Proofs are Programs

Basics - A small introduction to functional Programming and simple Proofs

Imperative vs. Functional Programming

```
int n = …  ;
int res = 1;
for (int i = 1, i < n, i++){
  res = res * i;
}
```

$$factorial \in \mathbb{N} \to \mathbb{N}$$

$$factorial(n) = \begin{cases} 1 & n = 0 \\ n * factorial(n-1) & n > 0 \end{cases}$$

- Program = sequence of instructions
- Stateful (variables)

Imperative vs. Functional Programming

```
int n = …  ;
int res = 1;
for (int i = 1, i < n, i++){
  res = res * i;
}
```

- Program = sequence of instructions
- Stateful (variables)

$$factorial \in \mathbb{N} \to \mathbb{N}$$

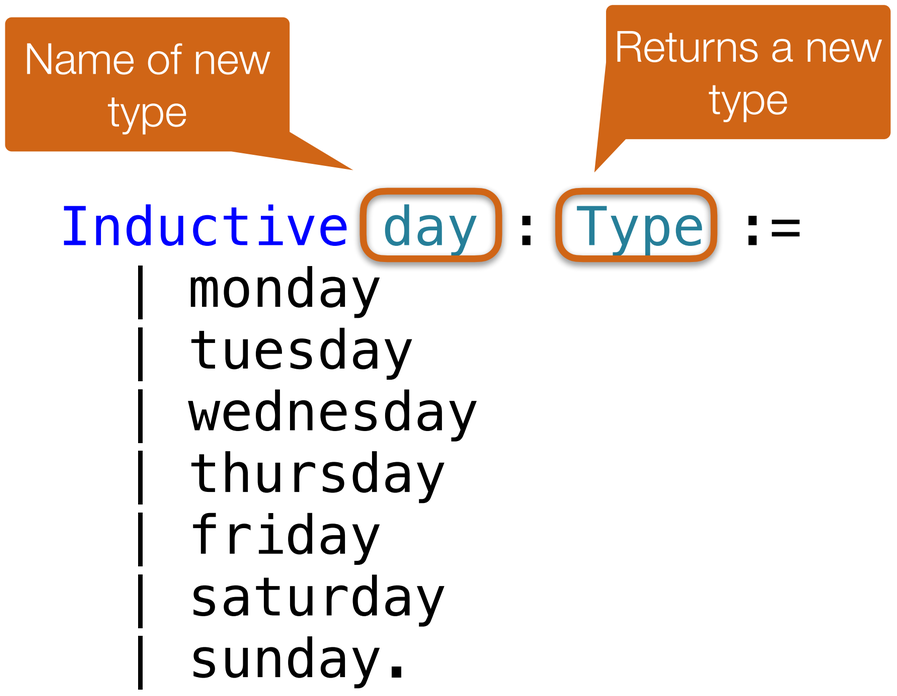
$$factorial(n) = \begin{cases} 1 & n = 0 \\ n * factorial(n-1) & n > 0 \end{cases}$$

- Function = Mapping from input to output values
- No state
- No side effects

Enumerated Types

```
Inductive day : Type :=
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
  | sunday.
```

Enumerated Types



Enumerated Types

Name of new type

Returns a new type

```
Inductive day : Type :=
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
  | sunday.
```

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday    => tuesday
  | tuesday   => wednesday
  | wednesday => thursday
  | thursday  => friday
  | friday    => monday
  | saturday  => monday
  | sunday    => monday
  end.
```

Enumerated Types

Name of new type

Returns a new type

```
Inductive day : Type :=
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
  | sunday.
```

Returns a value of type day

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday    => tuesday
  | tuesday   => wednesday
  | wednesday => thursday
  | thursday  => friday
  | friday    => monday
  | saturday  => monday
  | sunday    => monday
  end.
```

Enumerated Types

Name of new type

Returns a new type

```
Inductive day : Type :=
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
  | sunday.
```

Returns a value of type day

```
Definition next_weekday (d:day) : day :=
  match d with
    | monday    => tuesday
    | tuesday   => wednesday
    | wednesday => thursday
    | thursday  => friday
    | friday    => monday
    | saturday  => monday
    | sunday    => monday
  end.
```

Evaluating Functions

```
Compute (next_weekday friday).
(* ==> monday : day *)
```

Evaluates a function

Evaluating Functions

Evaluates a function

```
Compute (next_weekday friday).
(* ==> monday : day *)
```

```
Compute (next_weekday (next_weekday saturday)).
(* ==> tuesday : day *)
```

Booleans

```
Inductive bool : Type :=
  | true
  | false.
```

Booleans

```
Inductive bool : Type :=
  | true
  | false.
```

```
Definition negb (b:bool) : bool :=
  match b with
  | true => false
  | false => true
  end.
```

Booleans

```
Inductive bool : Type :=
  | true
  | false.
```

```
Definition negb (b:bool) : bool :=
  match b with
  | true => false
  | false => true
  end.
```

```
Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => b2
  | false => false
  end.
```

Cascading Function Definitions

```
Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => true
  | false => b2
  end.
```

Cascading Function Definitions

```
Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => true
  | false => b2
  end.
```

```
Compute (orb true false).
(* ==> true : bool *)
```

```
Compute (orb false false).
(* ==> false : bool *)
```

Notations (for convenience)

```
Notation "x && y" := (andb x y).

Notation "x || y" := (orb x y).
```

```
Compute (false || false || true).
(* ==> true: bool *)
```

Conditionals

```
Definition negb' (b:bool) : bool :=
  if b then false
  else true.
```

Conditionals

```
Definition negb' (b:bool) : bool :=
  if b then false
  else true.
```

```
Inductive bw : Type :=
  | black
  | white.
```

```
Definition invert (x: bw) : bw :=
  if x then white
  else black.
```

Conditionals

```
Definition negb' (b:bool) : bool :=
  if b then false
  else true.
```

```
Inductive bw : Type :=
  | black
  | white.
```

```
Definition invert (x: bw) : bw :=
  if x then white
  else black.
```

```
Compute (invert black).
(* ==> white: bw *)
```

```
Compute (invert white).
(* ==> black: bw *)
```