CSC_51051_EP: Dependent types

Samuel Mimram

2025

École polytechnique

Dependent types

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

- we would like quantifications to be typed ($\forall x \in A. \ldots$ instead of $\forall x. \ldots$),
- 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 λ-abstractions for terms and for proofs: $\lambda n.t : \forall x \in \mathbb{N}.A$ vs $\lambda x^A.t : A \to B$
- 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 $A \to B$ 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:

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

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

And we add a type Type of all types.

Agda notations

The Agda notation for

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

is

```
(x : A) → 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 $$e, e' \quad ::= \quad x \mid e\, e' \mid \lambda x^{e}.e' \mid \Pi(x : e).e' \mid \mathsf{Type}$$

- x: a variable,
- $e\, e'$: an application,
- $\lambda x^{e}.e'$: an abstraction,
- $\Pi(x : e).e'$: a Π-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 $\mathsf{FV}(e)$ for the free variables of an expression e.

$$
\begin{aligned}
\mathsf{FV}(x) &= \{x\} \\
\mathsf{FV}(t\ u) &= \mathsf{FV}(t) \cup \mathsf{FV}(u) \\
\mathsf{FV}(\lambda x^A.t) &= \mathsf{FV}(A) \cup (\mathsf{FV}(t) \setminus \{x\}) \\
\mathsf{FV}(\Pi(x : A).B) &= \mathsf{FV}(A) \cup (\mathsf{FV}(B) \setminus \{x\}) \\
\mathsf{FV}(\mathsf{Type}) &= \emptyset
\end{aligned}
$$

Expressions are considered up to α-**equivalence**: we can rename bound variables.

Terms: substitution

We write $e[u/x]$ for the expression e where x has been replaced by u.

$$
\begin{aligned}
x[u/x] &= u \\
y[u/x] &= y && \text{if } x \neq y \\
(t\ t')[u/x] &= (t[u/x])\,(t'[u/x]) \\
(\lambda y^{A}.t)[u/x] &= \lambda y^{A[u/x]}.t[u/x] && \text{with } y \notin \mathsf{FV}(u) \cup \{x\} \\
(\Pi(y : A).B)[u/x] &= \Pi(y : A[u/x]).B[u/x] && \text{with } y \notin \mathsf{FV}(u) \cup \{x\} \\
\mathsf{Type}\,[u/x] &= \mathsf{Type}
\end{aligned}
$$

Contexts

A **context** Γ is a list

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

of variables and expressions.

Its **domain** is

$$\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$$

The order really matters here: we can make sense of

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

but not of

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

Judgments

The judgments are more complicated than before:

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

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

to an argument of type

$$\mathrm{Vec}\,(3 + 2)$$

Judgments

We have three forms of **judgments**:

- Γ is a well-formed context:

$$\Gamma \vdash$$

- t has type A under the hypothesis Γ:

$$\Gamma \vdash t : A$$

- t and u are equal terms of type A under the hypothesis Γ:

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

  NB: this is *definitional* equality.

They mutually need each other!

Judgments

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

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

Here, this is represented by judgments of the form

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

Rules: contexts

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

$$\frac{}{\varnothing \vdash} \qquad \frac{\Gamma \vdash A : \mathsf{Type}}{\Gamma, x : A \vdash}$$

For instance,

$$\dfrac{\dfrac{\dfrac{\vdots}{n : \mathbb{N} \vdash \mathsf{Vec} : \Pi(\_ : \mathbb{N}).\ \mathsf{Type}} \qquad \dfrac{}{n : \mathbb{N} \vdash n : \mathbb{N}}\,(\mathsf{ax})}{n : \mathbb{N} \vdash \mathsf{Vec}\ n : \mathsf{Type}}\,(\Pi_{\mathsf{E}})}{n : \mathbb{N}, l : \mathsf{Vec}\ n \vdash}$$

