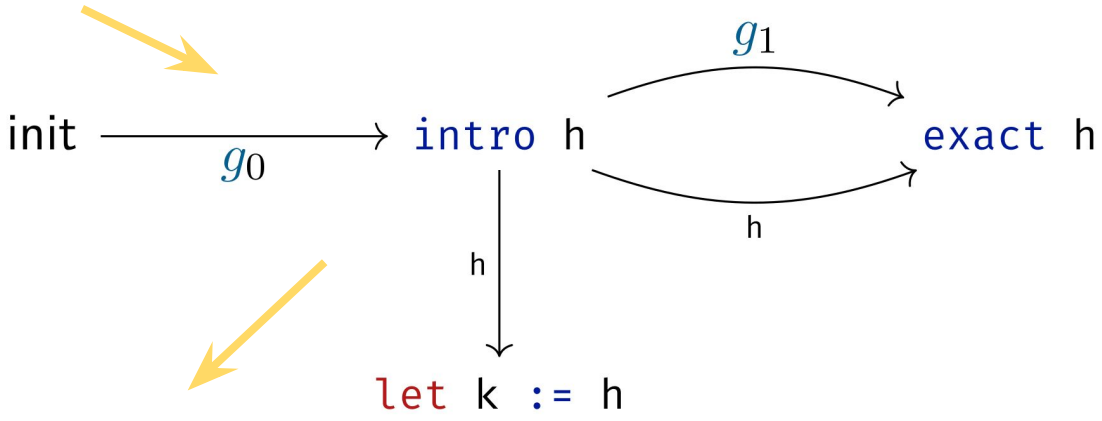
Tacflow
Inferring *Tactic Dependency Graphs* in Lean 4

1. Infer ***signatures*** for tactics
   capture logical inputs and outputs
2. Construct ***dependency graphs*** for tactic proofs
   capture logical dependencies between tactics

```
theorem id' (P : Prop) : P → P := by
  intro h
  let k := h
  exact h
```




```
theorem let_dead2 : P → (P → P → Q) → P := by
    intro h f
W   let k := f h
W   let k := k h
    exact h
```

```
Diagnostics:
Dead tactic can be removed:
  let k := k h

Note: This linter can be disabled with `set_option linter.
tacflow.dead false`
```

Basis for semantic analyses and tools on tactic proofs:

dead tactic analysis
parallel elaboration
lemma extraction
proof slicing
proof refactoring

Tacflow: Inferring Tactic Dependency Graphs in Lean 4

1 Introduction

Interactive theorem provers allow users to write machine-checked proofs for mathematical propositions. In provers based on dependent type theory, such as Rocq, Agda, and Lean, a proposition is expressed as as type, and the task of proof is to develop a term of that type. For instance, consider the claim that any proposition P implies itself, *i.e.*, $P \to P$. One can formulate the claim as a type and prove the claim by constructing a term in the following way:

```
theorem id (P : Prop) : P → P :=
  fun h => h
```

However, as it is often cumbersome to write such proof terms directly, many interactive theorem provers, including Lean, allow proofs to be written using *tactics*, which help to generate proof terms. The following alternative proof is given using the `intro` and `exact` tactics.

```
theorem id (P : Prop) : P → P := by
  intro h
  exact h
```

Intuitively, `intro h` observes that, to prove `P → P`, it is sufficient to prove `P` given a hypothesis `h` witnessing `P`. Then, `exact h` proves `P` by using `h`. More concretely, the *tactic script* is executed during elaboration, wherein each invocation *partially* constructs the proof term by reading and modifying a *proof state* containing *goals* and available *hypotheses*. The initial goal of the above proof is to construct a term of type `P → P`. First, `intro h` constructs a term proving `P → P`. This term is partial, with hole requiring a term `P`, which is the new goal. The tactic also makes a new hypothesis `h` available for use. Then, `exact h` closes the new goal by using `h`. This fills the hole and completes the proof term.

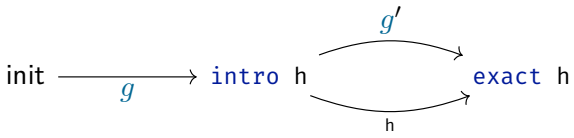
Though this proof is simple, tactics facilitate abstraction and automation in much larger and more complex proofs. Xin et al. (2025) observe the usefulness of *tactic dependency graphs* (TDGs), which abstract over syntactic details to capture the logical dependencies between tactic invocations in proof scripts. In particular, they develop a technique to automatically discover custom high-level tactics (compositions of existing tactics) by analysing semantic usage patterns across a proof corpus. More broadly, TDGs are a useful foundation upon which to build analyses to improve the quality, modularity, and performance of large proof corpuses; given such dependency information one could conceivably discover and extract lemmas from proofs, build proof slicers, and parallelise or incrementalise tactic execution.

