Keep Failed Proof Attempts in Memory

Yutaka Nagashima

Data61, CSIRO / NICTA**

Abstract. Eisbach made it possible to write custom methods within Isabelle's proof language, Isar. However, it still remains difficult to write custom methods correctly. We are developing a proof tracer for compound methods, which serves as a debugger for Eisbach methods. By remembering backtracked proof attempts of Eisbach methods, the tracer helps users realise where and how their methods go wrong.

1 Motivation

Background. Matichuk *et al.* introduced a proof method language, Eisbach [4], to Isabelle [6], making it easier for Isabelle users to write customised methods: with Eisbach, users can write their own compound methods without delving into the ML level any more. However, Eisbach's main contribution is in its syntactic support for writing methods; the other side of the difficulty, semantic complexity, still remains largely intact.

When a user applies their custom method to a proof obligation, the only useful information the user can obtain is the output of the method. As the method and proof obligation become more involved, the behaviour of the method becomes quickly intractable; if something goes awry inside a large compound method, the final goal state of the method usually does not tell where the method has gone wrong internally. One reason for this complexity lies in the Isabelle execution model that admits choice.

Choice. In general tactics in Isabelle can return multiple results as shown in the following signature of tactics:

```
type tactic = thm -> thm Seq.seq
```

where `thm` stands for a proof state, and `Seq.seq` for lazy sequence. Note that a value of `thm` may contain proof obligations as assumptions of meta-implications. This representation allows Isabelle to handle proof states in an uniform way, thus avoiding the `validation` function that appears in other ITPs that are based on LCF [7]. Since some tactics can return (possibly infinitely) many results, this formalisation provides more flexibility to tactics. Furthermore, combined by the

** NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

sequential method combinator, `(,)`, methods with multiple return values can specify a larger search space using backtracking over multiple atomic methods.

Even though this behaviour let users write powerful tactics and methods using backtracking, quite often backtracking arises unexpectedly, sometimes without users noticing it. However, investigating what happens inside a compound method is not straightforward, especially when methods with multiple results are combined sequentially with `(,)`. This is partly because `apply(meth1, meth2)` returns, in general, a different sequence than what `apply meth1 apply meth2` does.

The complexity of methods rises even higher when users also employ other method combinators, such as `?` to suppress method failures, `+` for repetition, and `;` to apply a method for all new subgoals emerging from the previous method. Clearly, we need a tool that helps us investigate what a compound method does internally when we apply it to a proof obligation.

2 Debugger for Compound Methods

2.1 Design Principle

Our aim is to provide a tool that supports users in seeing how their methods behave when applied to a proof obligation and finding out how to fix them if they behave in an unexpected way.

For this purpose, we decided to trace proof attempts of customised methods at *the right level*, storing *the right information* in *the right structure*. By "at the right level", we mean that we ignore small steps that happen within each atomic method. By ignoring mostly trivial steps, our tracing mechanism provides a broader view of how compound methods work. By "the right information", we mean that our tool does not only keep successful proof attempts in memory but it also remembers backtracked failed proof attempts. It is because of the nature of debugging that the tool should remember how a method fails not only how it succeeds. This is in contrast with our previous work [5], in which we generate fast proof scripts that do not involve many backtracked steps, by tracing proof attempts while scraping off failed proof attempts. Lastly, for "the right structure" we chose the following tree structure, we designed for this purpose.

```
datatype 'a hltree = HLTree of ('a * 'a hltree Seq.seq option);
```

where the pair type represents the combination of parent node and its sub-trees. We can use this tree structure to identify how many backtracked steps are taken after each node until the search backtracks to it for the last time.

Note that we use the type constructor `option` to denote the state of proof attempt: `HLTree (st, NONE)` denotes that the state `st` may be evaluated further by a method, while `HLTree (st, SOME Seq.empty)` denotes that the method applied to `st` failed, returning an empty sequence. Furthermore, this tree is horizontally lazy but vertically strict. We chose this evaluation order to emulate

Isabelle's basic method combinators that implement the depth first search, while admitting lazy evaluation for each atomic method.

Based on this data structure, `hltree`, we introduced a new type signature for traced tactics as follows:

```
type 'a traced_tactic = 'a -> 'a hltree
```

Comparing `traced_tactic` with `tactic` makes it clear that the only thing we have to do to trace proof attempts is to define some functions on `hltree` to replace the corresponding functions defined on `Seq.seq`. Since we are trying to develop a debugger for Eisbach, we only have to replace those functions used by Eisbach.