Rules: equality

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

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

and that we can convert terms and types

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = B : \mathsf{Type}}{\Gamma \vdash t : B} \qquad \frac{\Gamma \vdash t = u : A \qquad \Gamma \vdash A = B : \mathsf{Type}}{\Gamma \vdash t = u : B}$$

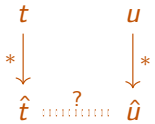
For instance,

$$\dfrac{\dfrac{}{\ldots \vdash f : \Pi(x : \mathsf{Vec}\ 5).A}\ (\mathsf{ax}) \qquad \dfrac{\dfrac{}{\ldots \vdash l : \mathsf{Vec}\,(3+2)}\ (\mathsf{ax}) \qquad \dfrac{\cdots \qquad \ldots \vdash 3+2=5 : \mathbb{N}}{\ldots \vdash \mathsf{Vec}\,(3+2) = \mathsf{Vec}\ 5}}{\ldots \vdash l : \mathsf{Vec}\ 5}}{f : \Pi(x : \mathsf{Vec}\ 5).A, l : \mathsf{Vec}\,(3+2) \vdash f\ l : A}\ (\Pi_{\mathsf{E}})$$

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 β-reduction).

This means that we must ensure that the corresponding reduction is

- terminating, and
- (locally) confluent.

Rules: axiom

The **axiom** rule is

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

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

The side conditions avoid bad jokes such as

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

Rules: Π-types

The last thing we need to introduce is rules for Π-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 (β-equivalence),
5. *uniqueness*: reconstructing a deconsctructed term does nothing (η-equivalence),
6. *congruence*: compatibility with equality.

Rules: Π-types

Formation:

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \Pi(x : A).B : \mathsf{Type}}(\Pi_{\mathsf{F}})$$

Introduction:

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

Elimination:

$$\frac{\Gamma \vdash t : \Pi(x : A).B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B[u/x]}(\Pi_{\mathsf{E}})$$

For instance,

$$\frac{\dfrac{}{f : \Pi(n : \mathbb{N}).\,\mathsf{Vec}\ n \vdash f : \Pi(n : \mathbb{N}).\,\mathsf{Vec}\ n}(\mathsf{ax}) \qquad \dfrac{}{\ldots \vdash 5 : \mathbb{N}}}{f : \Pi(n : \mathbb{N}).\,\mathsf{Vec}\ n \vdash f\ 5 : \mathsf{Vec}\ \ 5}(\Pi_{\mathsf{E}})$$

Rules: Π-types

Computation:

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

(this is β-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 η-equivalence!)

Rules: Π-types

Congruence:

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

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

In the following, we will usually omit congruence rules.

Arrow types

If we write

$$A \to B$$

for

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

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

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

and we recover the usual rules.

Arrow types

For instance, the elimination rule

$$\frac{\Gamma \vdash t : \Pi(x : A).B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B[u/x]}(\Pi_{\mathsf{E}})$$

becomes

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}(\to_{\mathsf{E}})$$

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. Π),
- a constructor for terms (e.g. λ),
- 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:

$$\frac{\Gamma \vdash}{\Gamma \vdash \bot : \mathsf{Type}}\ (\bot_{\mathsf{F}})$$

Introduction: n/a

Elimination:

$$\frac{\Gamma \vdash t : \bot \qquad \Gamma, x : \bot \vdash A : \mathsf{Type}}{\Gamma \vdash \mathsf{bot}(t, x \mapsto A) : A[t/x]}\ (\top_{\mathsf{E}})$$

Computation: n/a

Uniqueness: n/a

The empty type

In Agda, elimination and computation correspond to

