Induction-recursion and induction-induction in Agda

Madelief Slaats, Sophia Lin

December 6, 2024

Introduction

- Dependently typed
- Functional programming
- Martin-Löf's logical framework: $(x : A) \to B$

- Emacs-based
- Interactive: type checker
- Gradual refinement of code
- Inductive data types
- Pattern matching

Introduction

- Dependently typed
- Functional programming
- Martin-Löf's logical framework: $(x : A) \to B$

- Emacs-based
- Interactive: type checker
- Gradual refinement of code
- Inductive data types
- Pattern matching

Coq

```
Definition not (b:bool) :=
match b with
| true => false
| false => true
end.
```

Agda

```
not :  Bool → Bool
not true = false
not false = true
```

Example

- Natural numbers
- Booleans
- Lists
- Vectors

```
length-concat :  {A : Set} (xs ys :  List A) →
                 length (xs ++ ys) ≡ length xs + length ys
length-concat [] ys = {!  !}
length-concat (x ::  xs) ys = {!  !}
```

Base case:

```
length-concat [] ys = {!  !}
```

Base case:

```
length-concat [] ys = {!  !}
```

We need to prove:

```
length ([] ++ ys)    ≡    length [] + length ys
```

Base case:

```
length-concat [] ys = {!  !}
```

We need to prove:

```
length ([] ++ ys)        ≡        length [] + length ys
length ys
```

Base case:

```
length-concat [] ys = {!  !}
```

We need to prove:

`length ([] ++ ys)`	$\equiv$	`length [] + length ys`
`length ys`		`0 + length ys`

Base case:

```
length-concat [] ys = {!  !}
```

We need to prove:

```
length ([] ++ ys)        ≡       length [] + length ys
length ys                        0 + length ys
                                 length ys
```

Inductive case:

```
length-concat (x ::  xs) ys = {!  !}
```

Inductive case:

```
length-concat (x :: xs) ys = {! !}
```

We need to prove:

```
length ((x :: xs) ++ ys) ≡ length (x :: xs) + length ys
```

Inductive case:

```
length-concat (x ::  xs) ys = {!  !}
```

We need to prove:

```
length ((x ::  xs) ++ ys)  ≡  length (x ::  xs) + length ys
```

By definion of `++`*, we have:*

```
(x ::  xs) ++ ys = x ::  (xs ++ ys)
```

By definition of `length`*, we have:*

```
length (x ::  xs) = 1 + length xs
```

Inductive case:

```
length-concat (x :: xs) ys = {! !}
```

We need to prove:

```
length ((x :: xs) ++ ys) ≡ length (x :: xs) + length ys
length (x :: (xs ++ ys))
```

By definion of `++`*, we have:*

```
(x :: xs) ++ ys = x :: (xs ++ ys)
```

By definition of `length`*, we have:*

```
length (x :: xs) = 1 + length xs
```

Inductive case:

```
length-concat (x :: xs) ys = {! !}
```

We need to prove:

```
length ((x :: xs) ++ ys) ≡ length (x :: xs) + length ys
length (x :: (xs ++ ys))
1 + length (xs ++ ys)
```

By definion of `++`*, we have:*

```
(x :: xs) ++ ys = x :: (xs ++ ys)
```

By definition of `length`*, we have:*

```
length (x :: xs) = 1 + length xs
```

Inductive case:

```
length-concat (x ::  xs) ys = {!  !}
```

We need to prove:

```
length ((x ::  xs) ++ ys)  ≡  length (x ::  xs) + length ys
length (x ::  (xs ++ ys))       1 + length xs + length ys
1 + length (xs ++ ys)
```

By definion of `++`*, we have:*

```
(x ::  xs) ++ ys = x ::  (xs ++ ys)
```

By definition of `length`*, we have:*

```
length (x ::  xs) = 1 + length xs
```

Inductive case:

```
length-concat (x :: xs) ys = {! !}
```

We need to prove:

```
length ((x :: xs) ++ ys)  ≡  length (x :: xs) + length ys
length (x :: (xs ++ ys))        1 + length xs + length ys
1 + length (xs ++ ys)           1 + (length xs + length ys)
```

