Tinkering by Theory Formation*

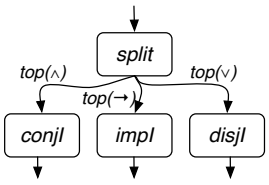
Gudmund Grov & Colin Farquhar (Heriot-Watt University, Edinburgh, UK)
Alison Pease (Dundee University, UK)
Simon Colton (Goldsmiths College, UK)

1 PSGraph and the Tinker tool

Most interactive theorem provers support encoding of common proof strategies as special function called *tactics*. Such tactics tend to work backwards from the goal, reducing a goal to a set of simpler sub-goals. Proof strategies are then created by combining such tactics using a *tactic language*. Such languages are often not designed to distinguish goals in cases where tactics produce multiple sub-goals. Thus when composing tactics, one has no choice but to rely on the order in which goals arrive, thus making them brittle to minor changes. For example, consider a case where we expect three sub-goals from tactic t_1, where the first two are sent to t_2 and the last to t_3. A small improvement of t_1 may result in only two sub-goals. This "improvement" causes t_2 to be applied to the second goal when it should have been t_3. The tactic t_2 may then fail or create unexpected new sub-goals that cause some later tactic to fail.

As a result: (1) it is often difficult to compose tactics in such a way that all sub-goals are sent to the correct target tactic, especially when different goals should be handled differently; (2) when a large tactic fails, it is hard to analyse where the failure occurred; (3) the reliance on goal order means that learning new tactics from existing proofs has not been as successful for tactics as it has been for discovering relevant hypotheses in automated theorem provers; and (4) large complex tactics are difficult to understand and maintain. As a result it is the easiest way for a user to deal with failure is to manually guide the proof until the tactic succeeds (or becomes unnecessary), rather than correcting the weakness of the tactic itself, complicating the overall proofs.

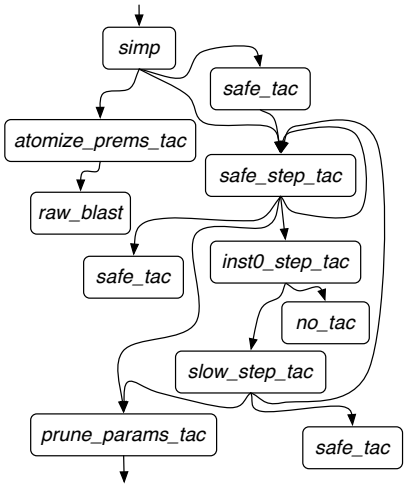
Proof-strategy graphs (PSGraphs) [5] overcome these deficiencies following a "plumbing" approach to combine tactics, and has been implemented in the *Tinker* tool [4]. Here, tactics appear as nodes in a special type of directed graph with the additional property that we allow dangling edges, i.e. edges without source and/or target nodes. Edges without source or target nodes serve as inputs and outputs to the graph as a whole. Tactics are then combined by "piping" them together with an edge in the graph. One evaluates a PSGraph by placing one or more goal-nodes, each containing a single goal, on input edges of the graph, then applying tactics to consume goals on the in-edges of a tactic-node and produce sub-goals on the out-edges. As a result, the goals appear to flow through the graph, hitting tactics along the way, until they are either consumed (i.e. closed), or reach the output edges of the graph, in which case they become output goals for the overall evaluation. A tactic-node may have multiple output edges, and to decide which edge a goal should be sent to we label edges with *goal-types*. These encode certain properties about a goal which dictate how it should then be handled. On the right an example PSGraph is shown, encoding a sub-part of the well known introduction tactic. Here, three introduction rules are given: conjunction introduction, implication introduction and disjunction introduction. Crucially, each of these tactics only works when the top level symbol is $\wedge$, $\longrightarrow$ or $\vee$, respectively – and this information is encoded in the incoming goal types to ensure a goal is sent to the correct place.



*Supported by EPSRC grants EP/J001058 and EP/H023852, and a Heriot-Watt DTA scholarship.

Such goal-types enables us to abstract over goal order and numbers, and provides a more declarative view of a proof structure, helping to identify *where*, *how* and *why* a proof failed. Importantly, it overcome problems previous attempts to learn tactics have had (e.g. [3]): the representation can capture properties of the goals at a sufficiently abstract level in order to, for example, identify case splits and termination conditions for iteration. We also believe this is crucial when learning from single instances, as in *proof by analogy*.