```
⊥-elim : (A : ⊥ → Set) → (x : ⊥) → A x
⊥-elim A ()
```

The unit type

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \top : \mathsf{Type}}\ (\top_{\mathsf{F}})$$

Introduction:

$$\frac{\Gamma \vdash}{\Gamma \vdash \star : \top}\ (\top_{\mathsf{I}})$$

Elimination:

$$\frac{\Gamma \vdash t : \top \qquad \Gamma, x : \top \vdash A : \mathsf{Type} \qquad \Gamma \vdash u : A[\star/x]}{\Gamma \vdash \mathsf{top}(t, x \mapsto A, u) : A[t/x]}\ (\top_{\mathsf{E}})$$

In OCaml

- $\top$ is `unit`
- $\star$ is `()`
- $\mathsf{top}(t, x \mapsto A, u)$ is `match t with () -> u`

The unit type

Computation:

$$\frac{\Gamma, x : \top \vdash A : \mathsf{Type} \qquad \Gamma \vdash u : A[\star/x]}{\Gamma \vdash \mathsf{top}(\star, x \mapsto A, u) = u : A[\star/x]}\ (\top_{\mathsf{C}})$$

Uniqueness:

$$\frac{\Gamma \vdash t : \top}{\Gamma \vdash t = \star : \top}\ (\top_{\mathsf{U}})$$

In OCaml,

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

The unit type

In Agda, elimination and computation correspond to

```
⊤-elim : (A : ⊤ → Set) → A tt → (t : ⊤) → A t
⊤-elim A a tt = a
```

Booleans

Formation:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Bool} : \mathsf{Type}}\ (\mathsf{Bool}_\mathsf{F})$$

Introduction:

$$\frac{\Gamma \vdash}{\Gamma \vdash 1 : \mathsf{Bool}}\ (\mathsf{Bool}_\mathsf{I}^1) \qquad \frac{\Gamma \vdash}{\Gamma \vdash 0 : \mathsf{Bool}}\ (\mathsf{Bool}_\mathsf{I}^0)$$

Elimination:

$$\frac{\Gamma \vdash t : \mathsf{Bool} \qquad \Gamma, x : \mathsf{Bool} \vdash A : \mathsf{Type} \qquad \Gamma \vdash u : A[1/x] \qquad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \mathsf{ite}(t, x \mapsto A, u, v) : A[t/x]}\ (\mathsf{Bool}_\mathsf{E})$$

In OCaml:

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

Booleans

Computation:

$$\frac{\Gamma, x : \mathsf{Bool} \vdash A : \mathsf{Type} \qquad \Gamma \vdash u : A[1/x] \qquad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \mathsf{ite}(1, x \mapsto A, u, v) = u : A[1/x]}\ (\mathsf{Bool}^1_{\mathsf{C}})$$

$$\frac{\Gamma, x : \mathsf{Bool} \vdash A : \mathsf{Type} \qquad \Gamma \vdash u : A[1/x] \qquad \Gamma \vdash v : A[0/x]}{\Gamma \vdash \mathsf{ite}(0, x \mapsto A, u, v) = v : A[0/x]}\ (\mathsf{Bool}^0_{\mathsf{C}})$$

Uniqueness:

$$\frac{\Gamma \vdash t : \mathsf{Bool}}{\Gamma \vdash \mathsf{ite}(t, x \mapsto \mathsf{Bool}, 1, 0) = t : \mathsf{Bool}}\ (\mathsf{Bool}_{\mathsf{U}})$$

In OCaml,

$$\texttt{if true then } u \texttt{ else } v = u \qquad \texttt{if } t \texttt{ then true else false} = t$$

$$\texttt{if false then } u \texttt{ else } v = v$$

Booleans

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:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Nat} : \mathsf{Type}}\ (\mathsf{Nat}_\mathsf{F})$$

Introduction:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Z} : \mathsf{Nat}}\ (\mathsf{Nat}_\mathsf{I}^\mathsf{Z}) \qquad \frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathsf{S}(t) : \mathsf{Nat}}\ (\mathsf{Nat}_\mathsf{I}^\mathsf{S})$$

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

Natural numbers

Elimination:

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

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

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

Natural numbers

For instance, the factorial function can be implemented with

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

Natural numbers

Computation:

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

$$\frac{\begin{array}{c}\Gamma \vdash t : \mathsf{Nat} \\ \Gamma, x : \mathsf{Nat} \vdash A : \mathsf{Type} \qquad \Gamma \vdash u : A[\mathsf{Z}/x] \qquad \Gamma, x : \mathsf{Nat}, y : A \vdash v : A[\mathsf{S}(x)/x]\end{array}}{\Gamma \vdash \mathsf{rec}(\mathsf{S}(t), x \mapsto A, u, xy \mapsto v) = v[t/x, \mathsf{rec}(t, x \mapsto A, u, xy \mapsto v)/y] : A[\mathsf{S}(t)/x]}\ (\mathsf{Nat}^{\mathsf{S}}_{\mathsf{C}})$$

In OCaml,

