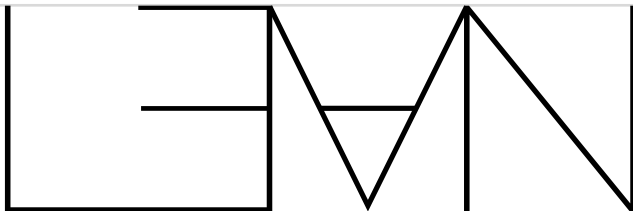




Theorembeweiserpraktikum

Even More Tactics

Jakob von Raumer, Sebastian Ullrich | SS 2022



THEOREM PROVER

General Hints

- Check *carefully* whether your implementation makes sense to you before starting the verification
 - *Only* `while` should be approximated
- There's nothing wrong with adapting the implementation to make verification easier
 - pair-returning function vs. pair of separate functions?
- There's nothing wrong with starting with a hacky, unstructured proof, then refactoring it when it's done

refine

We already know we can arbitrarily nest terms and tactics:

```
example (hr : r) (hrp : r → p) (hq : q) : p ∧ q := by
  exact ⟨(by simp_all), hq⟩
```

We can use `refine` to move out nested tactic blocks

```
refine ⟨?p, hq⟩
case p => -- t p
  simp_all
```

apply e where $e : (h : p) \rightarrow \dots \rightarrow q$ can be thought of as a special case of `refine`: `refine e ?h ...`

Even More about induction

We can also specify a “pre”-tactic to apply to all cases of `induction` / `cases`

```
example (n m : Nat) : n + m = m + n := by
  induction n with
    simp only [(· + ·), Add.add, Nat.add]
  | zero => done           --  $\vdash \text{Nat.add Nat.zero } m = m$ 
  | succ n ih => done     --  $\dots \vdash \text{Nat.add (Nat.succ } n) m = \text{Nat.succ (Nat.add } m n)$ 
```

The proof should still be structured though, so no `simp !`

Case Witnesses

Like the “dependent if” `if h : p then _ else _`, `cases` / `match` can take a “witness” variable that holds a proof of the equation that must have held in the respective case:

```
cases h : f x with
| zero => done    -- h : f x = Nat.zero ⊢ ...
| succ x' => done -- h : f x = Nat.succ x' ⊢ ...
```

⇒ usually the right thing to do when there's a `match` on a non-variable in the goal

Fun with Recursors

We've learned that pattern matching (and induction and recursion) is expressed internally via *recursors*:

```
recursor Option.rec.{u_1, u} : {α : Type u} → {motive : Option α → Sort u_1} →  
  motive none →  
  ((val : α) → motive (some val)) →  
  (t : Option α) → motive t
```

Fun with Recursors

We can also write our own recursor and use them in `cases` / `induction` !

```

theorem Option.rec_rec (p : Option (Option α) → Prop)
  (hsome_some : ∀ x, p (some (some x)))
  (hsome_none : p (some none))
  (hnone : p none) : ∀ o, p o
| some (some x) => hsome_some x
| some none => hsome_none
| none => hnone

example (p : Option (Option α) → Prop) : p o := by
cases o using Option.rec_rec with
| hsome_some x => done -- case hsome_some: ⊢ p (some (some x))
| _ => done -- case hsome_none: ..., case hnone: ...

-- also works without `with`
cases o using Option.rec_rec <|> simp_all

```

The Splitter

`split / split at h` automatically splits the goal into one new goal per nested `if` or `match` branch in the goal/ `h`.

Generated case names are relatively meaningless, so recommended only in combination with `<;>`:

```
example : (if n = 0 then n else 0) = 0 := by
  split <;> simp_all
```


The Splitter

split / split **at** h automatically splits the goal into one new goal per nested **if** or **match** branch in the goal/ h .

Generated case names are relatively meaningless, so recommended only in combination with **<;>** :

```
example : (if n = 0 then n else 0) = 0 := by
  split <;> simp_all
```

Use **unfold** to expose internal **match** of functions defined by pattern matching.

```
def bor : Bool → Bool → Bool
| true, _ => true
| _, true => true
| _, _   => false

example : bor b true = true := by
  -- cases b <;> simp [bor]
  unfold bor; split <;> simp_all
```

Tactics Used in Our Solution

57	simp_all	1	refine
56	simp	1	contradiction
	25	1	only
27	cases	1	apply
	9	1	using
17	exact		
15	intro		
14	injection		
13	induction		
10	have		
10	match		
8	rw		
8	case		
6	rfl		
6	split		
3	intros		
3	assumption		
2	trivial		