Tactic signatures The signature S of a tactic invocation describes the formal inputs and outputs. These are lists of *objects*, which are *goals* or *hypotheses*.

$$\begin{array}{llll} \text{Goal} & g & \text{Obj} & o \quad ::= \quad g \mid h \\ \text{Hyp} & h & \text{Sig} & S \quad ::= \quad o^* \to o^* \end{array}$$

For instance, the `intro` tactic reads the main goal, and, if it is of the shape `P → Q`, produces a new hypothesis of type `P`, and replaces the goal with a new one of type `Q`. In the above proof, the invocation `intro h` therefore takes in the main goal g and outputs a hypothesis `h` and a new goal g', yielding the signature $g \to g' \times \mathtt{h}$. Similarly, the invocation `exact h` takes the goal g' and the hypothesis `h` and outputs no new goals or hypotheses $g' \times \mathtt{h} \to \langle\rangle$.

Tactic dependency graphs Given a collection of signatures, one can construct a TDG. These are directed acyclic graphs of tactic invocations, with edges along objects marking dependency. The above proof yields the following TDG:



Tacflow In this work, we implement Tacflow, a framework for tactic dependency graphs in Lean 4. The framework automatically infers tactic invocation signatures, which capture the logical inputs and outputs of an invocation, and constructs TDGs (§2). We demonstrate a dead tactic linter atop the TDGs and evaluate its correctness on a small suite of example proofs (§3).

2 Implementation

Our prototype implements inference for tactic signatures based on dynamic analysis (§2.1) and construct dependency graphs (§2.2).

2.1 Signature inference

To infer the signature of a tactic invocation, one might proceed in one of several different ways. As tactics are implemented in ordinary Lean code using meta-programming APIs, one could perhaps analyse the source code of each tactic. Unfortunately, tactic code can be arbitrarily complex; whilst possible, such a static analysis would be a tall task.

In general, a tactic may have arbitrary effects, including I/O. Typical tactics are, however, relatively well-behaved in reading and modifying the proof state. For instance, tactics such as `intro`, `apply`, `simp`, or `induction` replace particular goals. They may additionally use hypotheses from the local context. Others, such as `let`, `have`, `obtain`, or `clear` introduce new hypotheses to or remove old ones from the local context. Even complex, high-powered tactics, which may call external solvers and tools, restrict inputs to objects in the proof state. In this way, it is reasonable to treat tactics as opaque functions over proof states.

Even so, a tactic may read and mutate the proof state in any number of ways, and it is impossible to track all of these effects without instrumenting primitive meta-programming operators. Our approach deals in lightweight approximations; it simply examines the proof states before and after the tactic's execution. Luckily, it is possible to extract these states from the elaboration process.

The inference engine builds upon `mathlib4`'s tactic analysis framework, which is used to implement linters for tactic scripts. The framework extracts tactic sequences from a proof, equipping each node in the sequence with the list with the *incomplete goals* and *meta-variable context* before and after the invocation.

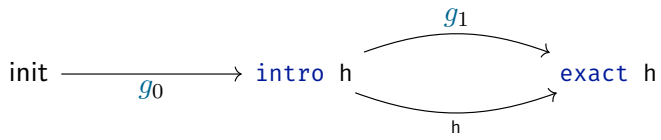
Diffing proof states Inference proceeds by computing a semantic difference between the pre- and post-invocation proof states. In particular, the algorithm compares the goal and hypotheses sets to determine (1) goals were removed, (2) goals were added, (3) hypotheses were removed, and (4) hypotheses were added. The removed objects correspond to the inputs, and the added objects to the outputs. Equality is, for goals, by the ids of the corresponding meta-variables (`MVarId`) and, for hypotheses, by the ids of the corresponding free variables (`FVarId`).

Goal replacement Unfortunately, this basic approach requires some modification to work well with Lean's metaprogramming paradigm. Consider, for instance an additional (spurious) `let`-binding in the proof above:

```
theorem id' (P : Prop) : P → P := by
  intro h
  let k := h
  exact h
```

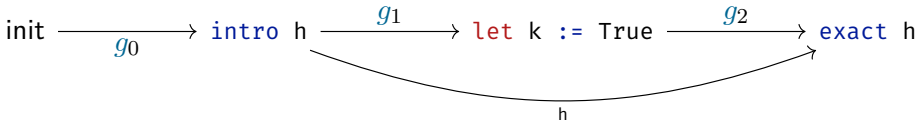
Intuitively, the `let` invocation simply adds a new hypothesis `k`, leaving the goal unmodified. Indeed, the type of the goal does not change; it is still `P` afterwards. Because, however, tactics progressively construct a proof term, the goal is replaced with a new one (with a different meta-variable). In fact, practically every tactic replaces the goal, even if it is left untouched from the user's perspective.

Whereas one would expect the following TDG for the above proof,



`let k := True`

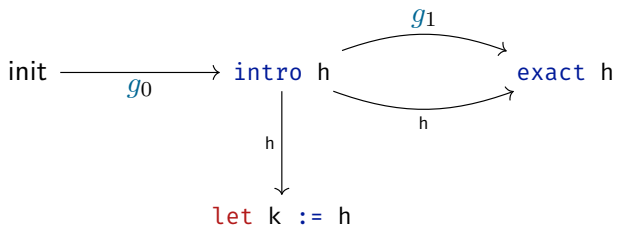
the basic approach would introduce edges to and from `let k := True`.



g_2 is introduced solely for the purposes of constructing the proof term. It is functionally identical to g_1; indeed, it possesses the same type and user-facing name. Of course, it is possible for a tactic to introduce a new goal of the same type and same user-facing name that is not semantically a "replacement," but this seems unlikely.

