Towards user-friendliness in proof assistants: automated strategies via algebraic effects and handlers

Proof assistants provide a framework for the modelling and verification of theories, as well as trustworthy software. However, such power is usually only available to experts. We propose a new approach, based on algebraic effects and handlers, to integrate different automated proof strategies that enable newcomers to take advantage of proof assistants without an in-depth understanding of underlying theory. Our approach gives newcomers an effect system, a handful of effectful strategies (tactics, proof search, and SMT solvers) and their handlers under a shared interface, while advanced users can extend our system with new effects and new handlers. Lastly, we prototype the system as a library in Agda. While our prototype is minimal, it shows how easily proofs can be carried out so long as the user has the correct intuition. We believe our system empowers non-experts and has the potential to bring verified software to relevant industries, such as finance.

Additional Key Words and Phrases: effect handlers, proof assistants, user experience

ACM Reference Format:
. 2020. Towards user-friendliness in proof assistants: automated strategies **via** algebraic effects and handlers. 1, 1 (September 2020), 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 VERIFICATION FOR THE MASSES

The usability of proof assistants such as Coq [8] and Agda [18] is hard to measure: all of them require significant domain knowledge in type theory, the theory that enables such assistants to exist. Coq and Isabelle/HOL are more commonly applied outside their niche [4, 19], and their programming style is sharply different from mainstream programming and even functional programming. Granted, they come with their own integrated development environment (IDE) [12, 22], which may make it easier for students and newcomers. To our knowledge, no usability study has been held for a proof assistant, however there exists studies that account for informal “usability” criteria – they appear to measure how fast users can familiarise themselves with the system or IDE [23, 24].

Folklore within the proof assistants community says that newcomers find Coq hard to use due to a large library of tactics and difficulty reading the code after its completion. Coq learning materials, such as “Software Foundations” by Pierce *et al*, suggest that users add code comments for structural induction cases and inductive hypothesis. As mentioned, no user studies were conducted to attest that such strategy improves the perceived quality of Coq proofs.

Granted, proof assistants are a relatively new technology and there are no guidelines or even an intuition on how such systems should work or what features should be available to the users. The closest model we have are proofs by hand. However, following that model for proof assistants has two main pitfalls: (a) proofs by hand are not widely taught outside of college-level mathematics courses [3]; and (b) proof by hand and automated proof employ fundamentally different paradigms, where the latter requires axioms and formulations to be encoded explicitly.

Author’s address:

Manuscript submitted to ACM

1.1 Empowering users of different levels of expertise

To popularise software verification, we must make proof writing less of a burden to the user. Currently, only specialists can write proofs for their programs, even though the working programmer surely understands the domain and invariants of their projects – otherwise they would not be able to write any software. *Our hypothesis is that programmers are well-equipped to derive and prove properties of the software they write, but they lack the mathematical maturity and vocabulary to carry out a formal proof.* We have evidence that students fail to produce well-made proofs due to the lack of mathematical maturity, even though they do understand the subject matter at hand [13, 14].

In our approach, we propose an effects and handlers view of such proofs, based on prior work developed on the Andromeda proof assistant [5]. Here, our users will program as they would normally and invoke the proof environment as an effect to prove certain properties as they go. Given that the proof environment is *just* an effect, we envision that different proof styles (e.g., Agda-style dependent types , SMT solver, proof search) can be "composed" under a shared interface, a proof object that can manipulate itself while querying different automated provers.

The reasons we employ algebraic effects and handlers are numerous:

(1) as proofs cannot be fully automated, all approaches that try to automate the process (e.g, proof search, SMT solver) may be non-deterministic or never find a solution. Therefore, the system should be able to handle "impure" computations and errors. Algebraic effects and handlers have well-defined semantics and provide a simple interface for performing effects. With them, we avoid indiscriminate effects that are often error-prone and intricate effects machinery such as monad transformers;
(2) the semantics of effects and handlers accommodate composition of arbitrary effects and the composition of multiple handlers, which means users have the ability to weaken more general strategies into specific ones while maintaining top-level handlers unmodified;
(3) effects and handlers have well-defined semantics, and it is the main feature of many new research languages, such as Koka [15], Eff [7], Frank [11] and Effekt [9], which suggests a greater potential due to their popularity.

