BEYOND THE TACTIC-STATE AUTOMATON

Daniel Selsam
Microsoft Research
Redmond, WA, USA
daselsam@microsoft.com

ABSTRACT

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 introduce a new way to define sophisticated search spaces and discuss several ways of connecting them to ML oracles.

1 INTRODUCTION

Although the details differ among systems, generally speaking 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. The goals within a tactic-state are often independent but may be coupled by shared *metavariables*, *i.e.* fixed, not-yet-determined values that appear in multiple goals. 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.

In the *tactic-state automata* idiom, an ML agent maps tactic-states to tactics. In light of the generality of tactics described above, it is worth considering why this approach may work at all. The answer is that 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. 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 idiom does not provide assistance in automation mode.

One may argue that the tactic-state automata idiom suffers no practical bound on its power since most proofs in most formal mathematics libraries have relatively short representations in terms of a relatively small number of tactic primitives. However, almost every proof in every formal mathematics library was already known informally to the formalizer. Even if the pre-written tactics are sufficient to *express* proofs of the theorems in question, this does not imply that tree-search in the action space induced by these primitives is a good way of *solving* challenging new problems, *e.g.* problems arising in the IMO Grand Challenge. Even if this inference would hold in the infinite-data limit, it seems particularly unjustified in practice given the severe dearth of available training data.

One natural question is: what are humans taught explicitly that tactic-state automata ML are forced to induce? While humans see only a meager number of proofs, they are explicitly taught many high-level problem solving strategies. Manifesting these strategies ostensibly requires writing novel, sophisticated (automation-mode) tactics.[1] However, the strategies taught for *e.g.* solving olympiad problems are in general so high-level, so ill-constrained, that a natural encoding of them will necessarily be littered with heuristic choices to make. Ideally, machine learning could provide good heuristics *within* these tactics, but this requires moving beyond the tactic-state automata idiom. The rest of this paper considers how this might be achieved.

[1] One could also co-train language models on descriptions of these strategies, though the plausibility of this approach has not been established.

2 THE SEARCH TRANSFORMER

We now present our main abstraction, which we call *the search transformer*, in its simplest form. Our presentation will reference many concepts from functional programming (*e.g.* monad transformers) though we will try to explain the relevant parts of each concept we reference.

The *search transformer* is built on the following three mutually inductive types:[2]

```
inductive SearchT (m : Type → Type) (α : Type) : Type
| mk : m (Status m a)

inductive Status (m : Type → Type) (α : Type) : Type
| done        : α → Status m a
| choicepoint : ChoicePoint m a → Status m a

structure ChoicePoint (m : Type → Type) (α : Type) := {
  choices : List (SearchT m a)
}
```

Here `m : Type → Type` is expected to be a *monad*, which for our present purposes can be interpreted as follows: `m` describes some set of effects such that for all types α, the type `m α : Type` represents programs that return elements of type α but that may in addition perform any of the effects allowed by `m`. An example is the *tactic monad*, `TacticM`, which (as discussed above) allows reading and writing to a tactic-state object. Thus an element of type `SearchT m α` is a computation in `m` that either returns an element of α as usual (via `done`) or else returns a list of `SearchT m α` values (via `choicepoint`) representing a set of possible futures to choose among. We define `choice xs` to mean `SearchT.mk (pure (choicepoint (ChoicePoint.mk xs)))`, where `pure x` is the `m α` computation that performs no effects and simply returns `x`.

When `m` is indeed a monad, `SearchT m α` is a monad as well. For our present purposes, the important implication is that we can implement `SearchT` programs using the convenient *do*-notation pioneered by Haskell Jones (1995) and since adopted by other languages including Lean. For example, we can write a program that nondeterministically chooses two bools and returns the pair in a natural way:

```
def twoBoolsDo : SearchT m (Bool × Bool) := do
  let b₁ ← choice [false, true]
  let b₂ ← choice [false, true]
  pure (b₁, b₂)
```

Extensions. There are countless ways to extend the simple `SearchT` presented above. There could also be a primitive for choosing unordered subsets of a set as in Bavishi et al. (2019) or to support principled decomposition into subgoals (*e.g.* by *tabling* as in Selsam et al. (2020b)). There could also be a primitive for generating a string, `generate : String → (String → SearchT m α) → Status m α`, where the string is to be generated by a language model and then parsed in the downstream computation.

