W-types

Bálint Kocsis Márk Széles

December 7, 2021

Examples of inductive types

```
inductive nat : Type
| zero : nat
| succ : nat → nat
open nat

inductive list (τ : Type u) : Type u
| nil : list
| cons : τ → list → list

inductive infinitree (σ : Type u)(τ : Type v) : Type (max u v)
| leaf : σ → infinitree
| node : τ → (nat → infinitree) → infinitree

inductive vec (τ : Type u) : nat → Type u
| nil : vec zero
| cons : ∀ (n : nat), τ → vec n → vec (succ n)

inductive leq : nat → nat → Prop
| leq_refl : ∀ (n : nat), leq n n
| leq_succ : ∀ (n m : nat), leq n m → leq n (succ m)

inductive acc (τ : Sort u) (r : τ → τ → Prop) : τ → Prop
| intro : ∀ (x : τ), (∀ (y : τ), r x y → acc y) → acc x
```

Motivation for W-types

- Formally treating inductive types is complicated
- One type to rule them all: all inductive types can be reduced to the W-type, assuming the existence of a few simple inductive types

Inductive types as trees

- Terms: syntax trees
- Constructors: nodes
- Recursive parameters: children
- W-types terminology: shapes and positions

Examples - `nat`

```
inductive nat : Type
| zero : nat
| succ : nat → nat
```

- 2 shapes: `zero` and `succ`
- `zero`: 0 positions
- `succ`: 1 position

Examples - `list` (first try)

```
inductive list (τ : Type u) : Type u
| nil : list
| cons : τ → list → list
```

- 2 shapes: `nil` and `cons`
- `nil`: 0 positions
- `cons`: 1 position + 1 non-recursive parameter?

Examples - `list` in reality

```
inductive list (τ : Type u) : Type u
| nil : list
| cons : τ → list → list
```

- 1 shape for `nil`
- 1 shape for every element of τ (`cons`)
- `nil`: 0 positions
- `cons x`: 1 position

Examples - `infinitree`

```
inductive infinitree (σ : Type u)(τ : Type v) : Type (max u v)
| leaf : σ → infinitree
| node : τ → (nat → infinitree) → infinitree
```

- σ many shapes for `leaf`
- τ many shapes for `node`
- `leaf x`: 0 positions
- `node y`: 1 position for every natural number

Formalizing the "shapes and positions" intuition

- Shapes and positions are encoded by types
- The type of positions depends on the shape
- One (closed) inhabitant for every shape and position

Formalizing the "shapes and positions" intuition

Let's take a look at `nat`:

- 2 shapes $\rightarrow$ we need a type with two (closed) inhabitants, e.g. `bool`
- 0 positions for `ff` (`zero`) $\rightarrow$ empty type
- 1 position for `tt` (`succ`) $\rightarrow$ unit type
- For a shape `b` : `bool`, this can be expressed as

```
if b then empty else unit
```

Formalizing the "shapes and positions" intuition

```
def α_nat : Type := bool

def β_nat : α_nat → Type
| ff := empty
| tt := unit

inductive nat : Type
| sup : ∀ (a : α_nat), (β_nat a → nat) → nat
open nat

def zero : nat := sup ff (λ h, empty.rec _ h)

def succ (n : nat) : nat := sup tt (λ _, n)

def rec {C : Type*}(z : C)(g : nat → C → C) : nat → C
| (sup ff _) := z
| (sup tt f) := g (f ()) (rec (f ()))
```

In general: the W-type

```
inductive W (α : Type u) (β : α → Type v) : Type (max u v)
| sup : ∀ (a : α), (β a → W) → W
```

- α encodes the **shapes**
- For an $a : \alpha$, $\beta\ a$ encodes the **positions** for a.
- Constructor takes a shape, and recursive arguments for all positions

Some additional types used for the encoding

In order to do the encoding for all types, we also need some simple inductive types, for example:

```
inductive sum (α : Type u) (β : Type v) : Type (max u v)
| inl : α → sum
| inr : β → sum

inductive sigma (α : Type u)(β : α → Type v) : Type (max u v)
| mk : forall (x : α), β x → sigma

inductive empty : Type

inductive unit : Type
| star : unit

inductive eq {α : Sort*} (x : α) : α → Prop
| refl : eq x
```

Examples

See the Lean file...

Bad news

```
def zero : nat := sup ff (λ h, empty.rec _ h)
def zero' : nat := sup ff (λ h, zero)
```

- There is no canonical choice of representation
- `zero` $\neq$ `zero'`, although `zero` $=$ `zero'` from `funext`

Bad news

```
def rec_dep {C : nat → Type*}(z : C zero)
  (g : ∀ (n : nat), C n → C (succ n)) : ∀ (n : nat), C n
```

- We must explicitly coerce along propositional equalities when defining the dependent eliminator (see Lean file)
- This also happens in the `succ` case, as we have no algorithmic uniqueness rules
- In an extensional setting (e.g. a set theoretic model), this is not a problem

Out of context cat pictures