```
natrec 0 u v     =    u
natrec (n+1) u v =    v n (natrec n u v)
```

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} a n = ℕ-elim (λ n → Vec A n) [] (λ n l → a ∷ l) n
```

Natural numbers

Uniqueness:

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

In OCaml,

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

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:

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \times B : \mathsf{Type}} \ (\times_{\mathsf{F}})$$

Introduction:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B} \ (\times_{\mathsf{I}})$$

Elimination:

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

In OCaml,

$$\mathsf{unpair}(t, z \mapsto C, \langle x, y \rangle \mapsto u) \quad = \quad \texttt{match } t \texttt{ with } (x, y) \texttt{ -> } u$$

Products

Computation:

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

Uniqueness:

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

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

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

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

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

Dependent sums

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

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

where we can more generally consider

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

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

Similarly, in type theory, we have

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

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

Dependent sums

Formation:

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \Sigma(x : A).B : \mathsf{Type}}\ (\Sigma_{\mathsf{F}})$$

Introduction:

$$\frac{\Gamma, x : A \vdash B : \mathsf{Type} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/x]}{\Gamma \vdash \langle t, u \rangle : \Sigma(x : A).B}\ (\Sigma_{\mathsf{I}})$$

Elimination:

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

Dependent sums

Computation:

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

Uniqueness:

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

Inductive types

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.

Basic properties

Lemma
If $\Gamma \vdash t : A$ *is derivable then both* $\Gamma \vdash$ *and* $\Gamma \vdash A : \mathsf{Type}$ *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

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

but not

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

Type constructors

We want to manipulate terms of type Type.

A polymorphic identity function has type

$$\mathsf{id} : \Pi(A : \mathsf{Type}).A \to A$$

In Agda:

```
id : (A : Set) → A → A
id A a = a
```

Typing polymorphic identity

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{???}{\vdash \mathsf{Type} : \mathsf{Type}}}{A : \mathsf{Type} \vdash}}{A : \mathsf{Type} \vdash A : \mathsf{Type}}\,(\mathrm{ax})}{A : \mathsf{Type}, x : A \vdash}}{A : \mathsf{Type}, x : A \vdash x : A}\,(\mathrm{ax})}{A : \mathsf{Type} \vdash \lambda x^{A}.x : A \to A}\,(\to_{\mathrm{I}})}{\vdash \lambda A^{\mathsf{Type}}.\lambda x^{A}.x : \Pi(A : \mathsf{Type}).A \to A}\,(\Pi_{\mathrm{I}})$$

The obvious guess

The obvious guess consists in adding the rule

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

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 = 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

```
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
```

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 `=`.

Finite sets in OCaml

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

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

OCaml complains:

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

Finite sets in OCaml

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
```

Finite sets in OCaml

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
```

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:

```
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

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

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 #-}
```

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 = Σ I (λ i → f i ≡ 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

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

Russell's paradox in type theory

`R` is not regular

```
R-nonreg : ¬ (regular R)
R-nonreg reg = reg ((R , reg) , refl)
```

`R` is regular

```
R-reg : regular R
R-reg ((.(set proj₁) , reg) , refl) = R-nonreg reg
```

