interactive theorem proving

Why? To become better mathematicians.

But interactive theorem proving makes it harder rather than easier to prove stuff... so let's work on that!
interactive theorem proving

Dreams:
- interactive style, in natural language
- effective automation, preserving human comprehensibility.
  - finishing tactics disposing of boring goals
  - interactive tactics that don't explode
- extensibility by users: mathematics, parsing, automation.
interactive theorem proving

Today: experiments in Lean

- dependent types, very similar to Coq
- meta-programming happens in the same language
- developed at MSR and CMU
- maths library/automation/tooling/code generation all ‘works in progress’.
- Lean 4 out ... ‘soon’?
Dependent type theory comes naturally to mathematicians:

```
structure Presheaf :=
(X : Top.{v})
(∅ : (open_set X) → C)

structure Presheaf_hom (F G : Presheaf.{u v} C) :=
(f : F.X → G.X)
(c : G.∅ → ((open_set.map f) ∘ F.∅))
```

— also: “3 is not a topology on 2”
A demo?

```coffeescript
theorem infinitude_of_primes (N : ℕ) : ∃ p ≥ N, prime p :=
begin
  let M := fact N + 1,
  let p := min_fac M,
  have pp : prime p, back [ne_of_gt],
  split,
  { by_contradiction, 
    simp at a, 
    have h₁ : p ∣ M, back,
    have h₂ : p ∣ fact N, back [prime.pos, le_of_lt],
    have h : p ∣ 1, back [nat.dvd_add_iff_right],
    back [prime.not_dvd_one],
  },
  assumption,
end
```
where next?

- Lean makes it significantly easier to write new tactics
- It's still way too cumbersome to write mathematics in Lean.
- As it becomes possible for mathematicians to write tactics (not just language developers), this may rapidly change!
- Meta programming happens in Lean, under the [meta] keyword
- Monadic programming to interact with tactic_state
- Pattern matching and antiquotations for expr munging
where next?

Category Theory in Lean

• Can we write enough automation, so that we can write 'human-like' proofs?
  (i.e. omitting lots of detail!)

Unimath:

Coq:

Isabelle:
variables (C : Type u1) [anon : category.{u1 v1} C]
include anon

def yoneda : C ⇒ ((C^op) ⇒ (Type v1)) := λ X, λ Y : C, Y → X.

def yoneda_evaluation : (((C^op) ⇒ (Type v1)) × (C^op)) ⇒ (Type (max u1 v1)) :=
  (evaluation (C^op) (Type v1)) ⧵ lift_functor.{v1 u1}

 @[simp] lemma yoneda_evaluation_map_down
  (P Q : (C^op ⇒ Type v1) × (C^op)) (α : P → Q) (x : (yoneda_evaluation C) P):
  ((yoneda_evaluation C).map α x).down = (α.1) (Q.2) ((P.1).map (α.2) (x.down)) := refl

def yoneda_pairing : (((C^op) ⇒ (Type v1)) × (C^op)) ⇒ (Type (max u1 v1)) :=
let F := (category_theory.prod.swap ((C^op) ⇒ (Type v1)) (C^op)) in
let G := (functor.prod ((yoneda C).op) (functor.id ((C^op) ⇒ (Type v1)))) in
let H := (functor.hom ((C^op) ⇒ (Type v1))) in
  (F ⧵ G ⧵ H)

 @[simp] lemma yoneda_pairing_map
  (P Q : (C^op ⇒ Type v1) × (C^op)) (α : P → Q) (β : (yoneda_pairing C) (P.1, P.2)):
  (yoneda_pairing C).map α β = (yoneda C).map (α.snd) ⧵ β ⧵ α.fst := refl

def yoneda_lemma : (yoneda_pairing C) ≃ (yoneda_evaluation C) :=
{ hom := λ F x, ulift.up ((x.app F.2) (1 F.2)) },
  inv := λ F x, { app := λ X a, (F.1.map a) x.down } }.
How does this work?


- an algorithm for automatic rewriting, using an edit distance heuristic and some machine learning.

  (in progress, w/ Keeley Hoek, ANU)
rewrite_search proves equational goals by rewriting subexpressions using specified (or discovered) lemmas.

A depth or breadth first search of the rewrite graph would be hopeless for all but the most trivial goals.

The basic version of rewrite_search uses an edit distance minimising search.
- We search from both sides of the goal $A=B$ simultaneously.

- Pretty print each side, and calculate edit distances.

- We track a list of interesting pairs, $A', B'$ with small edit distance

- At each step we consider a rewrite of $A'$ or $B'$ for the most interesting pair at that point.

- In the basic version, most interesting means smallest edit distance $d(A', B')$. 
Generalisations

- use A* rather than greedy search

(so most interesting is the minimiser of

\[ d(A, A') + d(A', B') + d(B', B'). \])

- modify edit distance

• look at tokens appearing in A and B, and run a classifier on them.

• increase the edit distance weighting for significant tokens

• dynamically update weights during the search, based on tokens in all \( \{ A_i \} \) and \( \{ B_i \} \).

• centre-of-mass classifier, or use libsvm in a modified version of Lean.
100: 16/1/25/22
5: 82/19/15
3: 65/18/13
2: 62/24/19

sum: 70/19/13