Aspects of heuristic proof search

Yves Jäckle

AI4Math Workshop
Hasso-Plattner-Institute
July 2026

Outline

Interactive Theorem Provers and Lean

Interactive theorem provers

An interactive theorem prover (ITP, or proof assistant) is software for writing machine checkable maths in a nice way.

- The user supplies the definitions, statements and proofs
- The machine rejects anything that does not typecheck
- Established systems: Lean, Rocq/Coq, Isabelle/HOL, Agda, ...

Lean 4

A proof assistant and a full programming language, built on dependent type theory.

- Lean supports metaprogramming : this is how tactics like `grow` are written
- Backed by a large mathematical library, Mathlib, and soon CSlib

A first look at Lean

The natural numbers

Lean by example

Let's define the natural numbers

```
axiom Nat : Type
-- there is a type of objects called *Nat*ural numbers
axiom Nat.zero : Nat
-- zero is a natural number
axiom Nat.succ (n : Nat) : Nat
-- given a natural number, there is a successor for it
```

(Normally `Nat` is an `inductive` type; here we axiomatize the primitives to keep them visible.)

`axiom` postulates a constant of a given type

Definitions name terms

```
def Nat.one : Nat := Nat.succ Nat.zero    -- defines 1
def Nat.two : Nat := Nat.succ Nat.one     -- defines 2
```

- `def` introduces a new name for a term of a stated type
- Lean has nice machinery to translate between litterals and this representation

Propositions as types: the $\leq$ relation

```
axiom Nat.leq (n m : Nat) : Prop
-- a relation between natural numbers:  n ≤ m

axiom Nat.leq_zero (n : Nat) : Nat.leq Nat.zero n
-- ∀ n, 0 ≤ n

axiom Nat.leq_step (n m : Nat) (h : Nat.leq n m) :
  Nat.leq (Nat.succ n) (Nat.succ m)
-- ∀ n m, n ≤ m → n+1 ≤ m+1
```

- `Prop` is the type of propositions
- We'll prove all we need from `leq_zero` and `leq_step`

Proofs as terms (1/3)

Goal: construct a term of type $1 \leq 2$.

```
def myFirstTheorem : Nat.leq Nat.one Nat.two := -- 1 ≤ 2
  sorry
```

- To *prove* a proposition is to construct a term of it as type
- The *statement* is the type
- The *proof* is the term
- `sorry` stands in for the missing term we'll fill out

Proofs as terms (2/3)

$1 \leq 2$ should follow from $0 \leq 1$ via `leq_step`.

```
def myFirstTheorem : Nat.leq Nat.one Nat.two :=
  Nat.leq_step    -- 1 ≤ 2 will follow from 0 ≤ 1
    Nat.zero      -- since 1 is 0+1
    Nat.one       -- and 2 is 1+1
    sorry         -- remaining argument = remaining goal:  0 ≤ 1
```

- Applying `leq_step` reduces the goal: it now suffices to prove subgoal $0 \leq 1$
  This is a backward step
- The two arguments $n := 0$, $m := 1$ determine the next goal
  The third arguments type *depends* on the previous arguments

Proofs as terms (3/3)

The last goal $0 \leq 1$ is exactly `leq_zero`.

```
def myFirstTheorem : Nat.leq Nat.one Nat.two :=
  Nat.leq_step Nat.zero Nat.one
    (Nat.leq_zero Nat.one)    -- proves 0 ≤ 1
```

- No `sorry` left: the term is complete and typechecks.
- Had we written `Nat.leq_zero Nat.two` instead, Lean would have thrown an error

Recursion

```
axiom Nat.recursion (n : Nat) (expr : Nat -> Sort u)
   (base : expr Nat.zero)
   (step : ∀ m, expr m -> expr (Nat.succ m)) : expr n
```

One principle, two readings:

- `expr : Nat → Prop` ⇒ induction (prove a statement for all n).
- `expr : Nat → Type` ⇒ recursion (define a function on n).