Search. A defining feature of the search transformer is that the search space induced by a `SearchT` program is abstracted away from the strategies that one may use to search the space. Thus search spaces and search strategies can be implemented separately. For example, here is pseudocode[3] for a generic depth-first search:

```
def dfs (ψ : SearchT m α) : m (Option α) := do
  let mut todo := #[ψ]
  while todo do
    let status ← todo.back
    todo := todo.pop
```

[2] We use the syntax of the Lean Theorem Prover de Moura et al. (2015) throughout, though due to idiosyncracies in Lean's metatheory, some additional indirection is required to define `SearchT` which we omit in this presentation.

[3] This pseudocode is simplified only slightly from working code.

```
    match status with
    | done x          => return (some x)
    | choicepoint cp => todo := todo ++ cp.choices
  return none
```

We first initialize a (mutable) `todo` stack with ψ, and as long as the stack is nonempty, we pop an element from it, run it, and either return the result (if it returns `done`) or add the new choices to the stack (if it returns `choicepoint`). Note that this snippet assumes that any branch-specific state is made explicit, *e.g.* by a `StateT` transformation *above* `SearchT`. We can relax this requirement by allowing `m` to provide `save` and `restore` methods for the part of its state that is branch-specific and having `dfs` call them at the appropriate times.

Other non-heuristic strategies are equally straightforward to implement, *e.g.* random search, breadth-first search, and iterative deepening. To implement heuristic search (*e.g.* MCTS) we need a function that maps `ChoicePoint`s to either policy scores, value estimates, or both:

```
structure Guess { policy : Vector Float, value : Float }
structure Oracle m a := { ChoicePoint m a → Guess }
```

The next section discusses several ways that machine learning could provide such an oracle.

3 Machine Learning

The search transformer as presented in Section 2 provides almost no information for a heuristic to go on. Specifically, when a search procedure stumbles on a new `choicepoint cp`, it has no way to even distinguish the choices, except based on trivialities like their positions in the list. This is because a `ChoicePoint` is simply a list of `SearchT m` α objects, and such objects have no inspectable structure. We now discuss several possible sources of signal for a learned heuristic.

Explicit prompts. The `ChoicePoint` type can be extended to take a value of some type σ; this argument could represent a user-specified *prompt* describing the choicepoint and so allow learning a value function $\sigma \to \mathbb{R}$. Note that we used the word *prompt* rather than *observation* because it does not in general suffice to describe only the current state of the user's datastructures; there may be relevant information that only exists implicitly on the stack or in the code defining the choices available at the current choicepoint. It may not be clear how a programmer should create such a prompt in general.

Explicit choice summaries. The `ChoicePoint` type can be extended further to take, for each candidate choice, a corresponding element of type γ that represents some *summary* of the choice and so allow learning a policy function $\sigma \to \gamma^k \to \mathbb{R}^k$. In the tactic-machine-idiom, the summary of a choice is effectively the string representing the corresponding tactic. Although simple, choice summaries pose similar issues as explicit prompts do: a choice may represent an arbitrarily sophisticated computation, and it may not be clear how a programmer should "summarize" it.

Pseudo environments. Additional data could be tracked by the nondeterministic programs in the form of a (pseudo) environment mapping identifiers to stacks of arbitrary (embeddable) datatypes, for the purpose of enriching the explicit prompts. The necessary bookkeeping could be hidden as much as possible by syntactic sugar. The `let` construct could be interpreted as 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.

