data Acc {A : Set} (_<_ : Rel A) (x : A) : Set where
```

```
  acc : ((y : A) → y < x → Acc _<_ y) → Acc _<_ x
```

We define a relation to be well-founded by the predicate

# Well-founded relations

In Agda, we can define the type of relations on  $A$  by

```
Rel : Set → Set1
```

```
Rel A = A → A → Set
```

We can then define the predicate of being accessible for an element by

```
data Acc {A : Set} (_<_ : Rel A) (x : A) : Set where
```

```
  acc : ((y : A) → y < x → Acc _<_ y) → Acc _<_ x
```

We define a relation to be well-founded by the predicate

```
WellFounded : {A : Set} → (_<_ : Rel A) → Set
```

```
WellFounded {A} _<_ = (x : A) → Acc _<_ x
```

## $\mathbb{N}$ is well-founded

The usual definition of the order turns out to make inductions difficult (see in TD). The following definition of the order turns out to be more adapted

```
data _<_ :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Set}$  where  
  <-base : {n :  $\mathbb{N}$ }  $\rightarrow$  n < suc n  
  <-step : {m n :  $\mathbb{N}$ }  $\rightarrow$  m < n  $\rightarrow$  m < suc n
```

## $\mathbb{N}$ is well-founded

The usual definition of the order turns out to make inductions difficult (see in TD). The following definition of the order turns out to be more adapted

```
data _<_ :  $\mathbb{N}$   $\rightarrow$   $\mathbb{N}$   $\rightarrow$  Set where  
  <-base : {n :  $\mathbb{N}$ }  $\rightarrow$  n < suc n  
  <-step : {m n :  $\mathbb{N}$ }  $\rightarrow$  m < n  $\rightarrow$  m < suc n
```

We can then show that  $\mathbb{N}$  is well-founded by

## $\mathbb{N}$ is well-founded

The usual definition of the order turns out to make inductions difficult (see in TD). The following definition of the order turns out to be more adapted

```
data _<_ :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Set}$  where
  <-base : {n :  $\mathbb{N}$ }  $\rightarrow$  n < suc n
  <-step : {m n :  $\mathbb{N}$ }  $\rightarrow$  m < n  $\rightarrow$  m < suc n
```

We can then show that  $\mathbb{N}$  is well-founded by

```
<-wellFounded : WellFounded _<_
<-wellFounded x = acc (aux x)
  where
    aux : (x y :  $\mathbb{N}$ )  $\rightarrow$  y < x  $\rightarrow$  Acc _<_ y
    aux .(suc y) y <-base = <-wellFounded y
    aux (suc x) y (<-step y<x) = aux x y y<x
```



## Well-founded induction

In order to define a recursive function  $f\ n$  on a well-founded relation (e.g.  $\mathbb{N}$ ), we can define an auxiliary function  $g$  which is  $f$  with a new argument: the proof that  $n$  is accessible.

We performing recursive calls, we only have to prove that the arguments are strictly smaller.

The trick is thus essentially the same as for fuel, excepting that it is closer to the usual mathematical practice.

We can finally define  $f\ n = g\ n\ (\text{<-wellFounded } n)$ .

Assuming the easy lemma

$$\text{div2-} < : (n : \mathbb{N}) \rightarrow \text{div2} (\text{suc } n) < \text{suc } n$$

## Well-founded bits

Assuming the easy lemma

`div2-< : (n : ℕ) → div2 (suc n) < suc n`

the function bits can be defined by well-founded induction by

## Well-founded bits

Assuming the easy lemma

```
div2-< : (n : ℕ) → div2 (suc n) < suc n
```

the function bits can be defined by well-founded induction by

```
wfbits : (n : ℕ) → Acc _<_ n → ℕ  
wfbits zero      _      = zero  
wfbits (suc n) (acc rs) =  
    suc (wfbits (div2 (suc n)) (rs (div2 (suc n)) (div2-< n)))
```

## Well-founded bits

Assuming the easy lemma