Recursion

```
def mySecondTheorem (n : Nat) : Nat.leq n (Nat.succ n) := -- n ≤ n+1
  Nat.recursion -- induction
    n (fun x => Nat.leq x (Nat.succ x))
   (base := Nat.leq_zero (Nat.succ Nat.zero)) -- 0 ≤ 1
   (step := -- ∀ n, n ≤ n+1 → n+1 ≤ (n+1)+1
     fun n ih => -- ∀ n : ℕ, ih : n ≤ n+1
      Nat.leq_step n (Nat.succ n) ih
   )
```

- Named arguments (ex: `(base := ...)`) for clarity
- Anonymous functions `fun`
- To prove ∀ display function that maps objects/hypotheses to a proof

Recursion

```
def mySecondTheorem (n : Nat) : Nat.leq n (Nat.succ n) := -- n ≤ n+1
  have fwd : Nat.leq Nat.zero (Nat.succ Nat.zero) := -- 0 ≤ 1
    Nat.leq_zero (Nat.succ Nat.zero)
  -- We now have 0 ≤ 1 in context
  Nat.recursion -- induction
   n (fun x => Nat.leq x (Nat.succ x))
   (base := fwd) -- matches subgoal
   (step := -- ∀ n, n ≤ n+1 → n+1 ≤ (n+1)+1
     fun n ih => -- ∀ n : ℕ, ih : n ≤ n+1
      Nat.leq_step n (Nat.succ n) ih
   )
```

- `have` adds fact proven from current hypotheses and facts
- This is a forward step

Equality and rewriting

Defining equality:

```
axiom Eq {T : Sort u} (a b : T) : Prop
-- the proposition  a = b

axiom Eq.refl {T : Sort u} (a : T) : Eq a a
-- reflexivity:  a = a

axiom Eq.rewrite {T : Sort u} (a b : T) (eq : Eq a b)
  (expr : T -> Sort v) (h : expr a) : expr b
-- from a = b and a expression containing a, get the same with b
```

- `{T : Sort u}` is an implicit argument (in braces): Lean infers it.
- `Eq.rewrite` is the substitution principle

Symmetry of equality (1/2)

Goal: from `eq : a = b`, produce a proof of $b = a$.

```
def Eq.symm {T : Sort u} (a b : T) (eq : Eq a b) : Eq b a :=
  sorry
```

Challenge: solve this faster then we move to the next slide

Symmetry of equality (2/2)

```
def Eq.symm {T : Sort u} (a b : T) (eq : Eq a b) : Eq b a :=
  -- b = a will follow from a = a after rewriting b into a
  Eq.rewrite a b eq
    (expr := fun x => Eq x a)  -- the expression
    (h := Eq.refl a)           -- supply expr a, i.e. a = a, via reflexivity
```

Proof search: the big picture

Backward search: many roads out of a goal

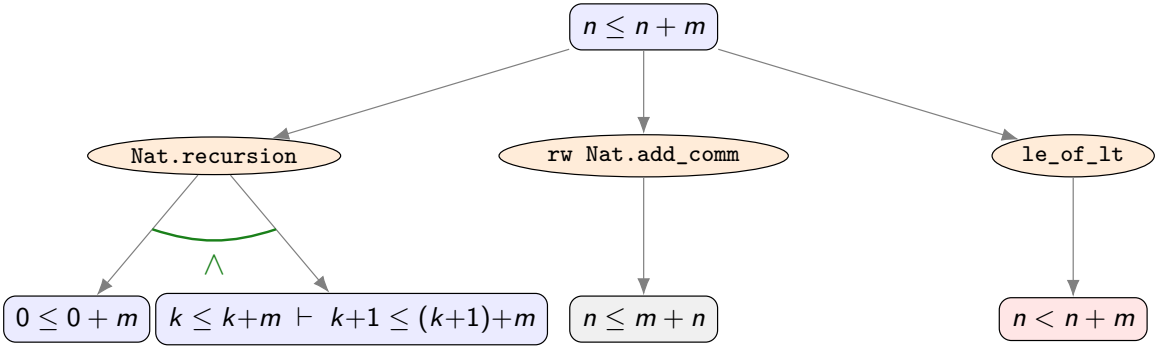
Recall a backward step turns a goal into (zero or more) subgoals.