By definion of `++`*, we have:*

```
(x :: xs) ++ ys = x :: (xs ++ ys)
```

By definition of `length`*, we have:*

```
length (x :: xs) = 1 + length xs
```

$$\texttt{1 + length (xs ++ ys)} \equiv \texttt{1 + (length xs + length ys)}$$

```
1 + length (xs ++ ys)  ≡  1 + (length xs + length ys)
```

We can prove this if we can prove:

```
length (xs ++ ys)  ≡   length xs + length ys
```

```
1 + length (xs ++ ys)  ≡  1 + (length xs + length ys)
```

We can prove this if we can prove:

```
length (xs ++ ys)  ≡   length xs + length ys
```

```
cong :  {A B : Set} {x y :  A} →
                 (f :  A → B) → x ≡ y → f x ≡ f y
```

```
1 + length (xs ++ ys)  ≡  1 + (length xs + length ys)
```

We can prove this if we can prove:

```
length (xs ++ ys)  ≡  length xs + length ys
```

```
cong :  {A B : Set} {x y :  A} →
                (f :  A → B) → x ≡ y → f x ≡ f y
```

Take f to be: λ n → 1 + n

```
1 + length (xs ++ ys)  ≡  1 + (length xs + length ys)
```

We can prove this if we can prove:

```
length (xs ++ ys)  ≡   length xs + length ys
```

```
cong :  {A B : Set} {x y :  A} →
                (f :  A → B) → x ≡ y → f x ≡ f y
```

Take f to be: `λ n → 1 + n`

```
length-concat :  {A : Set} (xs ys :  List A) →
                length (xs ++ ys) ≡ length xs + length ys
```

```
1 + length (xs ++ ys)  ≡  1 + (length xs + length ys)
```

We can prove this if we can prove:

```
length (xs ++ ys)  ≡   length xs + length ys

cong :  {A B : Set} {x y :  A} →
                (f :  A → B) → x ≡ y → f x ≡ f y
```

Take f to be: λ `n → 1 + n`

```
length-concat :  {A : Set} (xs ys :  List A) →
                length (xs ++ ys) ≡ length xs + length ys

cong (λ n → 1 + n) (length-concat xs ys)
```

Induction-recursion

Recall: Definition of an **inductive** type together with a **recursive** function.

```
Code   :  Set
decode :  Code → Set

eq :  (C : Code) → (x y :  decode C) → Code
eq (Π A B) x y = {!  !}
```

```
Code : Set
decode : Code → Set

eq : (C : Code) → (x y : decode C) → Code
eq (Π A B) x y = {! !}
```

$\Pi\ A\ B := \prod_{c:A} B(c)$

```
Π : (A : Code) → (decode A → Code) → Code
```

```
Code   :  Set
decode :  Code → Set

eq :  (C : Code) → (x y :  decode C) → Code
eq (Π A B) x y = {!  !}
```

$\Pi\ A\ B := \prod_{c:A} B(c)$

```
Π :  (A : Code) → (decode A → Code) → Code

decode (Π A B) = (c :  decode A) → decode (B c)
```

```
Code   :  Set
decode :  Code → Set

eq :  (C : Code) → (x y :  decode C) → Code
eq (Π A B) x y = {!  !}
```

Π A B := $\prod_{c:A} B(c)$

```
Π :  (A : Code) → (decode A → Code) → Code

decode (Π A B) = (c :  decode A) → decode (B c)

x, y = (c :  decode A) → decode (B c)
```

```
Code  :  Set
decode :  Code → Set

eq :  (C : Code) → (x y :  decode C) → Code
eq (Π A B) x y = {!  !}


Π A B := ∏c:A B(c)
Π :  (A : Code) → (decode A → Code) → Code


decode (Π A B) = (c :  decode A) → decode (B c)
x, y = (c :  decode A) → decode (B c)
(x c), (y c) = decode (B c)
```

```
Code   :  Set
decode :  Code → Set

eq :  (C : Code) → (x y :  decode C) → Code
eq (Π A B) x y = Π A {!  !}
```