```
div2-< : (n : ℕ) → div2 (suc n) < suc n
```

the function bits can be defined by well-founded induction by

```
wfbits : (n : ℕ) → Acc _<_ n → ℕ
```

```
wfbits zero _ = zero
```

```
wfbits (suc n) (acc rs) =  
  suc (wfbits (div2 (suc n)) (rs (div2 (suc n)) (div2-< n)))
```

```
bits : ℕ → ℕ
```

```
bits n = wfbits n (<-wellFounded n)
```

In practice, we don't want the average user to understand accessibility.

The standard library (`Data.Nat.Induction`) defines the following equivalent principle:

```
<-rec : (P : ℕ → Set) →  
        ((n : ℕ) → ((m : ℕ) → m < n → P m) → P n) →  
        (n : ℕ) → P n  
<-rec P r n = lem n (<-wellFounded n)  
  where  
    lem : (n : ℕ) → Acc _<_ n → P n  
    lem n (acc a) = r n (λ m m<n → lem m (a m m<n))
```

We can implement bits by implementing the auxiliary function

```
bits-rec : (n : ℕ) → ((m : ℕ) → m < n → ℕ) → ℕ
bits-rec zero    r = zero
bits-rec (suc n) r = suc (r (div2 n) (div2-<' n))
```

We can implement bits by implementing the auxiliary function

```
bits-rec : (n : ℕ) → ((m : ℕ) → m < n → ℕ) → ℕ
bits-rec zero    r = zero
bits-rec (suc n) r = suc (r (div2 n) (div2-<' n))
```

and then finally define

```
bits : ℕ → ℕ
bits = <-rec (λ n → ℕ) bits-rec
```



Part V

## Inductive types

A problem with inductive types (`data ...`) is that we cannot manipulate them:

- we cannot make a function which takes an inductive type and returns a new inductive type,
- we cannot reason on all the types defined inductively
- etc.

A problem with inductive types (`data ...`) is that we cannot manipulate them:

- we cannot make a function which takes an inductive type and returns a new inductive type,
- we cannot reason on all the types defined inductively
- etc.

Or can we?

A problem with inductive types (`data ...`) is that we cannot manipulate them:

- we cannot make a function which takes an inductive type and returns a new inductive type,
- we cannot reason on all the types defined inductively
- etc.

Or can we?

(at least for a large class of inductive types)

## Finite sets

In Agda, for a fixed  $n$ , it is easy to describe a type with exactly  $n$  elements.

## Finite sets

In Agda, for a fixed  $n$ , it is easy to describe a type with exactly  $n$  elements.

We have a type with 1 element:

```
data T : Set where  
  tt : T
```

# Finite sets

In Agda, for a fixed  $n$ , it is easy to describe a type with exactly  $n$  elements.

We have a type with 1 element:

```
data T : Set where
  tt : T
```

We can therefore build a type with 4 elements as

```
Four : Set
Four = T ⊔ T ⊔ T ⊔ T
```

# Finite sets

In Agda, for a fixed  $n$ , it is easy to describe a type with exactly  $n$  elements.

We have a type with 1 element:

```
data T : Set where
  tt : T
```

We can therefore build a type with 4 elements as

```
Four : Set
Four = T ⊔ T ⊔ T ⊔ T
```

We could have taken `Fin 4` but this requires  $\mathbb{N}$  and `Fin`...

whereas this construction is available as soon as we have:  $\top$ ,  $\oplus$  and  $\perp$ .



An element of

`Four : Set`

`Four = T ⊕ (T ⊕ (T ⊕ T))`

looks like `right (right (left tt))`, but we will write `0`, `1`, `2`, `3` for its elements.

An element of

`Four` : `Set`

`Four` =  $\top \uplus (\top \uplus (\top \uplus \top))$

looks like `right (right (left tt))`, but we will write `0`, `1`, `2`, `3` for its elements.

An important property of this type is that a function from `Four` to `Set` amounts to

An element of

`Four` : `Set`

`Four` =  $T \uplus (T \uplus (T \uplus T))$

looks like `right (right (left tt))`, but we will write `0`, `1`, `2`, `3` for its elements.

An important property of this type is that a function from `Four` to `Set` amounts to specifying four types!

# Inductive types

An inductive type consists in

- a finite set of constructors,
- a given number of terms of the type as arguments of each constructor.

For instance,