Take the goal $n \leq n + m$. Several backward steps apply:

- `Nat.recursion` (induction on n): splits into a base $0 \leq 0 + m$ *and* a step $k \leq k+m \vdash k+1 \leq (k+1)+m$.
  This is a useful step.
- rewrite with `Nat.add_comm` ($a + b = b + a$): new goal $n \leq m + n$.
  Perfectly legal — but gets us nowhere (here).
- `le_of_lt` ($a < b \to a \leq b$): new goal $n < n + m$.
  A dead end — it is false when $m = 0$.

At every goal there are many choices, and some lead nowhere.

The search space is an AND–OR tree



- OR node (a *goal*): proven if *some* applicable step is proven
- AND node (a *step*): proven only if *all* the subgoals it produces are proven

Search tactics

`grow` explores this AND–OR tree, in the spirit of Aesop

```
theorem test (l L : List Nat) (h : (l ++ L).length = 42) :
  (l.length + L.length) = 42 := by
   grows
```

Our running examples live in the world of linked lists:

- `length` (number of entries)
- `dedup` (removing duplicates)
- sublists (<+) (same order, same multiplicity)

Example 1: close to the human proof

In our examples we restrict search to a small set of lemmas

```
def stdSanbox := #[‘List.dedup_cons_of_mem’, ‘List.length_append,
   ‘List.mem_dedup, ‘List.dedup_sublist, ‘List.dedup_subset,
   ‘List.dedup_idem, ‘List.length_erase_le, ‘List.Perm.dedup, ...]
grow_load_sandbox stdSanbox ; elimSandbox ; funIndSandbox
```

`grow` finds a term close to what a user would write:

```
theorem test : ∀ (l L : List ℕ),
    (l ++ L).length = 42 → l.length + L.length = 42 :=
fun l L h => (fun t => Eq.symm (Eq.symm length_append) ▷ t) h
```

Close to output of source language: `rw [← length_append]; exact h`

Example 2: a proof with a messy term

```
theorem test' (l : List Nat) : l.dedup <+ l := by
  grows          -- apply dedup_sublist
```

grow succeeds, but the term it finds is needlessly complicated:

```
theorem test' : ∀ (l : List ℕ), l.dedup <+ l :=
fun l =>
  let g.1 := dedup_sublist l;    -- the useful forward step
  let g.2 := dedup_subset l;     -- a useless forward step
  (fun t.0 t.1 => rec (motive := fun x =>
      x.dedup <+ x → x.dedup.Subset x → x.dedup <+ x)
    t.0 t.1 l g.1 g.2)
    (fun g.3 g.4 => g.3) (fun g.5 g.6 g.7 g.8 g.9 => g.8)
```

Example 2: a proof with a messy term

```
theorem test' : ∀ (l : List ℕ), l.dedup <+ l :=
fun l =>
  let g.1 := dedup_sublist l;    -- the useful forward step
  let g.2 := dedup_subset l;     -- a useless forward step
  (fun t.0 t.1 => rec (motive := fun x =>
       x.dedup <+ x → x.dedup.Subset x → x.dedup <+ x)
     t.0 t.1 l g.1 g.2)
     (fun g.3 g.4 => g.3) (fun g.5 g.6 g.7 g.8 g.9 => g.8)
```

- show result as forward step
- make useless forward step
- show `x.dedup <+ x → x.dedup.Subset x → x.dedup <+ x` by induction
- base and step are the hypothesis
- apply this to our forward steps to get result

Aspects of proof search

What are the aspects of proof search ?

- Target lean source or `Lean.Expr`
- Allowed steps and tools: forward steps, induction, rewriting, tactics
- Find relevant lemmata and relevant steps
- Beyond our scope: find relevant definitions and statements to study
- Make it bug free and make it scale

Rewriting

First aspect: the search must build *correct terms*

"Make it bug free"

- The whole point of Lean is that a proof is a term the kernel checks
- Search steps must assemble into a type-correct term

Goal: rewrite $y + x$ into $x + y$ underneath two binders

```
example : (fun x y : Nat => y + x) = Nat.add := by
   rw [Nat.add_comm]          -- FAILS: rw won't rewrite under fun

example : (fun x y : Nat => y + x) = Nat.add := by
   simp_rw [Nat.add_comm]     -- succeeds
   -- (fun x y : Nat => x + y) = Nat.add
   rfl
```