Thus, the augmented algorithm exploits these characteristics to search for potential *replacements* for the "removed" goals. Precisely, a goal is *replaced* (rather than removed) if there is an *replacement* goal among the "added" goals with (1) a definitionally equal type and (2) an identical user-facing name. In our proof of `id'`, the `let k := True` invocation replaces g_1 with g_2, and this is made explicit in the signature: $\{g_1 \Rightarrow g_2\}\ \langle\rangle \to \texttt{k}$.

Hypothesis usage Suppose that the `let`-binding was instead `let k := h`, introducing a dependency on `intro h`:



The algorithm is further augmented to approximate such read hypotheses by collecting the existing hypotheses which appear in the terms assigned to new hypotheses. In this case, after `let k := h`,

the new hypothesis `k` is assigned to the term `h`, which contains the free variable `h`. These reads are also made explicit in the signature: $[\texttt{h}]\,\{g_1 \Rightarrow g_2\}\,\langle\rangle \to \texttt{k}$.

Unfortunately this heuristic is insufficient for tactics which hide the assignments. For instance, `have k := h` behaves as `let k := h; clear_value k`, and `clear_value` makes the assigned term inaccessible. Such tactics must be handled by additional heuristics, which we discuss in §4.

2.2 Graph construction

Given invocation signatures, TDG construction proceeds straightforwardly by introducing edges based on inputs (reads and removals) and outputs (additions). Replacements are handled carefully by treating the "original" version as canonical.

3 Evaluation

We demonstrate tactic dependency graphs and the inference algorithm by developing a linter for dead tactic invocations.

3.1 Case study: dead tactic linter

We implement a simple dead tactic linter atop tactic dependency graphs (§3.1). The linter plugs into Lean's built-in linting APIs and can be enabled with the following preamble:

```
import Tacflow.Linters
set_option linter.tacflow.dead true
```

Deadness A *truly dead* tactic invocation is one which may be removed from a correct proof without causing the proof to fail.

Analysis A *dead* tactic invocation is which is not backwards-reachable from any *terminal*. A *terminal* is an invocation closes one or more goal, *i.e.*, removes one or more goals without adding any. For instance, the invocation `exact h` in the above proof of `id'` is terminal. The invocations `let k := True` and `let k := h` are dead because they are not reachable from the terminals; their sole output, the hypothesis `k`, does not flow to any terminals.

The analysis is a backwards data-flow analysis on TDGs, beginning from the terminal invocations, which accumulates the set of dead tactic invocations.

Setup The analysis was evaluated on a small suite of 18 basic proofs using common tactics. Some proofs contained truly dead invocations, marked `-- DEAD`. Of 39 non-terminal invocations, 8 invocations were truly dead.

Results The linter correctly marked all 8 dead invocations as such, but also erroneously identified 5 additional invocations as dead, indicating that the inference algorithm under-approximates dependencies for these common tactics. Of the 5 false positives:

- 3 were invocations of `expose_names`, which gives names to anonymous hypotheses. Because the tactic does not modify the ids of these hypotheses and only changes their user-facing names, the algorithm assumes that the signature is $[]\,\{\}\,\langle\rangle \to \langle\rangle$. However, these invocations were necessary for later usages of the hypotheses.

- 1 was an invocation of `let` whose output hypothesis was used by `have`. It was considered dead because the algorithm could not detect the usage as discussed in §2.1.
- 1 seems to reflect a bug in TACFLOW's implementation.

4 Future Work

Tactic dependency graphs enable a range of semantic analyses. There is also room to explore and evaluate different kinds of inference heuristics.

Inference heuristics The present implementation of TACFLOW relies on `mathlib4`'s tactic analysis framework, which extracts *sequences* of tactics from Lean's infotrees. A more general framework would allow inference and analysis across nested sequences for more fine-grained TDGs.

§3 highlighted certain deficiencies in TACFLOW's inference heuristics, such as the inability to detect changes in hypothesis visibility. A more comprehensive evaluation of possible heuristics and their efficacy on real-world tactics is necessary.

Applications We envision that TACFLOW could form the basis for many tactic-based proof engineering tools. TDGs can be used to develop program/proof slicers, parallel elaborators, automated refactorings, and lemma extractors.

5 Artifact

TACFLOW is available at https://github.com/mirryi/tacflow.

References

Yutong Xin, Jimmy Xin, Gabriel Poesia, Noah Goodman, Qiaochu Chen, and Isil Dillig. 2025. Automated Discovery of Tactic Libraries for Interactive Theorem Proving. (2025). arXiv:2503.24036 [cs.PL] https://arxiv.org/abs/2503.24036