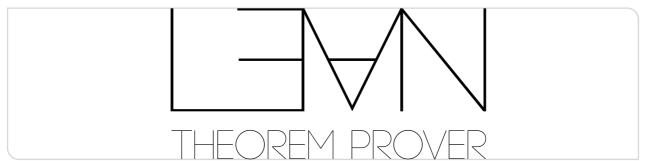




Theorembeweiserpraktikum

Even More Tactics

Jakob von Raumer, Sebastian Ullrich | SS 2021



focus vs. •



A correction: focus does not actually force the first goal to be closed

```
example (hp : p) (hq : q) : p \land q := by
  constructor
  focus -- 'r p'
    skip
  focus
   -- still `+ p`
   exact ha -- tupe mismatch
```

We introduced • (input as \centerdot) as a (prettier) way to do just that:

```
constructor
· skip -- unsolved goals: ... ⊢ p
· exact hg -- is still checked
```

This is basically an unnamed case While focus is not strictly structuring given this insight, we'll allow both this semester



What's Up with Those Dots Anyway

We've seen · before: in terms, it can be used as a nameless lambda

```
set_option pp.binder_types false  
#check (\cdot :: \cdot) -- fun a a_1 => a :: a_1 : ...  
#check (1 + 2 * \cdot) -- fun a => 1 + 2 * a : Nat \rightarrow Nat  
#check [\theta, 1].map (\cdot.succ) -- List.map (fun a => Nat.succ a) [\theta, 1] : List Nat
```

All occurrences of · are bound to the nearest surrounding parentheses, from left to right

refine



We already know we can arbitrarily nest terms and tactics:

```
example (hr : r) (hrp : r \rightarrow p) (hq : q) : p \land q := by exact \langle (by simp_all), hq\rangle
```

We can use refine to move out nested tactic blocks

```
refine <?p, hq>
case p => -- + 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 [(\cdot + \cdot), Add.add, Nat.add]
   zero => done -- + Nat.add Nat.zero m = m
   succ n ih => done -- ... + Nat.add (Nat.succ n) m = 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 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 \Rightarrow done --h: f x = Nat.zero + ...
  succ x' \Rightarrow done -- h : f x = Nat.succ x' + ...
```





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

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





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

```
theorem Option.rec_rec (p : Option (Option \alpha) \rightarrow Prop)
    (hsome_some : \forall x, p \text{ (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 \alpha) \rightarrow Prop) : p o := by
  cases o using Option.rec_rec with
   hsome_some x => done -- case hsome_some: F p (some (some x))
  => done -- case hsome none: .... case hnone: ...
 -- also works without `with`
 cases o using Option.rec_rec <;> simp_all
```