Developing and Debugging Proof Strategies by Tinkering

Developing & Debugging Proof Strategies by Tinkering*

Yuhui Lin, Pierre Le Bras and Gudmund Grov

Heriot-Watt University, Edinburgh, UK {Y.Lin, PL196, G.Grov}@hw.ac.uk

Abstract. Previously, we have developed a graphical proof strategy language, called *PSGraph* [4], to support the development and maintenance of large and complex proof tactics for interactive theorem provers. By using labelled hierarchical graphs this formalisation improves upon tactic composition, analysis and maintenance compared with traditional tactic languages. PSGraph has been implemented as the *Tinker* system, supporting the Isabelle and ProofPower theorem provers [5]. In this paper we present *Tinker2*, a new version of Tinker, which provides enhancements in user interaction and experience, together with: novel support for controlled inspection; debugging using breakpoints and a logging mechanism; and advanced recording, exporting and reply.

1 PSGraph & Tinker

Most interactive theorem provers provide users with a tactic language in which they can encode common proof strategies in order to reduce user interaction. To encode proof strategies, these languages typically provide: a set of functions, called *tactics*, which reduces sub-goals into smaller and simpler sub-goals; and a set of combinators, called *tacticals*, which combines tactics in different ways.

Composition in most tacticals either relies on the number and the order of sub-goals, or is to try all tactics on all sub-goals. The former is brittle as the number and the order could be changed if any of the sub-tactics changes; and the latter is hard to debug and maintain, as if a proof fails the actual position is hard to find. It is also difficult for others to see the intuition behind tactic design.

To overcome these issues we developed *PSGraph*, a graphical proof strategy language [4], where complex tactics are represented as directed hierarchical graphs. Here, the nodes contain tactics or nested graphs, and are composed by labelled wires. The labels are called *goal types*: predicates describing expected properties of sub-goals. Each sub-goal becomes a special *goal node* on the graph, which "lives" on a wire. Evaluation is handled by applying a tactic to a goal node that is on one of its input wires. The resulting sub-goals are sent to the out wires of the tactic node. To add a goal node to a wire, the goal type must be satisfied. This mechanism is used to control that goals are sent to the right place, independent of number and order of sub-goals. For more details see [4].

In [5], we introduced the Tinker tool, which implements PSGraph with support for the *Isabelle* and *ProofPower* theorem provers[1]. Tinker consists of two

* This work has been supported by EPSRC grants EP/J001058 and EP/K503915.

[1] A *Rodin* version is currently under development.

parts: the **CORE** and the **GUI**, each is shaded in a separated grey boxes in Fig. 1. The core is implemented in Poly/ML, and handles the key functionality. The GUI is implemented in Scala. They communicate over a JSON socket protocol. In addition to the Tinker GUI, a user will work with the GUI of the theorem prover; Tinker is only used for the proof strategies. To achieve theorem prover independence, most functionality is implemented using ML `functors`. Each theorem prover has a special `structure` that implements a provided `signature`, as indicated by `Isa_Tinker` and `PP_Tinker` in Fig. 1.

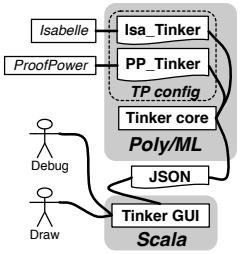
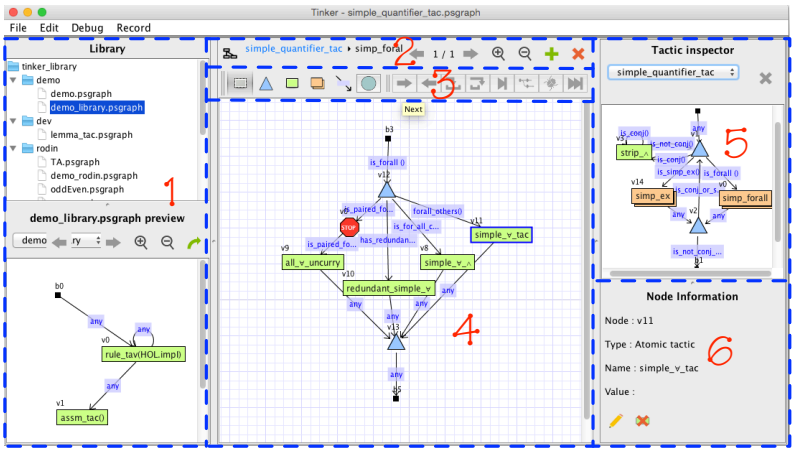


