Beyond the Tactic-State Automaton

Daniel Selsam

Microsoft Research

Overview

Most prior work applying machine learning to higher-order theorem proving has adopted the *tactic-state automata* idiom in which the ML agent maps tactic-states to tactic-state transformations (*i.e.* tactics). This approach is appealing in its simplicity but suffers many limitations. We propose instead defining search spaces using *nondeterministic tactics* and discuss several ways of connecting them to ML oracles.

Tactics

A *tactic-state* is a list of *goals*, where each goal is a sequent, *i.e.* a list of hypotheses and a goal to be proven. A *tactic* is an arbitrary (functional) program that can read and write to a tactic-state, and that may also fail. Like regular programs, tactics can call each other, take each other as functions, develop their own internal datastructures, return values of arbitrary types, and in general can perform a lot of work that is not made visible as modifications to the tactic-state. Tactic frameworks are designed to support two very different use-cases: *interactive* mode, in which users execute pre-written tactics to observe their effect on the tactic-state, and *automation* mode, in which users implement (potentially sophisticated) tactics that may be used later in interactive mode or be run on goals in an off-line manner.

Tactic-State Automata

In the *tactic-state automata* idiom, an ML agent tries to prove theorems in interactive mode, *i.e.* by mapping tactic-states to tactics which are then executed to return new tactic-states. There happens to exist many pre-written tactics suitable for interactive mode, and the tactic-state automata idiom may be sufficient to learn how to mix, match, and instantiate these pre-written tactics in better ways. However, this approach is fundamentally limited by the set of atomic tactics it happens to have access to. Humans write new tactics all the time that are not merely sequences of existing tactics, and these tactics cannot be expressed in the tactic-state automata idiom. It can also be challenging to write these tactics without the help of ML, since they often require heuristics.

Nondeterministic Tactics

We propose an alternative to tactic-state automata: *nondeterministic tactics*. With tactic-state automata, the ML agent sits *above* the tactics and selects tactics to execute. In contrast, nondeterministic tactics sit above the ML and query the ML for heuristic guidance at nondeterministic choicepoints. The ML's job is to *execute* these nondeterministic tactics by deciding at runtime how to instantiate the nondeterminism.

Choice

We achieve nondeterminism by adding a new primitive, *choose*, to a tactic language that selects from a finite set of candidates of arbitrary type. There are countless variations of *choose*; for example, we can add a second primitive *choose-string* that samples a string from a language model and then tries to parse it as a tactic. See the accompanying paper for a general way of adding nondeterminism to any monadic computation.

Examples

Tactic-state automaton:

```
def tacticStateAutomata : Tactic Unit :=
  while goalsRemaining do
    let tac <- choose tactics
    tac
```

Solving inequalities by contorting subterms to match known theorems:

```
def simpleInequalitySolver : Tactic Unit :=
  while goalsRemaining do
    let thm <- choose standardDozen
    let t   <- choose subtermsOfGoal
    makeLookLike t (lhs thm)
    rewrite thm at t
    simp
```

Solving geometry problems by adding auxiliary points that empirically satisfy desired properties:

```
def simpleGeoSolver : Tactic Unit := do
  let diagram <- buildDiagram
  while goalsRemaining do
    if goalHasVariables then
      let points <- chooseFromDiagram diagram
      instantiate points
    let thm <- choose geoTheorems
    apply thm
```

Machine Learning

Like tactic-state automata, nondeterministic tactics generally induce extremely challenging search spaces. We believe that ML is a promising approach for searching these spaces more efficiently.

Challenge: What to show ML?

A nice feature of tactic-state automata is that an embedding of the tactic-state datastructure alone can serve as a precise prompt for an ML oracle. Unfortunately, the situation is more complicated for nondeterministic tactics, since choicepoints may have relevant runtime state and differing downstream computations. For example, consider the simple inequality solver. When choosing the subterm `t` of the goal, it is necessary to consider, in addition to the goal itself, which `thm` was selected beforehand and what the downstream code plans to do with `t` and `thm`. We now survey a few mutually compatible options for how to condition an ML oracle.

Explicit Prompts

The *choose* primitive may be extended to take a piece of data representing a user-specified *prompt* representing the relevant information about this particular choicepoint. This is the approach taken for tactic-state automata, and as discussed above, in that case this prompt suffices since the past is irrelevant and the downstream computations are the same for all choicepoints. It may not be clear how to design an adequate prompt in general. There is a large design space: a prompt could even be in natural language.

Choice Summaries

The *choose* primitive may also be extended to take a list of pairs, where the second element of the pair is a summary embedding of the choice in question. For example, when choosing from a list of lemmas, the names and types of the lemmas may serve as the summaries that are passed to the ML oracle. For other choicepoints it may not be obvious how to summarize the choices, *e.g.* when choosing among other nondeterministic tactics.

Pseudo Environments

The tactic-state could be extended to include its own (pseudo) environment mapping identifiers to stacks of arbitrary (embeddable) datatypes, for the purpose of enriching the explicit prompts. The bookkeeping could be hidden as much as possible by syntactic sugar. The `let` construct could be sugar for first pushing the value to the pseudo environment, and then popping it when the variable goes out of scope. Every function call that may make a nondeterministic choice could first create a new local environment and then restore the old one upon exiting.

Metaprogramming

The last approach we consider seems the most principled on paper: use metaprogramming to directly inspect and embed all relevant information for every choice automatically. Specifically, the ML-friendly prompt for a choice can be an embedding of the downstream code yet to execute along with the subset of the current environment that will be inspected downstream. The feasibility of this approach depends heavily on the details of the language being extended. In Python, the `inspect` and `dis` modules make it relatively straightforward to construct a lossless encoding of a given choicepoint, by traversing the bytecode for each choice at runtime and collecting the used symbols and their values in the process. The accompanying paper describes how we achieved direct choice-inspection in Lean (version 4), and also reports on the roadblocks we hit during the process. The summary is that the generic metaprogramming approach cannot be made practical without runtime type information, because without undue care, most choices will contain large amounts of data that are not worth embedding.

Discussion

Ultimately, we see no silver bullet for guiding arbitrary nondeterministic tactics in practice. We also do not see how a tactic-state automaton could employ known techniques such as building geometry diagrams and inspecting them to make conjectures. We still consider it an open problem how to achieve the best of both worlds, expert strategies and ML.