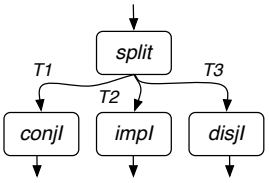
2 A theory formation approach for learning goal types



In order to extract a sufficiently abstract proof strategy both the graph structure and the goal types on the edges must be learnt, and here we address the sub-problem of *learning goal types*. In addition to being a sub-problem of such a strategy language, it can help with understanding existing known tactics – hopefully improving maintainability. To illustrate, on the right a first approach to encoding the common 'auto' tactic of Isabelle in PSGraph is depicted. Here, some information is abstracted away, such as "first try this tactic and if it fails try this one", as this does not help with the understanding – our ambitious goal is to *know* when to apply one or the other. Thus, we make an edge for each possibility in the graph. We are now left knowing the structure of the graph, however we do not know any of the goal-types. As the Isabelle library (and associated Archive of Formal Proofs) contains many examples where this tactic is used, we can try all possible paths of the 'auto' PSGraph and compare with the result of the standard 'auto' tactic. This information can then be used to label each sub-goal on each edge as a positive or negative example, and from that machine learn the goal-type.

Our goal is to *understand* the proof strategy, by discovering a predicate in the form of a goal-type capturing the valid goals of that path in the graph. Thus, more traditional statistical machine learning methods, which will give a classifier assigning a probability given a goal, are not desirable. Experience with the *rippling* proof strategy [1], encoded in PSGraph in [5, 4], has lead us to believe that this requires us to invent new predicates. In rippling, artefacts such as *embedding* of terms into other terms, *measure reduction* and *wave rules* are created in order to describe the goal-types. For other strategies, other artefacts may be required, and crucially these are unknown. This leads us to the *artefact generation* paradigm of AI, where, as apposed to the more common *problem solving* paradigm, the challenge is to generate new artefacts [2]. These artefacts can then be combined to describe a particular goal type.

An example of such tool is the HR *Automated Theory Formation* tool, and here we will outline some preliminary results of learning goal types using the latest version, called HR3 [2]. Beginning with an initial set of concepts and some examples, it applies a set of production rules to generate new concepts and conjectures. These concepts can either define objects of interest, such as integers, or a relationship between one or more such objects, for example the number of divisors of an integer. The production rules enable HR to perform operations such as negating, matching or composing concepts to generate new ones. These rules can be applied recursively to the newly discovered concepts to repeat the process until either no more can be found or the process is stopped manually.



To check for suitability, we conducted a small experiment with HR to see if it is able to learn goal types. Here, we looked at the simple introduction tactic discussed above, but removed the goal-types. This can be seen on the left where each of the goal types are replaced by a simple label, and our goal was to see if it HR could discover the required concept of *top* symbol from the set of simple examples. These had to be encoded in Prolog-like notation to enable HR to use them. Assuming the existence of concepts for: `edge`, as in the edge in the graph; `symbol` (as in e.g. $\longrightarrow$ written as `imp`), and `term`, to represent the terms of a goal, we start with a concept of a goal[1]:

```
goal(A,B,C,D) :- edge(A), symbol(B), term(C), term(D).
```

The goal $A \longrightarrow A$ on edge $T1$, for example, would be represented as

```
goal(T1,imp,A,A).
```

We then gave HR a handful of such examples of valid goals on edge $T1$. From two production rules, *instantiation* and *existential*, we could then find the desired concept.

First, the *instantiation* production rule will instantiate a variable with a constant, and in for our case the symbol is instantiated, creating the concept:

```
imp_goal(A,B,C) :- edge(A), term(B), term(C), goal(A,imp,B,C).
```

This concept captures the desired goal, however, we still need to give the terms as arguments. This can be overcome by the *existential* production rule, which existentially quantifies variables, represented by prefixing the name with _:

```
T1(A) :- edge(A), goal(A,imp,_X1,_X2).
```

This is the desired predicate for a goal type.

The experiment has shown feasibility for our approach. However, the example is very trivial and we expect that other common machine learning tools would have been able to discover this predicate. Next we plan to address how to test larger tactics, building up towards something like 'auto'. In such cases we will not know in advance what concepts will be required, and it will be very interesting to see what HR may come up. A key challenge here will be to prune the number of concepts generated, by developing measures of how interesting concepts are and strategies for production rule applications.

References

[1] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

[2] Simon Colton, Ramin Ramezani, and Maria Teresa Llano. The HR3 Discovery System: Design Decisions and Implementation Details. In *Proceedings of the AISB symposium on Computational Scientific Discovery*, 2014.

[3] Hazel Duncan. *The use of Data-Mining for the Automatic Formation of Tactics*. PhD thesis, University of Edinburgh, 2002.

[4] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. Tinker, Tailor, Solver, Proof. Submitted to UITP 2014.

[5] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. A Graphical Language for Proof Strategies. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 324–339. Springer, 2013.

[1] We have simplified notation slightly, and given the new concepts suitable names to ease the reading.