Fig. 1. Architecture

The main advantages of PSGraph over more traditional tactic languages (e.g. as found in Isabelle and ProofPower) are the ability of a step-by-step inspection of how sub-goals flow through the graph during evaluation, combined with features to debug and modify it. Such features are of great aid when debugging and maintaining proof strategies. It also provides a more intuitive representation to understand how the proof strategy works, also for non-developers (similar to graph visualisation of proofs in e.g. [7]). Low-level details can be hidden by using hierarchies to improve readability. Such features rely on good GUI support, which was only partially supported by the original Tinker tool [5]. Here, we introduce *Tinker2*, a new version of Tinker, which extends Tinker with new features, including supports for: library and hierarchical graphs; richer tactic and debugging options; and recording and replay. Fig. 2. shows the Tinker2 GUI and its layout.



1: Library panel
2: Hierarchy utilities
3: Drawing and evaluation controls
4: Graph panel
5: Hierarchical node inspector
6: Information panel

Fig. 2. The Tinker2 GUI and its layout.

We will use the ProofPower instance of Tinker2 in this paper, albeit we could just as well have used Isabelle as the features are identical. In §2 we focus on how to develop proof strategies from scratch; in §3 we discuss advanced features of evaluating, debugging, recording and replaying proofs; while we conclud and briefly discuss related and further work in §4.

2 Developing proof strategies

A user can draw a PSGraph from the *Graph panel* by selecting the type of node from the *Drawing and evaluation controls* panel (see Fig. 2). Nodes are connected by dragging a line between them. When selecting an entity, the details are displayed in the *Information panel*, and they can be edited by double clicking[2]. Fig. 3 shows the type of nodes that are supported by the tool.

Fig. 3. The node types.

Atomic tactics An atomic tactic wraps a tactic of the underlying theorem prover, which by default has the same as the name of the node. Tinker2 will automatically use all available tactics from the underlying prover. New tactics can be defined in the *tactic editor* of the Tinker2 GUI. To illustrate, the tactic definition

```
tactic all_∃_uncurry := fn [] => conv_tac all_∃_uncurry_conv;
```

creates a tactic with no argument (`fn []`). This tactic will be parsed and stored by the CORE, so that it can be used.

Hierarchical nodes Modularity is achieved by hierarchies. This can also help to reduce the complexity and size of a PSGraph by hiding parts of it. We will illustrate the new hierarchy features below.

Identity nodes Identity nodes are used to fanout and join wires. As the name suggests, they do not change the sub-goals.

Breakpoints A novel feature of Tinker2 is the introduction of breakpoint nodes, which can be added/removed from wires by a simple mouse click. We return to this is in §3.

Goal nodes A goal node wraps a sub-goal of a proof, and this can not be modified by the user, i.e. these nodes can only be changed through tactic applications, and introduced by the CORE when a new proof is started.

For the atomic tactics, a set of *atomic goal types* needs to be provided for each theorem prover. Tinker2 provides a Prolog-based language, with a dedicated editor, to develop these. To illustrate, the atomic goal type `top_symbol(t,s)` checks if term `t` has top symbol `s`. To declutter the graphs, we can define new goal types in the editor, which can then be used. For example:

```
is_conj() :- top_symbol(concl,conj).
```

```
(REPEAT (CHANGED
  (REPEAT strip_∧) THEN
  (TRY (all_∃_uncurry ORELSE
        redundant_simple_∃ ORELSE
        simple_∃_equation ORELSE
        simple_∃_∧)) THEN
  (TRY (all_∀_uncurry ORELSE
        redundant_simple_∀ ORELSE
        simple_∀_∧ ORELSE
        simple_∀_tac))))
```

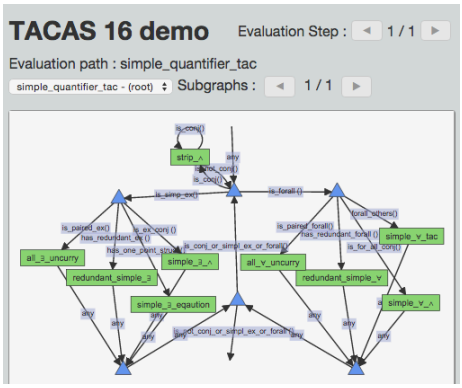


Fig. 4. `simple_quantifier_tac`: ProofPower (left) and PSGraph (right)

checks if the top symbol of the conclusion (`concl`) is a conjunction ∧ (`conj`).