2.2 Implementation Principle with an Example: `(,)`

As an example, we show how we trace a sequentially combined method `meth1, meth2` using `hltree`. In the default setting, the method combinator `(,)` is mapped to the sequential tactic combinator `THEN`, which performs one tactic followed by another and is defined as:

```
fun Seq.maps f xq =
  make (fn () =>
    (case pull xq of
      NONE => NONE
    | SOME (x, xq') => pull (append (f x) (maps f xq'))));

fun (meth1 THEN meth2) st = Seq.maps meth2 (meth1 st);
```

Note that `Seq.maps` applies `meth2` to all the results of `meth1 st`, implementing the backtracking scheme discussed in section 1. To emulate this behaviour, we first define `map_leaves`, a function on `hltree` that corresponds to `Seq.maps`.

```
fun map_leaves (meth:'a->'a hltree) (HLTree (st, NONE)) = meth st
 |  map_leaves (meth:'a->'a hltree) (HLTree (st, SOME hltrs) =
      HLTree (st, SOME (Seq.map (map_leaves meth) ltrs));
```

As is obvious from the definition, `map_leaves` maps the method `meth` to all the leaves of the given `hltree`. However, unlike `Seq.maps`, `map_leaves` does not merge results from different parent nodes nor replace the parent node with child nodes. Instead of that, `map_leaves` grows the `hltree` deeper, keeping ancestral nodes that represent intermediate goal states. Using `map_leaves`, we define `THEN` on `traced_tactic` as follows:

```
fun (meth1 THEN meth2) st = map_leaves meth2 (meth1 st);
```

The similarity between the two versions of `THEN` is not a coincidence; we try to follow the default Isabelle implementation as closely as possible while preserving failed proof attempts in `hltree`.

As the two versions of THEN, one on Seq.seq and the other on hltree, are meant to be equivalent, some functions based on these can be defined in the same way. For example, the TRY combinator in the ML structure Seq is defined as follows:

```
fun TRY f = ORELSE (f, succeed);
```

where ORELSE and succeed are defined on Seq.seq, while we defined the corresponding combinator as following:

```
fun TRY f = ORELSE (f, succeed);
```

where ORELSE and succeed are defined on hltree. The identical function bodies of these two functions raises our confidence in the assumption that our proof trace mechanism correctly emulates Isabelle's native execution model. In theory, we could treat Seq.seq and hltree as members of the same modular constructor class, avoiding code duplications like the one shown above; however, we have not done such refactoring since it involves a significant change to the existing Isabelle source code.

2.3 Integration to Isabelle/jEdit

We are integrating this proof tracing feature into Isabelle/jEdit. To avoid large code duplications, we decided to piggyback on Isabelle/Isar's standard mechanism to apply methods to proof obligation. This also helps our tracing mechanism to minimise its deviation from Isabelle's default execution mechanism.

We provided a configuration flag to turn on and off the tracing mechanism: users can turn on the tracing mechanism simply by typing:

```
declare [[ method_trace = true ]]
```

When this flag is turned on, this flag replaces Method.evaluate with our proof tracing function, HLTree.evaluate_to_seq, in the definition of apply_method inside the ML module Proof, thus changing how the apply command works.

Essentially, evaluate_to_seq does the following: given a proof goal and (possibly composite) method, it piggybacks on the existing Isar mechanism and applies a method to a proof obligation, tracing proof attempts using hltree. The modified apply command finishes its execution where the default apply command stops, returning the proof trace including backtracked steps. We made this possible by making hltree horizontally lazy but vertically strict. Then, evaluate_to_seq removes implicit branches that denote possible future steps, before counting how many backtracked steps each subtree involves. Finally, evaluate_to_seq flattens the hltree into a lazy sequence, making the type signature of evaluate_to_seq compatible with that of Method.evaluate. Users can investigate the stored trace using the Isar keyword, back.

To illustrate this reuse of Isar commands, we included two screen-shots, Figure 1 and Figure 2, as an example. In Figure 1, the cursor is on the apply command. Since the flag, `method_trace`, is set to `true`, this `apply` command constructs a `hltree`, storing the intermediate steps. After constructing the `hltree`, the output panel shows the original proof state and the number of backtracked steps. Figure 2 shows how to inspect the contents of the `hltree`. Since the modified `apply` command flattens the `hltree` into a lazy sequence, users can investigate the contents of the `hltree` using the `back` command. In Figure 2, you can see that the elimination rule `conjE` was first applied to `A ∧ B` instead of `C ∧ D`, which will cause backtracking in the next step.

One can turn off this tracing feature by typing:

```
declare [[ method_trace = false ]]
```

Below this line, the `apply` and `back` commands work as usual without producing or inspecting a `hltree`. In the example mentioned above, if you flip the flag in line 41 to `false`, `apply (erule conjE, assumption)` returns a lazy sequence that contains a single proved goal only, causing the subsequent `back` command to return an error message, `back: no alternatives`.