The world falls apart

```
absurd : ⊥
absurd = R-nonreg R-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.

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

Let's say that

$$\mathsf{Type} : \mathsf{TYPE}$$

where TYPE is the type of "big types".

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

Universes

We thus introduce types

$$\mathsf{Type}_i$$

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

- $\mathsf{Type}_0 = \mathsf{Type}$ is the type of usual/small types,
- Type_1 is the type of "big types",
- Type_2 is the type of "very big types",

and so on.

Universes

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

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

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

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

Universes

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

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

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

and so on.

Universes

The rule for Σ-types is

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

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

```
LIST : Set₁
LIST = Σ Set (λ A → List A)
```

we get the error

```
Set₁ != Set
when checking that the expression Σ Set (λ A → List A) has type Set
```

Part IV

Termination

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
```

Termination

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)
```

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:

```
div2 : ℕ → 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.

$$\begin{array}{cccccc} 0_b = & 1_b = 1 & 2_b = 10 & 3_b = 11 & 4_b = 100 & \ldots \\ \text{bits}(0) = 0 & \text{bits}(1) = 1 & \text{bits}(2) = 2 & \text{bits}(3) = 2 & \text{bits}(4) = 3 & \ldots \end{array}$$

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

```
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 ⩽ 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 `suc n ⩽ suc f`, that `div2 (suc n) ⩽ f`. Namely,

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

This thus follows from two easy lemmas:

- division is increasing: `{m n : ℕ} → m ⩽ n → div2 m ⩽ div2 n`
- a technical result: `(n : ℕ) → div2 (suc n) ⩽ n`

Proof of the technical lemma

We can show (left as exercise):

```
⩽-div2 : {m n : ℕ} → m ⩽ n → div2 m ⩽ div2 n
```

and

```
⩽-suc : {n : ℕ} → n ⩽ suc n
```

Thus,

```
lem : (n : ℕ) → div2 (suc n) ⩽ n
lem zero    = z⩽n
lem (suc n) = ⩽-trans (⩽-div2 ⩽-suc) (s⩽s (lem n))
```

In maths, the second case is

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

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 $n \in \mathbb{N}$.

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 π. Suppose that for every rule

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

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

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 $\quad \ldots R\, x_3\, R\, x_2\, R\, x_1\, R\, x_0$

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

$$\ldots < n_3 < n_2 < n_1 < n_0$$

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

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

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

$$\ldots S\,(n-3)\,S\,(n-2)\,S\,(n-1)\,S\,n$$

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

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

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

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

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

For S on $\mathbb{N}$

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

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 $x \in A$,

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

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. □

Well-founded recursion

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 $g : {\downarrow}x \to B$ associates an element of B. Then there is a unique function $f : A \to B$ 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
```

We consider the relation such that, for every tree $t = \texttt{Node}(a, t_1, t_2)$, 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.

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 $\mathrm{Acc}(A)$ is the smallest set on which well-founded induction would work.

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

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

Lemma
A relation R on A is well-founded iff $A = \mathrm{Acc}(A)$.

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 _<_ : ℕ → ℕ → Set where
  <-base :   {n : ℕ} →          n < suc n
  <-step : {m n : ℕ} → m < n → 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 : ℕ) → y < x → 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 $f\ n = g\ n$ `(<-wellFounded 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)
```

Recursion

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))
```

Recursion

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

W-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.

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.

We have a type with 1 element:

```
data ⊤ : Set where
  tt : ⊤
```

We can therefore build a type with 4 elements as

```
Four : Set
Four = ⊤ ⊎ ⊤ ⊎ ⊤ ⊎ ⊤
```

We could have taken `Fin 4` but this requires $\mathbb{N}$ and `Fin`...
whereas this construction is available as soon as we have: $\top$, $\uplus$ and $\bot$.