Self-contained ML-powered subroutines. Another approach that is complementary to ranking choices during search is to provide self-contained ML-powered subroutines for various search strategies to call. For example, ML may be used within an information retrieval system that maps goals to plausibly relevant lemmas without any specific heed as to how the lemmas will actually be used by a given caller. Many problem solving strategies are parameterized by previously established facts, and so such an API may provide useful support for a wide range of strategies. A self-contained ML-powered module that conjectures upper or lower bounds for various terms may be widely applicable

as well. ML can also be applied within stand-alone provers for simpler logics, *e.g.* within superposition solvers Loos et al. (2017) or SAT solvers Selsam & Bjørner (2019). Of course, an ML-based prover trained in the tactic-state automata idiom could constitute a useful subroutine as well. More specialized ML-powered subroutines could also provide value in certain circumstances, *e.g.* one that suggests promising auxiliary points for geometry problems. This approach is appealingly modular and in most circumstances would be best practice; however, modularity can be a double-edged sword in machine learning, especially when data is scarce, since the more one fragments the data, the less each model has to train on.

Direct inspection via metaprogramming. The last approach we consider is to use metaprogramming to directly inspect the `SearchT m` α candidates in order to automatically encode each one in a form suitable for a machine learning system. This is essentially the approach proposed in Selsam et al. (2020a), though whereas they built an entirely new type of programming language to support it, here we consider lightweight approaches to harness similar power within general purpose languages. The feasibility depends heavily on the details of the language being extended.

In Python, a barebones `SearchT` program can be approximated as a thunk that either returns a special `ChoicePoint` object or a regular value. In this encoding, the `inspect` module for inspecting live objects together with the `dis` module for disassembling Python bytecode make it relatively straightforward to construct a lossless encoding of a given choice. Specifically, the bytecode of the choice (and whatever functions it calls in turn) can be traversed at runtime, and all symbols can be easily resolved as well. The situation is more complicated in Lean (version 4) since Lean is a high-performance language whose runtime does not include type information, function names, nor an environment. Nonetheless we can simulate the Python approach as follows:

1. Create a new inductive type `Object` to represent runtime objects.
2. Add a new primitive `inspect : NonScalar → IO Object` that structurally traverses any *non-scalar* and produces a corresponding `Object`. Using the procedure `dladdr`, it can resolve function (`void *`) addresses to (mangled) names.
3. To inspect a non-scalar term `x`, call `inspect (unsafeCast x : NonScalar)`.
4. To resolve function names appearing in the resulting `Object`, create a new Lean environment that imports the necessary modules; then after unmangling the function names, one can lookup the Lean IR code corresponding to each function referenced in the `Object` (and in other functions recursively thereafter).

This approach has a significant disadvantage in Lean compared to *explicit prompts* for inspecting the current state itself: a custom, type-aware embedding of the state datastructure itself may be much more compact than the runtime object that represents it. For example, it is common to show machine learning models only *pretty-printed* expressions, which discard troves of irrelevant information from the original expressions. Similarly, a Lean tactic-state does not just include the list of open goals but also includes a *metavariable context* containing information about previously solved goals that is not relevant for solving the open ones. Neither of these concerns would be significant if there were runtime type information since the embeddings could be user-defined and type-dependent. Unfortunately, we do not see how to make the generic metaprogramming approach practical without runtime type information.

4 Discussion

Ultimately, we see no silver bullet for guiding arbitrary nondeterministic tactics in practice. On the other hand, 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.

Acknowledgments

We thank Jesse Michael Han, Ryan Krueger, Leonardo de Moura, Sebastian Ullrich and Patrice Godefroid for helpful discussions and feedback.

REFERENCES

Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. Autopandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pp. 378–388. Springer, 2015.

Mark P Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming*, 5(1):1–35, 1995.

Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. *arXiv preprint arXiv:1701.06972*, 2017.

Daniel Selsam and Nikolaj Bjørner. Guiding high-performance sat solvers with unsat-core predictions. In *International Conference on Theory and Applications of Satisfiability Testing*, pp. 336–353. Springer, 2019.

Daniel Selsam, Jesse Michael Han, Leonardo de Moura, and Patrice Godefroid. Universal policies for software-defined mdps. *arXiv preprint arXiv:2012.11401*, 2020a.

Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. Tabled typeclass resolution. *arXiv preprint arXiv:2001.04301*, 2020b.