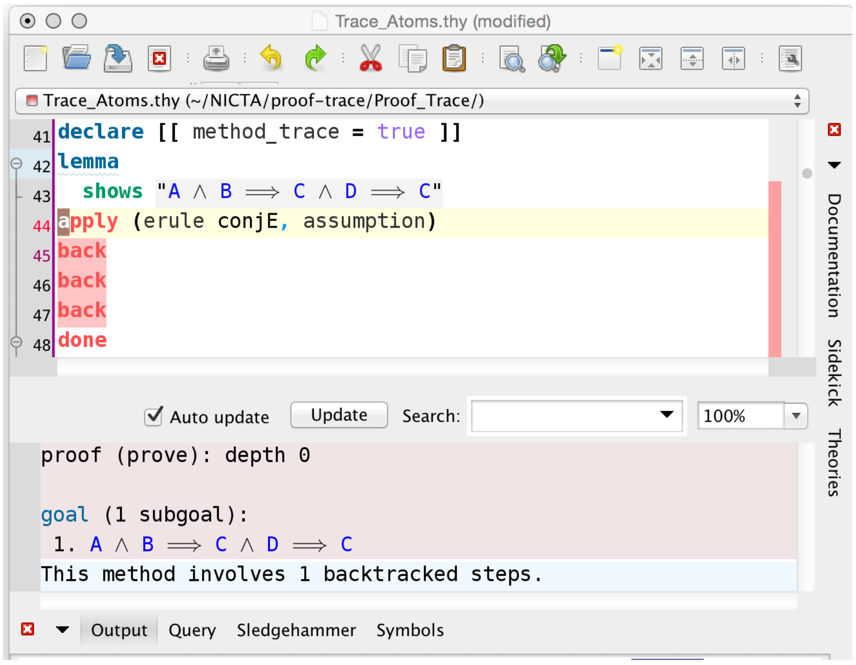
3 Future Work

We have confirmed that our tracing mechanism works as expected against small example proofs. However, we consider that there is room for improvement to make this mechanism more reliable and useful.

Firstly, we have to validate our implementation. So far, we are trying to make our mechanism correctly emulate Isabelle's default proof procedure simply by reusing a large amount of Isabelle/Isar code and making the functions on `HLTree` similar to the corresponding functions on `Seq.seq`, minimising the deviation from the default procedure. To ensure the correctness of the implementation, we need to conduct many tests, probably using existing proof scripts.

Secondly, we have to provide a graphic user interface in Isabelle/HOL to make our proof trace more useful. Currently, it only supports text based interaction, which re-uses the `back` command, changing the behaviour of the `apply` command. We consider this as a temporary workaround and plan to add a jEdit panel for our proof tracing, so that users can investigate the runtime behaviour of their custom methods, without affecting the result of the `apply` command. In the panel, we plan to add the following features:

- The panel shows a trace of concrete atomic methods that does not involve backtracking on the method level.
- Each atomic method is tagged with numbers representing the number of backtracked steps and time spent until the final backtracking to that state.
- Users can click on each node to see the pretty printed intermediate proof goal in that node.
- Users can expand each node on the trace to see what steps are taken before leaving the state for the last time.



Trace_Atoms.thy (modified)

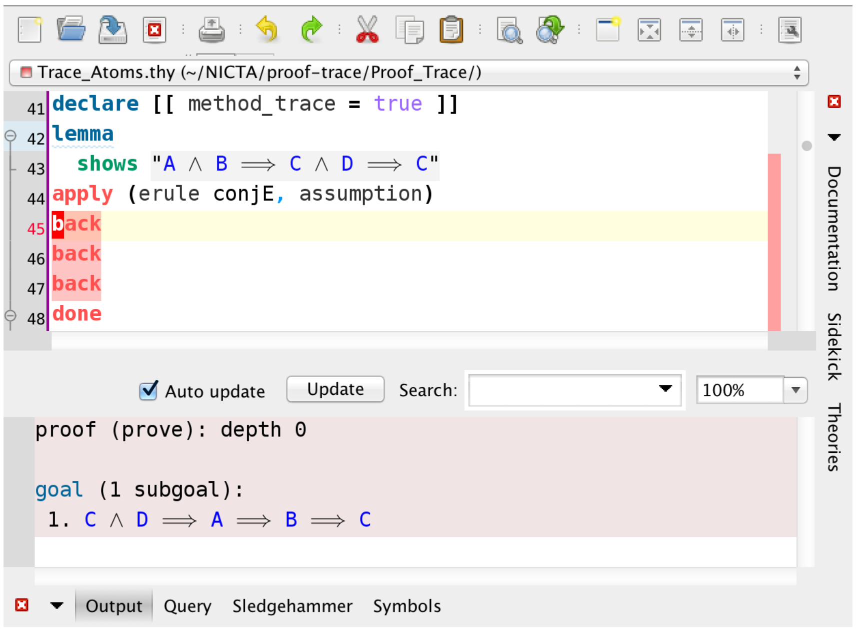
Trace_Atoms.thy (~/NICTA/proof-trace/Proof_Trace/)

```
declare [[ method_trace = true ]]
lemma
  shows "A ∧ B ⟹ C ∧ D ⟹ C"
apply (erule conjE, assumption)
back
back
back
done
```

```
proof (prove): depth 0

goal (1 subgoal):
 1. A ∧ B ⟹ C ∧ D ⟹ C
This method involves 1 backtracked steps.
```