1.2 Goals and Contributions

The present project intends to be a compiler-level implementation as part of the Juvix[1] programming language. Juvix is designed with focus on working programmers familiar with strongly-typed functional programming.

Goals. The project exists to enable users to write safe, verified code without investing time into learning dependent types and theorem proving. The project's goals are to:

(1) make software verification more accessible to programmers familiar with functional programming;
(2) foster knowledge sharing between domain experts and regular programmers;
(3) lower the barrier to entrance into software verification to non-experts.

Contributions. The project is a work-in-progress, but this paper makes three important contributions that pave the way for the full implementation:

(1) a specification of a user interface for automated proofs based on algebraic effects and handlers that permits user-defined effects for proof strategies, and also user-defined handlers for existing effects (Section 2);
(2) a short technical account of the effort required to implement such a user interface for proof assistants via a a minimal prototype in Agda that validates the feasibility of such system (Section 3).

[1] Juvix is a dependently-typed programming language for smart contracts. More information at juvix.org

2 INTRODUCING WITCH

In this section, we introduce our approach, named Witch[2]. We define a *witch* as an assistive tool for theorem provers which congregates many different strategies for proof automation. Our specification for a *witch* for Juvix uses algebraic effects and handlers as the means of congregation.

Algebraic Effects and Handlers. Algebraic effects were introduced by Plotkin *et al* [20], and their handlers were introduced later on by Plotkin & Pretnar [21]. An algebraic effect is defined by a set of operations, and handlers that denote said operations. A handler may interpret operations at the compiler-level, which was formally named co-model, but colloquially called "top-level handler". Handlers are the most general form to denote a set of operations, and less general classes of handlers also exist [2].

The Essence of Witch. In Figure 1, we sum up our *witch* approach. As for the syntax, we use `operation { params } via handler`, which is an improvement over `handler(operation, params)`, since effect handling is similar to function application, but also carries effect information. The user defines data types and functions as usual, and then uses a *witch* to prove properties about said definitions. The Examples section in Figure 1 shows this in action. The examples are simplified for the sake of clarity – we avoid more complex properties and complicated proofs. The proof of commutativity of addition under natural numbers and of associativity of list concatenation are shown, and use the three main effects: Solver, Proof Search and Tactics. In the proof assistant literature, there exists no precise definition of commonly used terms "solver", "proof search" and "tactics". All these terms are used in different communities, and mean some variation of "automatic strategy to construct a term under certain constraints". Here, however, we use the following definitions:

- The Solver effect is used for integration with external solvers via IO; we believe should suffice for the user to write, e.g. `SMT`, and handlers should implement internal strategies to choose between the different theories supported by solvers. If the black-box approach to solvers presents itself a challenge, a specialisation of the handler is possible, e.g. `operation { params } via Z3.QF-UFLIA` [17].
- The Proof Search effect is used for library-level algorithms; users may choose to implement their own algorithms using a limited set of meta-programming[3] constructs that are handled at top-level[4].
- The Tactics effect is used for strategies that simplify the goal at least one step, and may not complete all proof goals. This style is mainly used in the proof assistant Coq. While we do not foresee using as many as many tactics implemented as in Coq, we believe tactics such as `eauto`, `rewrite`, `unfold` are useful to users of all levels of expertise.
- Lastly, the Error effect is used for feedback to the user, since any of the strategies may fail. Error has two operations, `throw` and `trace`: the former notifies the user that a strategy has failed, while the latter registers[5] completed sub-goals during the strategy's attempt to complete the proof.

[2]Our tool's name is a play on the assistant tools colloquially called "wizards". There is no consensus of what a wizard is or what exactly the tasks are it is supposed assist with. Wizards seem to be used mostly for multiple-step and/or configuration features, however. We went for the name "witch" to align it to the idea of assistant tools, while dodging the overloaded, yet nebulous terminology.

[3]By meta-programming, we mean "code that manipulates itself", and not "programming that happens in a higher level of abstraction". For dependently typed programming languages, the usual term is reflection. However, we prefer not use reflection since it has another meaning in terms of effectful computations.

[4]The meta-programming constructs are operations of the Typechecker effect whose handlers are not available for the user.

[5]Internally, the Typechecker effect should have a tree that stores all currently saved traces.