- `rw` refuses to apply under a binder.
- `simp_rw` allows it as part of the simplification

Pitfall: rewriting with a side condition

```
#check Nat.add_div_left
-- Nat.add_div_left (a : ℕ) {b : ℕ} (H : 0 < b) : (b + a) / b = a / b + 1

example (a : Nat) :
    (fun (b : Nat) (h : 1 < b+1) => (b + a) / b) = (fun _ _ => 42) := by
  rw [Nat.add_div_left]         -- FAILS

example (a : Nat) :
    (fun (b : Nat) (h : 1 < b+1) => (b + a) / b) = (fun _ _ => 42) := by
  simp_rw [Nat.add_div_left]    -- FAILS, even though it is @[simp]
```

- The lemma carries a side condition $0 < b$
- `grow` implements a solution to allow this rewrite

Doing it by hand: thread the side condition

The big idea:

```
example (a : Nat) :
    (fun (b : Nat) (h : 1 < b+1) => (b + a) / b) = (fun _ _ => 42) :=
  have inside (b : Nat) (h : 1 < b+1) (ng : 1 < b+1 → 0 < b) :
    (b + a) / b = a / b + 1 :=
      Nat.add_div_left a (ng h)
  have first_binder (b : Nat) (ng : 1 < b+1 → 0 < b) :
    (fun (h : 1 < b+1) => (b + a) / b) = (fun _ => a / b + 1) :=
      funext (fun h => inside b h ng)
  have second_binder (ng : ∀ b, 1 < b+1 → 0 < b) :
    (fun b (h : 1 < b+1) => (b + a) / b) = (fun b _ => a / b + 1) :=
      funext (fun b => first_binder b (ng b))
  by
  refine @Eq.ndrec _ _ (fun p => p = (fun (b : Nat) (h : 1 < b+1) => 42))
    ?newgoal_main _ (second_binder ?newgoal_cond).symm
  · dsimp
    sorry -- ⊢ (fun b x => a / b + 1) = fun b h => 42
  · sorry -- ⊢ ∀ (b : ℕ), 1 < b + 1 → 0 < b
```

Pitfall: rewriting inside a dependent type

Next issue:

```
def Fin.isNonZero (n : Nat) (x : Fin (n+1)) := x ≠ 0

example (n m : Nat) (x : Fin (n+m + 1)) :
    x.isNonZero (n+m) → ∃ y : Fin (m+n + 1), y.isNonZero (m+n) := by
  rw [Nat.add_comm]          -- FAILS

example (n m : Nat) (x : Fin (n+m + 1)) :
    x.isNonZero (n+m) → ∃ y : Fin (m+n + 1), y.isNonZero (m+n) := by
  simp_rw [Nat.add_comm]     -- FAILS
```

- Pattern `n+m` appears inside the dependent type `Fin (n+m+1)` – the type of `x`.
- Usual rewrite-term is not type correct : x's type is not rewritten

Doing it by hand: revert, rewrite, reintroduce

A solution is to revert:

```
example (n m : Nat) (x : Fin (n+m + 1)) :
    x.isNonZero (n+m) → ∃ y : Fin (m+n + 1), y.isNonZero (m+n) :=
  have reverted : ∀ (x : Fin (n+m + 1)),
      x.isNonZero (n+m) → ∃ y : Fin (m+n + 1), y.isNonZero (m+n) := by
    refine @Eq.ndrec _ _
      (fun p => ∀ (x : Fin (p + 1)),
        x.isNonZero p → ∃ y : Fin (m+n + 1), y.isNonZero (m+n))
      ?newgoal
      _ (Nat.add_comm m n)
    dsimp
    intro x
    -- ⊢ Fin.isNonZero (m + n) x → ∃ y, Fin.isNonZero (m + n) y
    intro H
    use x
  reverted x
```

Much more term building fun to be had!

Set-tries

Set-tries: taking forward steps efficiently

- A forward step applies a theorem whose hypotheses are all present in our local context.
- So we constantly ask: given the facts we currently have, which theorems have all of their hypotheses available?