Finite sets

An element of

```
Four : Set
Four = ⊤ ⊎ (⊤ ⊎ (⊤ ⊎ ⊤))
```

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 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 BTree : Set where
  leaf : BTree
  node : BTree → BTree → BTree
```

is specified by

- $A = \texttt{Two}$
- $B(0) = \texttt{Zero}$, $B(1) = \texttt{One}$

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

- $A = \mathtt{One} \uplus C$
- $B(0) = \mathtt{Zero}$, $B(c) = \mathtt{One}$.

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

- $A = \mathbb{N}$
- $B(n) = \texttt{Fin}\ n$

W-types

Given types $A : \mathsf{Type}$ and $B : \Pi(x : A).\, \mathsf{Type}$, we write

$$\mathsf{W}(x : A).B$$

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) → ⊤})
```

W-types

In fact, we can directly give the rules for W-types.

Formation:

$$\frac{\Gamma \vdash A : \mathrm{Type} \qquad \Gamma, x : A \vdash B : \mathrm{Type}}{\Gamma \vdash \mathrm{W}(x : A).B : \mathrm{Type}} \; (W_{\mathrm{F}})$$

Introduction:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/x] \to \mathrm{W}(x : A).B}{\Gamma \vdash \mathrm{sup}(t, u) : \mathrm{W}(x : A).B} \; (W_{\mathrm{I}})$$

Given a constructor t and arguments $u_i \in \mathrm{W}(x : A).B$,
we have a new term $t(u_1, \ldots, u_n)$.

W-types

Elimination:
$$\frac{\Gamma \vdash t : \mathsf{W}(x : A).B \qquad \Gamma, x : \mathsf{W}(x : A).B \vdash C : \mathsf{Type} \qquad \Gamma, x : A, y : B \to \mathsf{W}(x : A).B, z : \Pi(w : B).C[(y\ w)/x] \vdash u : C[\mathsf{sup}(x, y)/x]}{\Gamma \vdash \mathsf{Wrec}(t, x \mapsto C, xyz \mapsto u) : C[t/x]}\ (W_{\mathsf{E}})$$

Computation:
$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : \mathsf{W}(x : A).B \vdash C : \mathsf{Type} \qquad \Gamma \vdash u : B[t/x] \to \mathsf{W}(x : A).B \qquad \Gamma, x : A, y : B \to \mathsf{W}(x : A).B, z : \Pi(w : B).C[(y\ w)/x] \vdash v : C[\mathsf{sup}(x, y)/x]}{\Gamma \vdash \mathsf{Wrec}(\mathsf{sup}(t, u), x \mapsto C, xyz \mapsto v) = v[t/x, u/y, \lambda w.\, \mathsf{Wrec}(u\ w, x \mapsto C, xyz \mapsto v)/z] : C[s}$$

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.

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 λ-terms in Agda, without bothering with α-conversion. An idea could be to use the fact that Agda already implements this in the meta-theory, so that a λ-term $\lambda x.t$ could be implemented as a function `Term → Term`!

Let's try this:

```
data Term : Set where
  abs : (Term → Term) → 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.
```

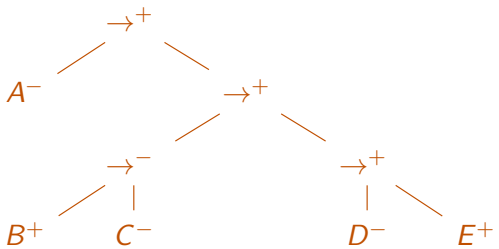
Polarity of types

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.

A type is **strictly positive**, when it is positive and did not change sign.

$$A \to (B \to C) \to D \to E$$



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:

```
{-# NO_POSITIVITY_CHECK #-}
data Term : Set where
  abs : (Term → Term) → Term

app : Term → Term → Term
app (abs f) t = f t

ω : Term
ω = abs (λ x → app x x)

loop : Term
loop = app ω ω
```

Limitations for inductive types

Without this restriction, we could write looping terms:

```
{-# NO_POSITIVITY_CHECK #-}
data Term : Set where
  abs : (Term → ⊥) → Term

app : Term → Term → ⊥
app (abs f) t = f t

ω : Term
ω = abs (λ x → app x x)

loop : ⊥
loop = app ω ω
```

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:

```
{-# 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 $A : \mathrm{Type}$
- a predicate $P : A \to \mathrm{Type}$

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)
```