Fig. 1. `hltree` construction with the `apply` command.



Trace_Atoms.thy (~/NICTA/proof-trace/Proof_Trace/)

```
declare [[ method_trace = true ]]
lemma
  shows "A ∧ B ⟹ C ∧ D ⟹ C"
apply (erule conjE, assumption)
back
back
back
done
```

```
proof (prove): depth 0

goal (1 subgoal):
 1. C ∧ D ⟹ A ⟹ B ⟹ C
```

Fig. 2. Trace inspection with the `back` command.

Lastly, we are planning to use this tracing mechanism for supervised learning. Our plan is to generate a number of successful and failed proof attempts by applying the runtime tactic generation discussed in the `PSL` paper [5] against existing proof scripts while tracing them using `hltree`. This would be in contrast with the preceding proof mining work [1,3], which investigates manually written proof scripts. Manually written proofs are expected to be good quality information, but it shows one way to prove a conjecture while there are often many different ways to prove the same conjecture. We expect that this will result in large data from which we can learn mechanically how to avoid proof attempts that are likely to fail in order to improve the performance of automatic proof search.

4 Related Work

There have been several attempts to trace Isabelle's proof procedure. Lars Hupel developed a visualization tool-kit for interactively tracing simplifier in Isabelle/jEdit [2]. We will follow his visualization approach but on a different level: while his tool shows the inner steps of atomic tactics, ours ignores them focusing on a larger view of the behaviour of user defined methods.

We, ourselves, previously developed a proof tracing mechanism to generate fast proof scripts that do not involve many backtracked steps [5]. In that project, we focused on the successful parts of proof attempts, scraping off backtracked steps at runtime using monad transformers. On the contrary, our focus in this project is on the failed parts: we keep failed proof attempts in memory and let users investigate where and how their composite methods go wrong.

5 Conclusion

In this paper, we discussed the on-going development of our debugger for Eisbach methods and its future work. Our aim is to provide an interactive platform to investigate the runtime behaviour of custom methods written in Eisbach. For this purpose, we store proof traces in lazy trees, including failed proof attempts. We are trying to make our tool faithful to the native execution of Isabelle/Isar; however, its correctness is yet to be validated.

Acknowledgement

We thank Gerwin Klein for comments that greatly improved the manuscript.

References

1. Blanchette, J., Haslbeck, M., Matichuk, D., Nipkow, T.: Mining the archive of formal proofs. In: Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, Volker Sorge (ed.) Conference on Intelligent Computer Mathematics. pp. 3–17. Springer, Washington DC, USA (jul 2015)

2. Hupel, L.: Interactive simplifier tracing and debugging in isabelle. In: Intelligent Computer Mathematics - International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings. pp. 328–343 (2014), `http://dx.doi.org/10.1007/978-3-319-08434-3_24`
3. Matichuk, D., Murray, T., Andronick, J., Jeffery, R., Klein, G., Staples, M.: Empirical study towards a leading indicator for cost of formal software verification. In: International Conference on Software Engineering. p. 11. Firenze, Italy (feb 2015)
4. Matichuk, D., Murray, T., Wenzel, M.: Eisbach: A proof method language for isabelle. Journal of Automated Reasoning 56(3), 261–282 (mar 2016)
5. Nagashima, Y., Kumar, R.: A proof strategy language and proof script generation for isabelle (2016)
6. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002), `http://dx.doi.org/10.1007/3-540-45949-9`
7. Paulson, L.C.: The foundation of a generic theorem prover. J. Autom. Reasoning 5(3), 363–397 (1989), `http://dx.doi.org/10.1007/BF00248324`