As a running example, we will use a simple tactic to eliminate quantifiers in ProofPower, called `simple_quantifier_tac`. This simplifies goals by: 1) eliminating top level conjunction (∧) as much as possible; 2) eliminating the top level existential quantifier (∃) if they are redundant or can be simplified with the one point rule[3]; 3) eliminating the top level universal quantifiers ∀. A possible implementation using ProofPower's tactic language is shown in Fig. 4 (left), where `strip_∧` eliminates ∧; `all_∃_uncurry` and `all_∀_uncurry` change paired quantifiers to uncurried versions; `redundant_simple_∃` and `redundant_simple_∀` remove the quantified variables if they are not used in the body; `simple_∃_∧` and `simple_∀_∧` distribute quantifiers over ∧; `simple_∃_equation` simplifies goals with the one point rule; and `simple_∀_tac` instantiates each ∀ quantifier with an arbitrary free variable.

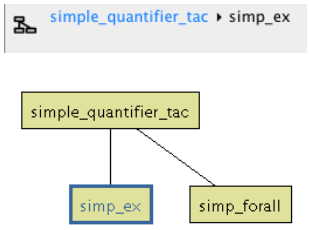


Fig. 5. Hierarchy Utilities

The right hand side of Fig. 4 shows an encoding of the same tactic in PSGraph, developed using the described GUI[4]. This can be further simplified, by "boxing" the sub-graphs that simplifies ∃ and ∀, respectively, using hierarchical nodes. This simplified version is given in Fig. 6. Tinker2 allows such "boxing" of sub-graphs into hierarchies, by a simple mouse click. Tinker2 also supports a range of features to work with hierarchies. In the *Hierarchical node inspector*, users can preview the internal structure of an hierarchical node. In the *Hierarchy utilities* panel, the hierarchical path of the current graph under editing is shown, as well as a tree view of the hierarchical structure of a PSGraph. A screenshot of a tree is shown in Fig. 5. It is also easy to move between and edit hierarchical nodes.

Reuse of PSGraphs is supported by a *library*. This feature is provided in the *Library panel* (see Fig. 2). The items in the library are PSGraphs. Therefore, the

[2] More details of running the tool is available from the user manual [2].
[3] In the one point rule $\exists x.P(x) \wedge x = t$ becomes $P[t/x]$
[4] See [9] for larger view, replay and video of this and other PSGraphs in Tinker2.

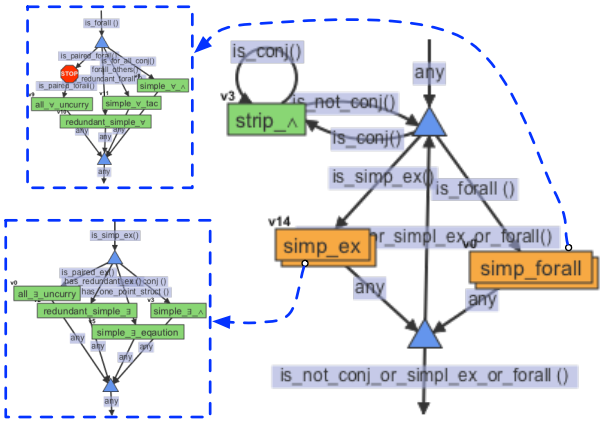


Fig. 6. Hierarchical PSGraph of `simple_quantifier_tac` tactic.

library can also be customised by simply copying PSGraph files into the library directory. When importing an item from the library to the current PSGraph, Tinker2 will copy it to the graph that the user is currently editing and merge all the required information, such as defined tactics and goal types.

3 Evaluating, debugging, exporting & replaying

A PSGraph in Tinker2 can be applied as a normal tactic/method within an Isabelle or ProofPower proof script. This is the normal execution. However, if it fails, it can instead be run in an 'interactive mode' where the GUI is used to visualise and guide how the proof proceeds and identify where it failed. Compared with the first version of Tinker, users can now: 1) select which goal to apply; 2) choose between stepping into and stepping over the evaluation of hierarchical nodes; 3) apply and complete the current hierarchical tactic; 4) apply and finish the whole proof strategy; 5) insert a breakpoint and evaluate a graph automatically until the break point is reached by a goal. These options are illustrated in the *Drawing and evaluations controls* panel of Fig. 2 (see also [9]), which also shows a break point in the graph.