```
data Nat : Set where  
  zero : Nat  
  suc  : Nat → Nat
```

# Inductive types

An inductive type consists in

- a finite set of constructors,
- a given number of terms of the type as arguments of each constructor.

For instance,

```
data BTree : Set where  
  leaf : BTree  
  node : BTree → BTree → BTree
```

# Inductive types

An inductive type consists in

- a finite set of constructors,
- a given number of terms of the type as arguments of each constructor.

For instance,

```
data BTree : Set where  
  leaf : BTree  
  node : BTree → BTree → BTree
```

We could also allow arguments which are not of the type itself, but we will see that everything generalizes easily.

# Specifying inductive types

We can therefore specify an inductive type by

- a type  $A$  with  $n$  elements: the constructors,
- for each  $x \in A$ , a type  $B(x)$  with  $n_x$  elements: the number of arguments of  $x$ .

For instance,

```
data Nat : Set where
  zero : Nat
  suc   : Nat → Nat
```

is specified by

- $A = \text{Two}$
- $B(0) = \text{Zero}$ ,  $B(1) = \text{One}$

# Specifying inductive types

We can therefore specify an inductive type by

- a type  $A$  with  $n$  elements: the constructors,
- for each  $x \in A$ , a type  $B(x)$  with  $n_x$  elements: the number of arguments of  $x$ .

For instance,

```
data BTree : Set where
  leaf : BTree
  node : BTree → BTree → BTree
```

is specified by

- $A = \text{Two}$
- $B(0) = \text{Zero}$ ,  $B(1) = \text{Two}$



## Specifying inductive types

Note that the types  $A$  or  $B$  don't have to be finite.

For instance,

```
data List (C : Set) : Set where  
  nil    : List C  
  cons   : C → List C → List C
```

## Specifying inductive types

Note that the types  $A$  or  $B$  don't have to be finite.

For instance,

```
data List (C : Set) : Set where
```

```
  nil      : List C
```

```
  cons-c   : List C → List C
```

with one `cons-c` for each  $c : C$ ,

# Specifying inductive types

Note that the types  $A$  or  $B$  don't have to be finite.

For instance,

```
data List (C : Set) : Set where
```

```
  nil      : List C
```

```
  cons-c   : List C → List C
```

with one `cons-c` for each  $c : C$ , is specified by

- $A = \text{One} \uplus C$
- $B(0) = \text{Zero}$ ,  $B(c) = \text{One}$ .

## Specifying inductive types

Trees are usually defined as

## Specifying inductive types

Trees are usually defined as

```
data Tree : Set where  
  node : List Tree → Tree
```

## Specifying inductive types

Trees are usually defined as

`data Tree : Set where`

`node-n : Tree → Tree → ... → Tree → Tree`

with  $n \in \mathbb{N}$ .

## Specifying inductive types

Trees are usually defined as

```
data Tree : Set where  
  node-n : Tree → Tree → ... → Tree → Tree  
with  $n \in \mathbb{N}$ .
```

They can thus be specified with

# Specifying inductive types

Trees are usually defined as

`data Tree : Set where`

`node-n : Tree → Tree → ... → Tree → Tree`

with  $n \in \mathbb{N}$ .

They can thus be specified with

- $A = \mathbb{N}$
- $B(n) = \text{Fin } n$



## W-types

Given types  $A : \text{Type}$  and  $B : \Pi(x : A). \text{Type}$ , we write

$$W(x : A).B$$

for the type whose elements are in the inductive type specified by  $A$  and  $B$ .

## W-types

Given types  $A : \text{Type}$  and  $B : \Pi(x : A). \text{Type}$ , we write

$$W(x : A).B$$

for the type whose elements are in the inductive type specified by  $A$  and  $B$ .

Those can be defined in Agda by

## W-types

Given types  $A : \text{Type}$  and  $B : \Pi(x : A). \text{Type}$ , we write

$$W(x : A).B$$

for the type whose elements are in the inductive type specified by  $A$  and  $B$ .

Those can be defined in Agda by