Abstractly

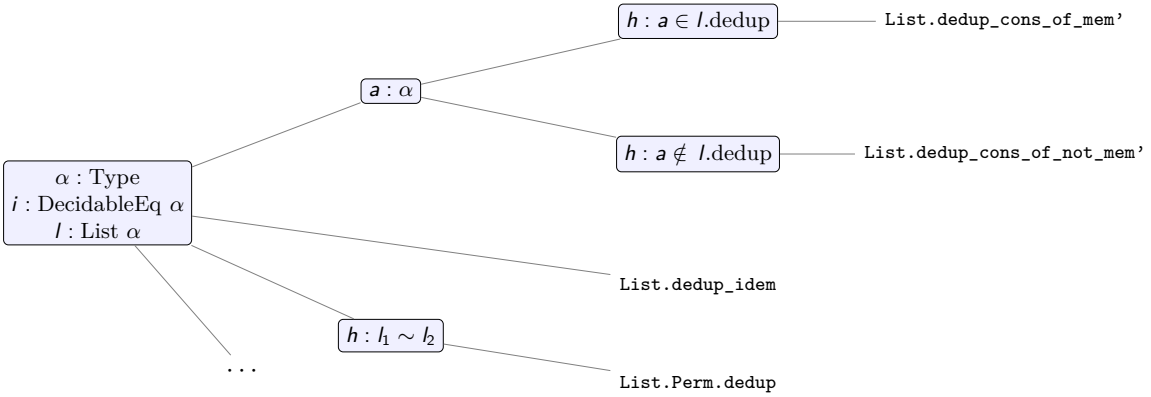
Give:

- family of target sets (each theorem's hypotheses)
- a query set (our current facts)

report every target set that is $\subseteq$ the query set.

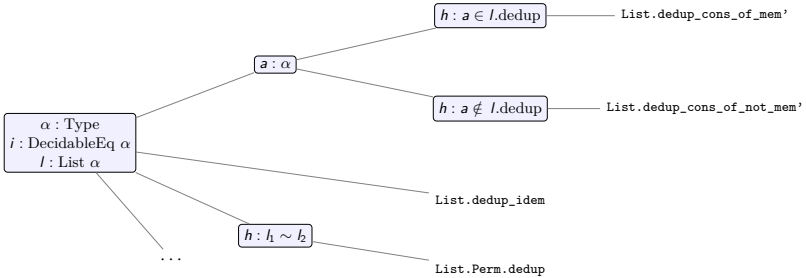
A set-trie is a data-structure to do this efficiently.

Sharing common hypotheses



A root-to-leaf path lists the hypotheses a theorem needs.

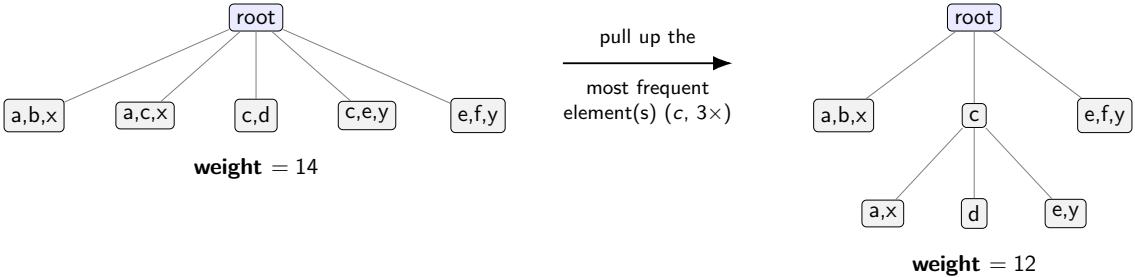
Sharing common hypotheses



- In `grow` we consider non-inferable hypotheses only (sinks)
- Note the term dependence, even among sinks
- Subsets of hypotheses (at set-trie nodes) can be queried efficiently via a path index
- Larger example in code

Building a set-trie: a greedy heuristic

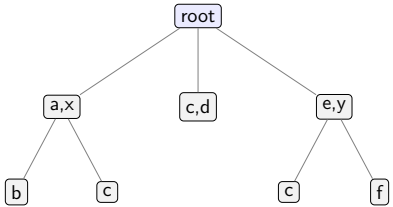
Goal: a trie whose total key size (summed over nodes) is small.
Family $\{\{a, b, x\}, \{a, c, x\}, \{c, d\}, \{c, e, y\}, \{e, f, y\}\}$.



