Theorembeweiserpraktikum

Decidability and Type Classes

Jakob von Raumer, Sebastian Ullrich | SS 2021

Normalisation of Terms

- If we do not assume any global axioms via `axiom`, Lean is able to compute any function:

```
#eval if (5 % 3 = 2) then "This" else "That" -- outputs "This"
```

- Every closed (= not depending on any free variables) expression of a type A can be reduced to a repeated application of constructors of A. This is called *normalisation*.
- That's convenient: Let's define a function that returns 1 if the Riemann hypothesis is true and 0 otherwise:

```
def RH : Prop := _ -- add a formalization of the proposition that is the Riemann hypothesis here

def RH_nat : Nat := if RH then 1 else 0
```

- Unfortunately gives the output

```
failed to synthesize instance Decidable RH
```

Decidability

The type `Decidable` ist just an inductive type that looks very much like `Or` :

```
class inductive Decidable (p : Prop) : Type where
  | isFalse (h : ¬ p) : Decidable p
  | isTrue  (h : p) : Decidable p
```

The difference to `p ∨ ¬p` is that this is an inductive *data type* in `Type` .

Note: We can always match on objects of inductive data types, but we can only match on objects of inductive propositions when our goal is a proposition! This prevents that our definitions can make a difference on *which* proof of a premise we give.

The Law of Excluded Middle

- The only thing that stands between us and our fame is an instance of something like `RH ∨ ¬RH`.
- We need to choose our poison:
  1. Get better at analytic number theory and either prove or refute the Riemann hypothesis.
  2. Write 'open Classical', making all propositions decidable. But then, we're not axiom free anymore and lose computability!

```
open Classical

noncomputable def RH_nat : Nat := if RH then 1 else 0
```

- Wait, but how did Lean compute the first example, then?
- Even without `Classical`, we are able to show decidability for *some* propositions.
- Lean uses *type classes* to automate decidability proofs!

Decidability of the Equality on `Nat`

Karlsruhe Institute of Technology

To prove that equality on `Nat` is decidable, we match on both numbers:

```
theorem decideNatEq (m n : Nat) : Decidable (m = n) :=
  match m, n with
  | zero,   zero   => isTrue rfl
  | zero,   succ n => isFalse (fun h => by injection h)
  | succ m, zero   => isFalse (fun h => by injection h)
  | succ m, succ n =>
    match decideNatEq m n with
    | isTrue h  => isTrue (h ▸ rfl)
    | isFalse h => isFalse (fun h' => by injection h' with h'; exact h h')
```

The tactic `injection` proves that constructors like `succ` are injective and that they are distinct, i. e. that `¬(zero = succ n)`.

Deciding More Complex Terms

- Let's give Lean something tougher to chew on:

```
#eval if (4, 2) = (5 - 1, 1 + 1) ∧ (["Hello", "World"].length < 3) then 0 else 1 -- outputs 0
```

- This is how `if ... then ... else ...` is defined:

```
def ite (c : Prop) [h : Decidable c] (t e : α) : α :=
  match h with
  | isTrue _  => t
  | isFalse _ => e
```

- This is how decidability of conjunction is proved:

```
instance [dp : Decidable p] [dq : Decidable q] : Decidable (And p q) :=
  match dp with
  | isTrue hp =>
    match dq with
    | isTrue hq => ...
```

What Do the Square Brackets Stand for?

- Lean's way to automatically track closure properties is by *type classes*.
- Using type classes happens in three steps:
  1. Mark an inductive type, proposition, or structure as `class` (in place of `structure` in the latter case)
  2. Generate instances of the type class by marking theorems or definitions by using `instance` instead of `theorem` or `def`.

     The declaration name is optional for `instance`, since we usually use them implicitly.
  3. Mark arguments of theorems or definitions using the type class to use Lean's inference mechanism by marking them with square brackets `[inst : MyTypeClass]` or `[MyTypeClass]` instead of curly brackets.

  Lean will then try to recursively fill in those arguments with the available instances.

Examples of Type Classes

- `Decidable p` for a proposition `p : Prop`.
- `Nonempty α` for a type `α : Type`.
- Algebraic structures on a given type (or two types, e. g. vector spaces).

Type Classes and Notation

We can use type class resolution to resolve the "canonical" structure behind a notation:

```
-- 1. Define the type class
class HasMul (α : Type) : Type where
  mul : α → α → α

#check @HasMul.mul -- Prints HasMul.mul : {α : Type} → [self : HasMul α] → α → α → α

-- 2. Define the notation itself
infixl:65 " *' " => HasMul.mul

-- 3. Instantiate the type class
instance : HasMul Nat := { mul := Nat.mul }

-- 4. Profit
#eval 5 *' 2 -- outputs 10
```

Be Careful with Type Class Instances

- Type class instance resolution happens silently, so we do not automatically get feedback on which type class instance Lean found.
- It is good style to make sure only one instance exists for each set of parameters or they are at least equal.
- We can print the name of the instance with the command `#synth HasMul Nat`.