$\Pi\ A\ B := \prod_{c:A} B(c)$

```
Π :  (A : Code) → (decode A → Code) → Code

decode (Π A B) = (c :  decode A) → decode (B c)
x, y = (c :  decode A) → decode (B c)
(x c), (y c) = decode (B c)
```

```
Code :  Set
decode :  Code → Set

eq :  (C : Code) → (x y :  decode C) → Code
eq (Π A B) x y = Π A (λ c → {!  !})
```

$$\Pi\ A\ B := \prod_{c:A} B(c)$$

```
Π :  (A : Code) → (decode A → Code) → Code

decode (Π A B) = (c :  decode A) → decode (B c)

x, y = (c :  decode A) → decode (B c)

(x c), (y c) = decode (B c)
```

```
Code   :  Set
decode :  Code → Set

eq :  (C : Code) → (x y :  decode C) → Code
eq (Π A B) x y = Π A (λ c → eq (B c) (x c) (y c))
```

Π A B := $\prod_{c:A} B(c)$

```
Π :  (A : Code) → (decode A → Code) → Code

decode (Π A B) = (c :  decode A) → decode (B c)

x, y = (c :  decode A) → decode (B c)

(x c), (y c) = decode (B c)
```

Induction-induction

Recall: Definition of an **inductive** type together with an **inductive** family.

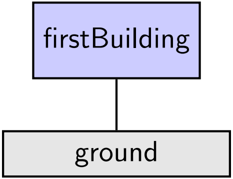
```
mutual
  data Platform : Set where
    ground : Platform
    extension : (p : Platform) → Building p → Platform

  data Building : Platform → Set where
    onTop : (p : Platform) → Building p
    hangingUnder : {p : Platform} → (b : Building p) →
                                    Building (extension p b)
```

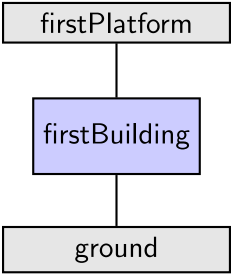
`ground : Platform`

ground

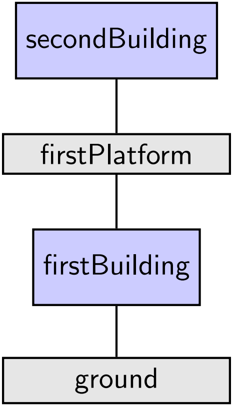
```
firstBuilding :  Building ground
firstBuilding = onTop ground
```



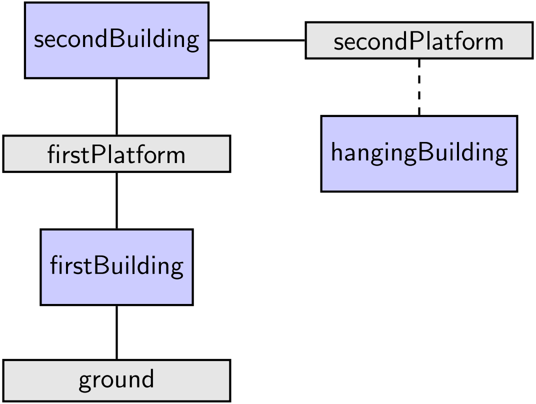
```
   firstPlatform :  Platform
firstPlatform = extension ground firstBuilding
```



```
secondBuilding :  Building firstPlatform
secondBuilding = onTop firstPlatform
```



```
hangingBuilding :  Building (extension firstPlatform secondBuilding)
        hangingBuilding = hangingUnder secondBuilding
```



Thanks for listening, any questions?