Pick the element most common among siblings, make it a shared parent

. . . but greedy is not optimal

The same family also fits in a weight-10 trie — which greedy never finds:



weight $= 10$

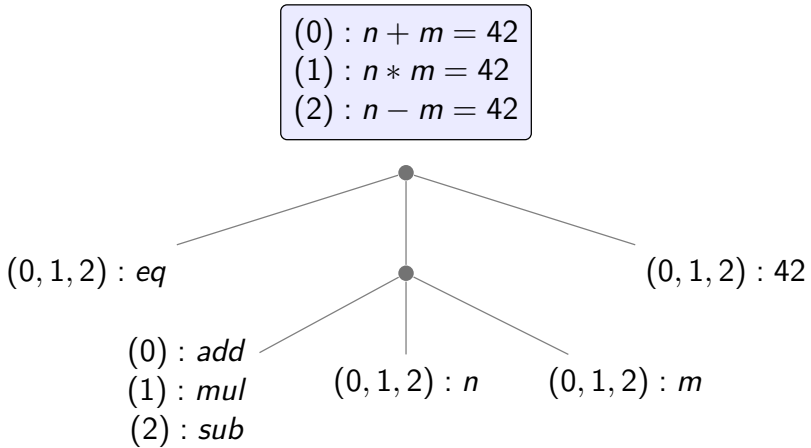
Greedy gives 12, the optimum here is 10 — and we know of no efficient algorithm for the optimum.

Path indexing

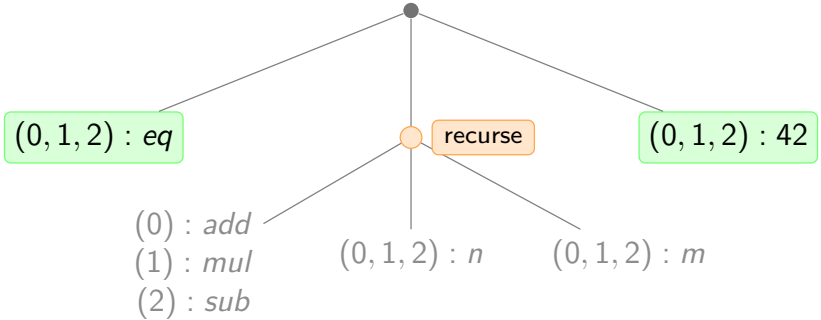
Path indexing: storing sets of expressions

- Like discrimination trees, a path index stores a set of syntax trees (`Expr`)
- Used for efficient membership queries
- Each stored expression gets an index; the tree mirrors the shape of `Expr`, and every node records which indices carry which symbol at that position.
- Will lead to scoring heuristic of the next section.

One index for three statements

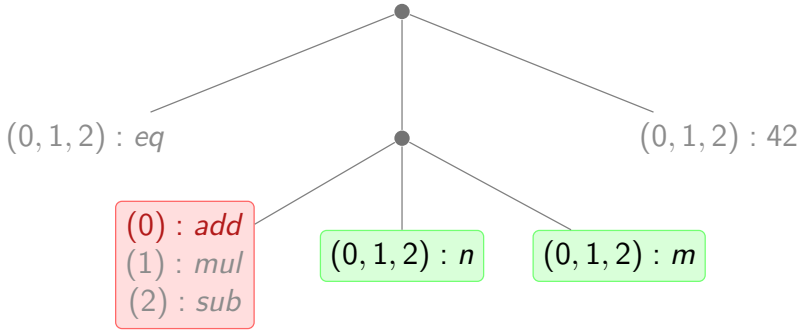


Querying for membership: $n + m = 42$ (depth 1)



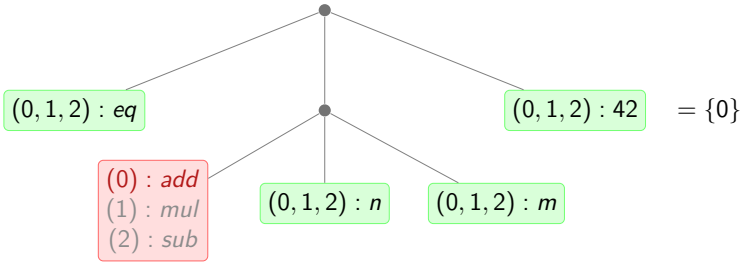
Split $n + m = 42 \equiv eq\,(n + m)\,42$. The head eq and the literal 42 each match all indices, $\{0, 1, 2\}$; the middle argument $n + m$ needs a recursion.