To support debugging, an *evaluation log*, which shows the details of the current proof status, can be displayed. The log uses tags that can be used to filter the log to tags of interests. It also contains a real-time development mode that allows users to develop proof strategies seamlessly during proof tasks. Here, a user can freely edit the PSGraph (except for the goal nodes), e.g. change a tactic node, and then submit the changes to continue the current evaluation with the updated PSGraph. This is achieved using a new communication protocol, with details available in the second author's UG thesis [1], Note that this is currently not sufficiently constrained as one could edit paths a sub-goal has already passed thus invalidating the proof status, which we are now working on (see §4).

Tinker2 provides new features to export PSGraphs and record proofs. A PSGraph can be exported to the SVG format, e.g. to use in a paper; Fig. 6 illustrates this as the SVG diagram has been exported from Tinker2. The recording feature

can be switched on/off to start/pause recording changes made to a graph. These changes could have been made by the user or by the tool during evaluation. Once completed, such recording can be exported to a light-weight web application (written in HTML / CSS and JavaScript) via a generated JSON file. Fig. 4 (right) shows a screenshot of this, while [9] shows an example of this together with several screencasts of the GUI.

4 Conclusion, Related & Future Work

We have introduced a new version of the Tinker tool, called Tinker2, with a range of novel features to develop, debug, maintain, record and export hierarchical proof strategies. With Tinker2, users can easily reuse existing PSGraphs to develop and debug structured and intuitive hierarchical proof strategies. The most relevant work is the first version of the Tinker tool [5], which we have compared with throughout. It is also important to note that Tinker/Tinker2 is built on top of the Quantomatic graph rewriting engine [6], which is used internally as a library function. The second author has also developed web-based version of Tinker, which supports a subset of the GUI features discussed here [1]. With the exception of simple proof visualisation (e.g. [7]), we are not familiar with any other graphical proof tools to support theorem provers. While there are tactic languages that support robust tactics (e.g. Ltac [3] for Coq), we believe that the development and debugging features of Tinker2 are novel.

With D-RisQ (`www.drisq.com`) we are using Tinker2 to encode their highly complex Supertac proof strategy in ProofPower [8]. Several enhancements have been motivated by this work. In the future, we would like to improve static checking of PSGraph, such as being able to validate a PSGraph before evaluation. We also plan to improve the layout algorithm, and develop and implement a better framework for combining evaluation and user edits of PSGraphs.

References

1. P. Le Bras. Web Based Interface for Graphical Proof Strategies, 2015. Undergraduate CS Honours Thesis, available from `https://goo.gl/LWG522`.
2. P. Le Bras, G. Grov, and Y. Lin. Tinker: User Guide. `http://ggrov.github.io/tinker/userGuides.pdf`.
3. D. Delahaye. A Proof Dedicated Meta-Language. *Electronic Notes in Theoretical Computer Science*, 70(2):96–109, 2002.
4. G. Grov, A. Kissinger, and Y. Lin. A Graphical Language for Proof Strategies. In *LPAR*, pages 324–339. Springer, 2013.
5. G. Grov, A. Kissinger, and Y. Lin. Tinker, Tailor, Solver, Proof. In *UITP 2014*, volume 167 of *ENTCS*, pages 23–34. Open Publishing Association, 2014.
6. A. Kissinger and V. Zamdzhiev. Quantomatic: A Proof Assistant for Diagrammatic Reasoning. In *CADE-25*, volume 9195 of *LNCS*, pages 326–336. Springer, 2015.
7. T Libal, M Riener, and M Rukhaia. Advanced Proof Viewing in ProofTool. In *UITP 2014*, volume 167 of *EPTCS*, pages 35–47. Open Publishing Association, 2014.
8. C. O'Halloran. Automated Verification of Code Automatically Generated from Simulink. *ASE*, 20(2):237–264, 2013.
9. P. Le Bras Y. Lin and G. Grov. Tinker2 - TACAS 16 paper resources. `http://ggrov.github.io/tinker/tacas16/`. Accessed: 2015-10-17.