```
data W (Cons : Set) (Arg : Cons → Set) : Set where
  sup : (c : Cons) (a : Arg c → W Cons Arg) → W Cons Arg
```

## W-types

Given types  $A : \text{Type}$  and  $B : \Pi(x : A). \text{Type}$ , we write

$$W(x : A).B$$

for the type whose elements are in the inductive type specified by  $A$  and  $B$ .

Those can be defined in Agda by

```
data W (Cons : Set) (Arg : Cons → Set) : Set where
  sup : (c : Cons) (a : Arg c → W Cons Arg) → W Cons Arg
```

e.g.

```
 $\mathbb{N} : \text{Set}$ 
 $\mathbb{N} = W \text{ Bool } (\lambda \{ \text{false} \rightarrow \perp ; \text{true} \rightarrow \top \})$ 
```

## W-types

Given types  $A : \text{Type}$  and  $B : \Pi(x : A). \text{Type}$ , we write

$$W(x : A).B$$

for the type whose elements are in the inductive type specified by  $A$  and  $B$ .

Those can be defined in Agda by

```
data W (Cons : Set) (Arg : Cons → Set) : Set where
  sup : (c : Cons) (a : Arg c → W Cons Arg) → W Cons Arg
```

e.g.

```
List : (A : Set) → Set
List A = W (Maybe A) (λ {nothing → ⊥ ; (just x) → T})
```

In fact, we can directly give the rules for W-types.

In fact, we can directly give the rules for W-types.

Formation:

In fact, we can directly give the rules for W-types.

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash W(x : A).B : \text{Type}} \quad (W_F)$$



In fact, we can directly give the rules for W-types.

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash W(x : A).B : \text{Type}} \quad (W_F)$$

Introduction:

In fact, we can directly give the rules for W-types.

Formation:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash W(x : A).B : \text{Type}} \quad (W_F)$$

Introduction:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B[t/x] \rightarrow W(x : A).B}{\Gamma \vdash \text{sup}(t, u) : W(x : A).B} \quad (W_I)$$

Given a constructor  $t$  and arguments  $u_i \in W(x : A).B$ ,  
we have a new term  $t(u_1, \dots, u_n)$ .

Elimination:

## W-types

Elimination:  $\Gamma \vdash t : W(x : A).B \quad \Gamma, x : W(x : A).B \vdash C : \text{Type}$   
$$\frac{\Gamma, x : A, y : B \rightarrow W(x : A).B, z : \Pi(w : B).C[(y\ w)/x] \vdash u : C[\text{sup}(x, y)/x]}{\Gamma \vdash \text{Wrec}(t, x \mapsto C, xyz \mapsto u) : C[t/x]} \quad (W_E)$$

## W-types

Elimination:  $\Gamma \vdash t : W(x : A).B \quad \Gamma, x : W(x : A).B \vdash C : \text{Type}$   
$$\frac{\Gamma, x : A, y : B \rightarrow W(x : A).B, z : \Pi(w : B).C[(y\ w)/x] \vdash u : C[\text{sup}(x, y)/x]}{\Gamma \vdash \text{Wrec}(t, x \mapsto C, xyz \mapsto u) : C[t/x]} \quad (W_E)$$

Computation:

## W-types

Elimination:  $\Gamma \vdash t : W(x : A).B \quad \Gamma, x : W(x : A).B \vdash C : \text{Type}$   
$$\frac{\Gamma, x : A, y : B \rightarrow W(x : A).B, z : \Pi(w : B).C[(y \ w)/x] \vdash u : C[\text{sup}(x, y)/x]}{\Gamma \vdash \text{Wrec}(t, x \mapsto C, xyz \mapsto u) : C[t/x]} \quad (W_E)$$

Computation:  $\Gamma \vdash t : A \quad \Gamma, x : W(x : A).B \vdash C : \text{Type} \quad \Gamma \vdash u : B[t/x] \rightarrow W(x : A).B$   
$$\frac{\Gamma, x : A, y : B \rightarrow W(x : A).B, z : \Pi(w : B).C[(y \ w)/x] \vdash v : C[\text{sup}(x, y)/x]}{\Gamma \vdash \text{Wrec}(\text{sup}(t, u), x \mapsto C, xyz \mapsto v) = v[t/x, u/y, \lambda w. \text{Wrec}(u \ w, x \mapsto C, xyz \mapsto v)/z] : C[s]}$$

## W-types

Elimination:  $\Gamma \vdash t : W(x : A).B \quad \Gamma, x : W(x : A).B \vdash C : \text{Type}$   
$$\frac{\Gamma, x : A, y : B \rightarrow W(x : A).B, z : \Pi(w : B).C[(y\ w)/x] \vdash u : C[\text{sup}(x, y)/x]}{\Gamma \vdash \text{Wrec}(t, x \mapsto C, xyz \mapsto u) : C[t/x]} \quad (W_E)$$

Computation:  $\Gamma \vdash t : A \quad \Gamma, x : W(x : A).B \vdash C : \text{Type} \quad \Gamma \vdash u : B[t/x] \rightarrow W(x : A).B$   
$$\frac{\Gamma, x : A, y : B \rightarrow W(x : A).B, z : \Pi(w : B).C[(y\ w)/x] \vdash v : C[\text{sup}(x, y)/x]}{\Gamma \vdash \text{Wrec}(\text{sup}(t, u), x \mapsto C, xyz \mapsto v) = v[t/x, u/y, \lambda w. \text{Wrec}(u\ w, x \mapsto C, xyz \mapsto v)/z] : C[s]}$$

In Agda:

```
elim : {Cons : Set} {Arg : Cons → Set}
      (C : W Cons Arg → Set) →
      ((c : Cons) → (a : Arg c → W Cons Arg) → C (sup c a)) →
      (x : W Cons Arg) → C x
elim C f (sup c a) = f c a
```

## Some limitations

This is perfectly fine if you want to implement inductive types.

In practice, most provers have a more specific implementation of inductive types because

- we want to be able to give intuitive names to the constructors,
- we want to implement extensions of these inductive types (mutually recursive inductive types, inductive-recursive types, inductive-inductive types),
- some other minor details.



## Some limitations

This is perfectly fine if you want to implement inductive types.

In practice, most provers have a more specific implementation of inductive types because

- we want to be able to give intuitive names to the constructors,
- we want to implement extensions of these inductive types (mutually recursive inductive types, inductive-recursive types, inductive-inductive types),
- some other minor details.

We can be tempted to consider more general inductive types, but this has to be done carefully...

## Limitations for inductive types

Suppose that we want to define  $\lambda$ -terms in Agda, without bothering with  $\alpha$ -conversion. An idea could be to use the fact that Agda already implements this in the meta-theory, so that a  $\lambda$ -term  $\lambda x.t$  could be implemented as a function `Term → Term`!

## Limitations for inductive types

Suppose that we want to define  $\lambda$ -terms in Agda, without bothering with  $\alpha$ -conversion. An idea could be to use the fact that Agda already implements this in the meta-theory, so that a  $\lambda$ -term  $\lambda x.t$  could be implemented as a function `Term → Term`!

Let's try this:

```
data Term : Set where
  abs : (Term → Term) → Term
```

## Limitations for inductive types

Suppose that we want to define  $\lambda$ -terms in Agda, without bothering with  $\alpha$ -conversion. An idea could be to use the fact that Agda already implements this in the meta-theory, so that a  $\lambda$ -term  $\lambda x.t$  could be implemented as a function `Term  $\rightarrow$  Term`!

Let's try this:

```
data Term : Set where
  abs : (Term  $\rightarrow$  Term)  $\rightarrow$  Term
```

It looks fine but we get an error

Term is not strictly positive, because it occurs to the left of an arrow in the type of the constructor abs in the definition of Term.

## Limitations for inductive types

Another example is the following definition of the predicate even:

```
data even : (n : ℕ) → Set where
  even-zero : even zero
  even-suc   : {n : ℕ} → ¬ (even n) → even (suc n)
```

which raises

even is not strictly positive, because it occurs to the left of an arrow in the type of the constructor even-suc in the definition of even.

## Polarity of types

The **polarity** (*positive / negative*) of an implicative type is

- positive for the topmost type,
- stays the same when we go to the right of an arrow,
- changes sign when we go to the left of an arrow.

## Polarity of types

The **polarity** (*positive / negative*) of an implicative type is

- positive for the topmost type,
- stays the same when we go to the right of an arrow,
- changes sign when we go to the left of an arrow.

$$A \rightarrow (B \rightarrow C) \rightarrow D \rightarrow E$$

## Polarity of types

The **polarity** (*positive / negative*) of an implicative type is

- positive for the topmost type,
- stays the same when we go to the right of an arrow,
- changes sign when we go to the left of an arrow.

$$A \rightarrow ((B \rightarrow C) \rightarrow (D \rightarrow E))$$

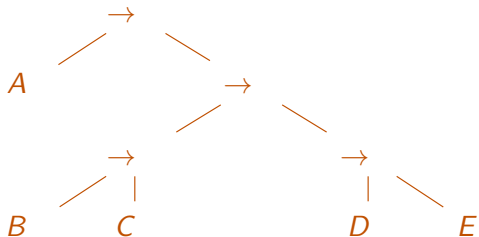


# Polarity of types

The **polarity** (*positive / negative*) of an implicative type is

- positive for the topmost type,
- stays the same when we go to the right of an arrow,
- changes sign when we go to the left of an arrow.

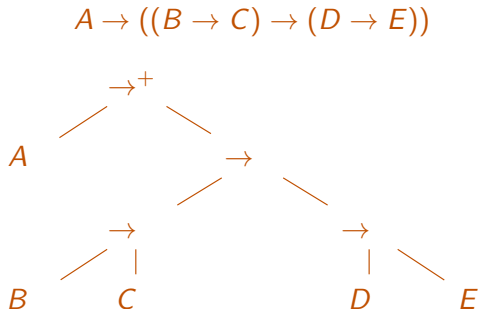
$$A \rightarrow ((B \rightarrow C) \rightarrow (D \rightarrow E))$$



# Polarity of types

The **polarity** (*positive / negative*) of an implicative type is

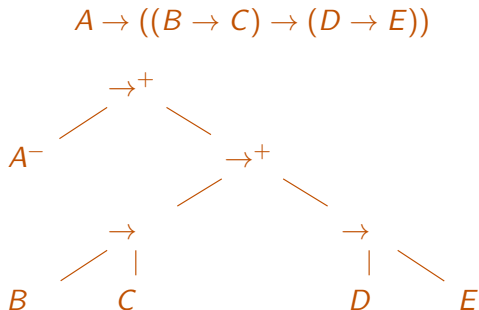
- positive for the topmost type,
- stays the same when we go to the right of an arrow,
- changes sign when we go to the left of an arrow.



# Polarity of types

The **polarity** (*positive / negative*) of an implicative type is

- positive for the topmost type,
- stays the same when we go to the right of an arrow,
- changes sign when we go to the left of an arrow.

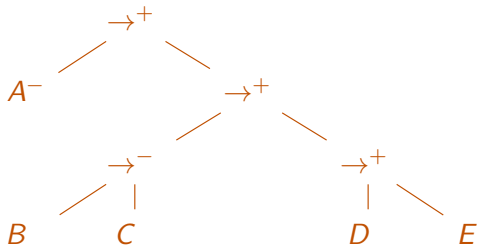


# Polarity of types

The **polarity** (*positive / negative*) of an implicative type is

- positive for the topmost type,
- stays the same when we go to the right of an arrow,
- changes sign when we go to the left of an arrow.

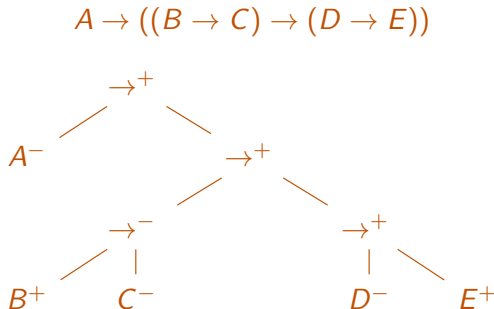
$$A \rightarrow ((B \rightarrow C) \rightarrow (D \rightarrow E))$$



# Polarity of types

The **polarity** (*positive / negative*) of an implicative type is

- positive for the topmost type,
- stays the same when we go to the right of an arrow,
- changes sign when we go to the left of an arrow.



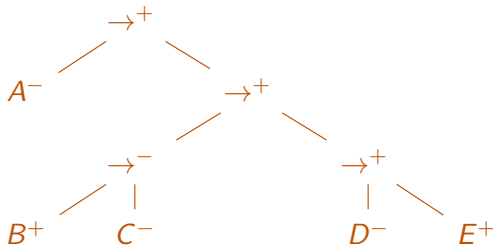
# Polarity of types

The **polarity** (*positive / negative*) of an implicative type is

- positive for the topmost type,
- stays the same when we go to the right of an arrow,
- changes sign when we go to the left of an arrow.

A type is **strictly positive**, when it is positive and did not change sign.

$$A \rightarrow ((B \rightarrow C) \rightarrow (D \rightarrow E))$$



## Limitations for inductive types

Agda imposes the following restriction on inductive types: *all the occurrences of the recursively defined type must occur strictly positively in all arguments of constructors.*

This is why the following gets rejected:

