From Tactics to Structure Editors for Proofs

Extended Abstract

Xuanrui (Ray) Qi
Department of Computer Science
Tufts University
Medford, Massachusetts, USA
xqi01@cs.tufts.edu

ABSTRACT

For many users of theorem provers, tactics are merely an inelegant solution to a difficult problem. However, we believe that tactics deserve more attention and appraisal, and well-designed tactics are actually a well-structured way to construct programs. In this work, we will explore the connection between tactics and structure editors, as well as emphasize the importance of designing tactics with well-defined semantics with respect to proof terms and proof states.

CCS CONCEPTS

• **Theory of computation** → **Type theory**; • **Software and its engineering** → **Software development techniques**; *Domain specific languages*;

KEYWORDS

Tactics; structure editors; proof assistants; dependent types

ACM Reference Format:
Xuanrui (Ray) Qi. 2019. From Tactics to Structure Editors for Proofs: Extended Abstract. In *Proceedings of Off the Beaten Track 2019 (OBT'19).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROPOSAL

An alternative to traditional text-based editors, structure editors have the benefit that their edit states directly correspond to program structure [9]. However, despite the many advantages of structure editors, which will not be reiterated here, traditional source-based editors can be much efficient for most programmers. A nice middle ground would be to combine text-based editing and structure editing, which has already generated a line of research. A lucrative feature of structure editors, for example, is the ease of adding automation to the editor; one may want to have a structure editor's automation features while retaining the ability to directly edit the source code in text form.

On the other hand, a similar dilemma exists in the world of theorem proving with dependent types: Agda and Idris-style "proofs-as-terms" development of proofs have the benefit of clarity and directness, but for large, complex proofs, Coq-style tactic-based theorem proving can be much easier to handle, due to the high level of automation made possible by tactics. However, tactics are widely regarded as "hacky" and "inelegant", as they are highly opaque and mask the computational nature of proofs à la Curry-Howard.

Much of the previous work on tactics, such as Mtac [10] and Mtac2 [3] have essentially been on bringing tactic-based theorem proving more in-line with "direct-style" proving. However, we believe that it is important to treat tactics as first-class citizens, and that the dilemma between tactics and direct proof construction is a false dilemma. Instead of having to choose between one or another, why not *embrace* both and combine direct-style proof construction and tactic-based theorem proving within a single system?

2 TACTICS AS EDIT ACTIONS

In a structure editor like Hazel, applying an edit action transforms an *edit state* into another edit state [8]. Similarly, a tactic transforms a *proof state* into another proof state. A *cursor* in Hazel corresponds to a *focused goal* in a theorem prover. More correspondences could be drawn between Hazel-style structure editors and Coq-style tactic-based theorem provers, and the following table lists a few:

Structure editor (e.g. Hazel)	**Theorem prover (e.g. Coq)**
edit action	tactic
edit state	proof state
cursor	focused goal
moving the cursor	focusing on a different goal
empty hole	unproven goal
non-empty hole	partially proven goal
action macros	tacticals

In other words, just as proofs are programs, tactics are just edit actions for proofs, and a theorem prover is just a structure editor for proofs. However, with Ltac as its "action language", Coq is not a *good* structure editor. For example, unlike good structure editors for programs, it does not actually show us the overall state of the proof, and neither does it suggests tactics to use based on the current proof state: a root cause of this chaotic behavior here is that we are not actually sure *how* tactics work and *why* they actually work (or not).

3 LESSONS FROM HAZEL

We believe that lessons from the development and design of Hazel could help here. One major complaint about tactics — especially Coq tactics — is that they do not have clear static specifications, and thus can be tedious to debug or maintain, and almost impossible to reason about. A solution to this would be to *formalize* the semantics

OBT'19, January 2019, Lisbon, Portugal
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

of tactics in a Hazel-style edit action model, specifying the effect of each tactic on a proof state.

It is not the case that semantics of tactics have never been formalized before: the semantics for both Coq's Ltac [2] and Edinburgh LCF's tactic language [7] have been formalized. Our proposal here, however, differ from previous work in our *approach* towards semantics for tactics: instead of considering proofs and goals as a stack machine separate from the underlying dependently-typed language, we view proof states as incomplete programs, in the spirit of Curry-Howard, and specify the semantics of tactics in terms of how they *manipulate* and *change* the proof state. This view of proof states is not new either [5, 6]. Recently, Korkut's work on edit-time tactics in Idris [4] has explored this approach as well, but we feel that our vision is somewhat different from edit-time tactics in Idris: specifically, edit-time tactics in Idris are considered a form of metaprogramming macros that exist outside of the abstract syntax tree, and require ad-hoc editor extensions. However, we envision tactics to be fully integrated into the term language, with first-class editor support. As far as we know, no one has yet formalized the semantics of tactics this way.