Querying for membership: $n + m = 42$ (depth 2)



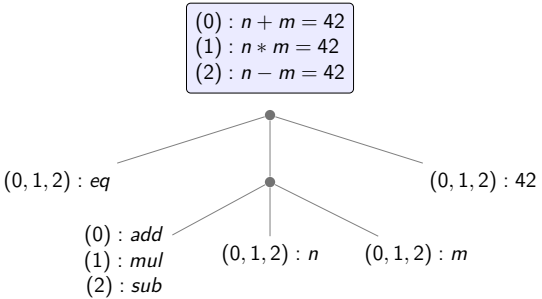
In the recursion, n and m both match $\{0, 1, 2\}$, but the head *add* matches only $\{0\}$.
Intersecting: $\{0, 1, 2\} \cap \{0, 1, 2\} \cap \{0\} = \{0\}$.

Querying for membership: $n + m = 42$ (result)



Back at the root: $\underbrace{\{0,1,2\}}_{eq} \cap \underbrace{\{0\}}_{n+m} \cap \underbrace{\{0,1,2\}}_{42} = \{0\}$ — so $n + m = 42$ is stored, at index 0.

Aspects



- Matching is done up to reduction ($2 + 2$ should match 4):
  we handle this with normal forms and partial builds.
- And up to unification: nodes may carry metavariables, so a query can retrieve every stored expression it *unifies* with, not only the syntactically identical one.
  Ex.: if n is a metavariable, match $(a * b) + m = 42$ with assignment
  These assignements must be tracked/propagated !

The search heuristic

A memorisation heuristic for steps

Task

During the search: is applying this theorem a good next step?

- From a library of proofs: every time a theorem is applied, record the proof state — its goal and the local hypotheses present.
- Memorisation. A theorem is a promising candidate when, in `grow`'s current state,
  - the search goal *matches* a recorded goal, and
  - the search context *contains* the recorded hypotheses.
- The recorded goals/context get generalised with a path-index procedure — *covered later*.

So the first task is to sample proof states from proofs.

Sampling proof states

We replay a proof and, at each theorem application, emit a sample.

```
theorem testProof (l L : List Nat) :
    l.dedup.dedup.length + L.length ≤ (l ++ L).length := by
  rw [dedup_idem, length_append]
  apply Nat.add_le_add_right
  apply Sublist.length_le
  apply dedup_sublist
```

A sample records the goal and the hypotheses it relied on, and the theorem applied.

A few samples from that proof

```
Goal:  l.dedup.length ≤ l.length
Hyps:  (none)
Kind:  thm List.Sublist.length_le
```

Directly from proof:

```
theorem testProof (l L : List Nat) :
   l.dedup.dedup.length + L.length ≤ (l ++ L).length := by
  rw [dedup_idem, length_append]
  apply Nat.add_le_add_right
  -- sampled here
  apply Sublist.length_le
  apply dedup_sublist
```

A few samples from that proof

```
Goal:  l.dedup.length + L.length ≤ l.length + L.length
Hyps:  l.dedup <+ l
Kind:  thm Nat.add_le_add_right
```

One proof terms, many derivations. Sample ficticious foward step:

```
theorem testProof (l L : List Nat) :
   l.dedup.dedup.length + L.length ≤ (l ++ L).length := by
  rw [dedup_idem, length_append]
  have ficticous := dedup_sublist l
  -- ficticious foward step
  -- sample here
  apply Nat.add_le_add_right
  apply Sublist.length_le
  apply ficticous
```