```
data Term : Set where
  abs : (Term → Term) → Term
```

## Limitations for inductive types

Without this restriction, we could write looping terms:



## Limitations for inductive types

Without this restriction, we could write looping terms:

```
{-# NO_POSITIVITY_CHECK #-}  
data Term : Set where  
  abs : (Term → Term) → Term
```

## Limitations for inductive types

Without this restriction, we could write looping terms:

```
{-# NO_POSITIVITY_CHECK #-}  
data Term : Set where  
  abs : (Term → Term) → Term  
  
app : Term → Term → Term  
app (abs f) t = f t
```

## Limitations for inductive types

Without this restriction, we could write looping terms:

```
{-# NO_POSITIVITY_CHECK #-}  
data Term : Set where  
  abs : (Term → Term) → Term  
  
app : Term → Term → Term  
app (abs f) t = f t  
  
 $\omega$  : Term  
 $\omega$  = abs ( $\lambda x \rightarrow$  app x x)
```

## Limitations for inductive types

Without this restriction, we could write looping terms:

```
{-# NO_POSITIVITY_CHECK #-}  
data Term : Set where  
  abs : (Term → Term) → Term  
  
app : Term → Term → Term  
app (abs f) t = f t  
  
 $\omega$  : Term  
 $\omega$  = abs ( $\lambda x \rightarrow$  app x x)  
  
loop : Term  
loop = app  $\omega$   $\omega$ 
```

## Limitations for inductive types

Without this restriction, we could write looping terms:

```
{-# NO_POSITIVITY_CHECK #-}
```

```
data Term : Set where
```

```
  abs : (Term →  $\perp$ ) → Term
```

```
app : Term → Term →  $\perp$ 
```

```
app (abs f) t = f t
```

```
 $\omega$  : Term
```

```
 $\omega$  = abs ( $\lambda$  x → app x x)
```

```
loop :  $\perp$ 
```

```
loop = app  $\omega$   $\omega$ 
```

## Limitations for inductive types

Without this restriction, we could write looping terms:

```
{-# NO_POSITIVITY_CHECK #-}
```

```
data Term : Set where
```

```
  abs : (Term →  $\perp$ ) → Term
```

```
  app : Term → Term →  $\perp$ 
```

```
  app (abs f) t = f t
```

```
 $\omega$  : Term
```

```
 $\omega$  = abs ( $\lambda$  x → app x x)
```

```
loop :  $\perp$ 
```

```
loop = app  $\omega$   $\omega$ 
```

This explains why we forbid negative types

## Limitations for inductive types

Without this restriction, we could write looping terms:

```
{-# NO_POSITIVITY_CHECK #-}
```

```
data Term : Set where
```

```
  abs : (Term →  $\perp$ ) → Term
```

```
  app : Term → Term →  $\perp$ 
```

```
  app (abs f) t = f t
```

```
 $\omega$  : Term
```

```
 $\omega$  = abs ( $\lambda$  x → app x x)
```

```
loop :  $\perp$ 
```

```
loop = app  $\omega$   $\omega$ 
```

This explains why we forbid negative types

(the restriction to strictly positive ones in Agda is not clear to me).

## Limitations for inductive types

The proof can even be further simplified:



## Limitations for inductive types

The proof can even be further simplified:

```
{-# NO_POSITIVITY_CHECK #-}  
data Term : Set where  
  abs : (Term →  $\perp$ ) → Term
```

## Limitations for inductive types

The proof can even be further simplified:

```
{-# NO_POSITIVITY_CHECK #-}  
data Bad : Set where  
  bad : (Bad → ⊥) → Bad
```

## Limitations for inductive types

The proof can even be further simplified:

```
{-# NO_POSITIVITY_CHECK #-}  
data Bad : Set where  
  bad : (Bad → ⊥) → Bad  
not-Bad : Bad → ⊥  
not-Bad (bad f) = f (bad f)
```

## Limitations for inductive types

The proof can even be further simplified:

```
{-# NO_POSITIVITY_CHECK #-}
```

```
data Bad : Set where
```

```
  bad : (Bad →  $\perp$ ) → Bad
```

```
not-Bad : Bad →  $\perp$ 
```

```
not-Bad (bad f) = f (bad f)
```

```
actually-Bad : Bad
```

```
actually-Bad = bad not-Bad
```

## Limitations for inductive types

The proof can even be further simplified:

```
{-# NO_POSITIVITY_CHECK #-}  
data Bad : Set where  
  bad : (Bad → ⊥) → Bad  
  
not-Bad : Bad → ⊥  
not-Bad (bad f) = f (bad f)  
  
actually-Bad : Bad  
actually-Bad = bad not-Bad  
  
absurd : ⊥  
absurd = not-Bad actually-Bad
```

# Inductive-inductive types

Sometimes, inductive types are too limited.

In **inductive-inductive** types, one can define at the same time

- a type  $A : \text{Type}$
- a predicate  $P : A \rightarrow \text{Type}$

There are some conditions for this to be well-defined that we will not dig into.

## Inductive-inductive types

For instance, we can define sorted lists with

```
data SortedList : Set
```

```
data _≤*_ : ℕ → SortedList → Set
```

```
data SortedList where
```

```
  nil : SortedList
```

```
  cons : (x : ℕ) (l : SortedList) (le : x ≤* l) → SortedList
```

```
data _≤*_ where
```

```
  ≤*-empty : {x : ℕ} → x ≤* nil
```

```
  ≤*-cons : {x y : ℕ} {l : SortedList} →
```

```
    x ≤ y → (le : y ≤* l) → x ≤* (cons y l le)
```