Having formal semantics is a good first-step towards a sound and productive tactic language, but it is not enough. We would also want to prove that our tactic semantics are actually *sensible*: in other words, we would prove a metatheorem akin to the *action sensibility* theorem in [8]. In theorem proving parlance, we may informally restate this theorem as: if a proof state with a focused goal is statically meaningful, i.e. after removing the focus the term is well typed, then applying any tactic to the proof state will give us a statically meaningful proof state. As we introduce dependent types, however, a type-level sensibility theorem would be required: before we claim that applying a tactic gives us a well-typed term, we must first show that the type ascribed to the resulting proof state (after "focus erasure") is indeed a meaningful type.

This theorem is somewhat similar to Mtac's preservation and safety theorems [10], and should enforce the same safety guarantees, but our approach has two advantages over Mtac: (1) our approach results in a much more idiomatic proof style, and (2) our approach allows reasoning about tactics which could not be implemented withinin the Mtac framework (such as low-level tactics like `apply`). Compared to the Mtac approach, our approach is much more heavyweight, but we conjecture that only a small subset of "core" tactics would need to be formalized and proven correct; most useful tactics could be implemented in terms of those core tactics. It would also be more difficult to prove tactic correctness within the proof system itself, but techniques akin to elaborator reflection [1] might be able to make this possible.

Finally, one important virtue of tactics is that they are programmable and customizable: so are edit actions. While Hazel does not have action-level programming as of yet, this is a feature that is currently being actively developed, and we may expect to have a sound metatheory for tactic-level programming.

Having these machinery in place, it is not hard to envision a *combination* of tactic-based and directly-style theorem proving: just like structure editors may have limited support for free-form input, someone proving a theorem might write an otherwise complete proof term, place the focus (i.e., "cursor") on the desired goal, and then use tactics (i.e., "edit actions") to construct that part of the proof.

4 AN EXAMPLE: SPECIFYING A `REWRITE` TACTIC

At the moment, it is impossible to precisely specify a `rewrite` edit action à la Hazel, as designing a dependently typed version of Hazel is still an open research problem. However, assuming that a dependently typed version of Hazelnut has a similar structure as Hazelnut presented in [8], we can write an imprecise inference rule for the `rewrite` edit action.

For the purposes of our example, let us consider an H-expression language similar to McBride's Oleg development calculus [6], and a Z-expression language that simply superimposes a "cursor" (or "focus"; we will use the two terms interchangeably) onto an H-expression.

$$\boxed{\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'}$$

$$\frac{\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \qquad \dot{\Gamma} \vdash s = t \Rightarrow \mathrm{eq}(\dot{\tau}') \qquad \hat{e}\,[s \mapsto t] = \hat{e}' \qquad s \sqsubseteq \hat{e}}{\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\texttt{rewrite}\ s=t} \hat{e}' \Rightarrow \dot{\tau}} \quad (\textsc{Rewrite})$$

Here, the judgment $s \sqsubseteq e$ means that s is a sub-expression in e. It is clear that this formalization is consistent with our intuition about the `rewrite` tactic, but as this formalization is by nature preliminary and imprecise, it might not accurately capture details about `rewrite`.

ACKNOWLEDGMENTS

I would like to thank Cyrus Omar, Ian Voysey, Matthew A. Hammer, and Joomy Korkut for discussion that inspired this work, as well as Jacques Garrigue for Coq hacks which ultimately motivated me to carry out this work. I thank Cyrus, Ian and Joomy again for providing feedback on the draft version of this talk proposal.

REFERENCES

[1] David Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 284–297. https://doi.org/10.1145/2951913.2951932

[2] Wojciech Jedynak, Małgorzata Biernacka, and Dariusz Biernacki. 2013. An Operational Foundation for the Tactic Language of Coq. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming (PPDP '13)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2505879.2505890

[3] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. 2018. Mtac2: Typed Tactics for Backward Reasoning in Coq. *Proc. ACM Program. Lang.* 2, ICFP, Article 78 (July 2018), 31 pages. https://doi.org/10.1145/3236773

[4] Joomy Korkut. 2018. *Edit-Time Tactics in Idris.* Master's thesis. Wesleyan University.

[5] Marko Luther and Martin Strecker. 1998. *A guided tour through* Typelab. Technical Report 98-03. Universität Ulm.

[6] Conor McBride. 1999. *Dependently Typed Functional Programs and their Proofs.* Ph.D. Dissertation. University of Edinburgh.

[7] Robin Milner. 1984. The use of machines to assist in rigorous proof. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 312, 1522 (1984), 411–422. https://doi.org/10.1098/rsta.1984.0067

[8] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming*

Languages (POPL 2017). ACM, New York, NY, USA, 86–99. https://doi.org/10.1145/3009837.3009900

[9] Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Commun. ACM* 24, 9 (Sept. 1981), 563–573. https://doi.org/10.1145/358746.358755

[10] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2013. Mtac: A Monad for Typed Tactic Programming in Coq. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 87–100. https://doi.org/10.1145/2500365.2500579