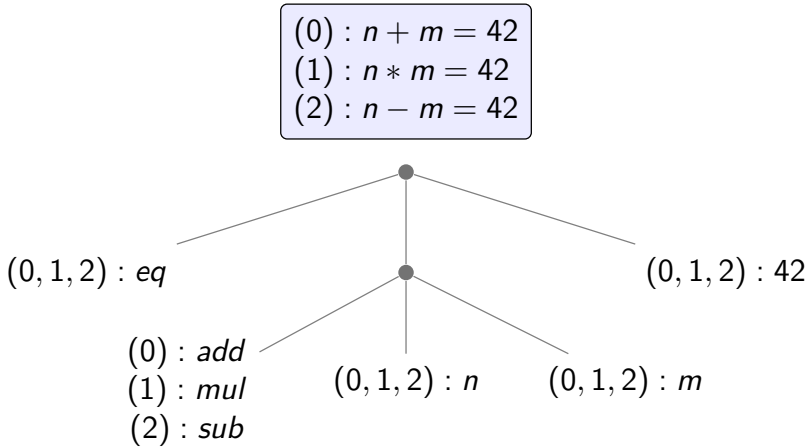
From samples to a pattern: generalisation

We can do better then memorisation.
For each theorem, we merge all its recorded goals into one path index, then generalise it:

- Leave frequent sub-expressions as they are.
- For infrequent ones, infer their type and look for a frequent type.
- Replace an infrequent expression by a metavariable of that type.

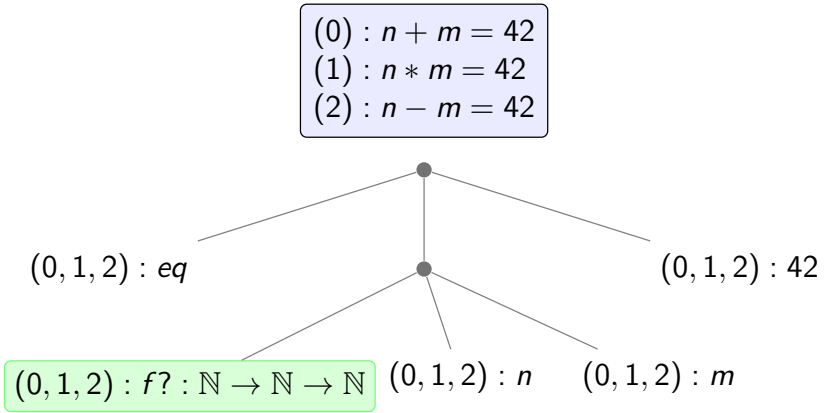
A search goal then counts as *resembling* the samples when it unifies with this pattern

Reminder: the recorded goals, as one path index



Here, we call a pattern infrequent when it occurs in $\leq \frac{1}{3}$ of its siblings.

Generalising the index



Each of *add*, *mul*, *sub* occurs in just $\frac{1}{3}$ of the siblings — infrequent — so they collapse to a metavariable $f? : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. $n/m = 42$ will now match !

Generalisation: open aspects

- Metavariable garbage collection: Generalisation can create metavariables that later get generalised further and are no longer referenced by the index — these can be removed.
- Metavariable reuse: We want as few metavariables as possible for efficiency (keep path indices small): no need to have two $(m? : \mathbb{N})$ if no goals share them.
  This is delicate and still in a research state.
- On real data. Running the procedure over a larger body of samples:

```
-- 6_Heuristic.lean, line 56
#eval explore_gen_thm_core_S
  `LeanGrow.Src.Caching.Score.Test.DummySamples
```

. . . the output is, frankly, quite hard to see through.

Closing words

Closing words

- Heuristic proof search can't compete with LLMs
  ... for now. But maybe one day ?
- Currently a statistical/phenomenological approach: take theorem as search step if goal and context match previously seen frequent applications.
  Move towards learning: determine goal path index and context set-trie that *minimise* search step count or proof term size for a given set of target queries. How ?
- Work in progress: rewrite with dependent constants ? Link induction reverts to samples ? Optimal set-tries and fuzzy queries ? Sample tactic applications ? Move to Aesop. Fix bugs. Optimize. Fix more bugs.

Thank you for your attention!

Questions?