Witch's Architecture. A *witch* sits between the typechecker and the interactive mode of a proof assistant. It uses meta-programming constructs provided by the typechecker and notifies the interactive mode about whether the current attempted strategy was successful. The user may manipulate proof handlers by composing or writing their own.

Downsides of our Approach. There are two main styles for writing proofs within a proof assistant: external verification and internal verification. The latter encourages code that is *correct-by-construction*, meaning only valid states can be encoded and values carry all information necessary to make proofs trivial; while the former encourages writing code as usual, and reconstructing information needed for proofs only when necessary. Witch's primarily focus is on external verification. We believe that newcomers will struggle to write proof-aware code, since it departs strongly from mainstream programming, even within the functional style. We expect users to write their code as they normally would do and then prove necessary properties. Witch does not support writing correct-by-construction code. As proofs are erased during compilation-time in Juvix, Witch could be extended to be indexed by proof objects and parameterised over any type, such as `∀ {A: Type} (x y: A) →`  `(x * y ≡ y * x)` .

3 TECHNICAL EFFORT REQUIRED AND PROTOTYPE IN AGDA

Although Witch can be implemented as a library, it requires certain features from the proof assistant. The most important feature is type-level meta-programming: Witch requires that the proof assistant can manipulate its own proof terms, generate new variables, holes, goals, among others, and manipulate terms and types in a type-safe fashion. Secondly, the proof assistant should provide an interface to the outside world while type checking It is not essential, its absence means that no external solver can be integrated, diminishing the value of the tool. Lastly, the proof assistant must support algebraic effects and handlers. If there is no built-in solution, a library-level solution is possible – as we did in Agda. However, a solid implementation may not be trivial to achieve.

In Agda, we translate effect handling into optimised continuation-passing style that inlines all applications. This strategy can also be implemented at the compiler-level, if necessary. There are many strategies to efficiently implement algebraic effects and handlers, but we opted for the one that seemed the most straight-forward method to implement it in Agda.

Why Agda? Agda provides higher-level meta-programming constructs, and its coding style is a reminiscent of the ML family of functional programming languages. Agda's meta-programming constructs are built-in and a monadic interface is available under the module `TC` within the standard library.

4 RELATED AND FUTURE WORK

4.1 Comparison with Similar Approaches

Andromeda proof assistant. As mentioned, this project was inspired by the Andromeda proof assistant [5, 6]. It follows the LCF tradition of theorem provers, where there exist two distinct languages. One habits a safe nucleus that verifies hypotheses, on the other hand, the second hand comprises a polymorphic λ calculus where users implement their theories and relay the verification to the nucleus.

Meta-F.* F* is a dependently typed language with (monadic) effects and liquid types. For tactics, F* uses a library called Meta-F* [16], where tactics are considered an effect that uses meta-programming provided by the typechecker to manipulate proof objects. Agda's Witch implementation is heavily inspired by Meta-F*.

Mtac and Cybele. The idea of proof strategies happening within an effectful structure was first introduced in Cybele [10] and later extended into Mtac [25]. Mtac extends Coq's powerful main language, providing the user with a dependently-typed tactic language.

4.2 Improving Witch Further

While it is hard to say whether we have achieved our goals before user evaluations, Witch is a first step towards a future where proof assistants are friendlier to users, particularly newcomers. The present work is in its infancy, and there are many open problems regarding Witch's ability to guide newcomers towards correct-by-construction software, which constitutes the main improvement we intend for Witch. We hypothesise possibility of inserting into the source code a *proof sketch* that contains sub-goals of interest for the user and guides the typechecker to rebuild the derivation tree faster. This change would, therefore, relocate Witch to an assistant that lives solely within the interactive mode – as opposed to a framework for proof automation in user code. The implementation of such feature may present itself as a challenge, however.

In parallel with the usability work, we plan to develop a formalisation of Witch as a dependent typed language with effectful, dependently typed operations and effect handler, based on previous work on fibred algebraic effects [1].

REFERENCES

[1] Danel Ahman. 2018. Handling fibred algebraic effects. *Proc. ACM Program. Lang.* 2, POPL (2018), 7:1–7:29. https://doi.org/10.1145/3158095

[2] Danel Ahman and Andrej Bauer. 2020. Runners in Action. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science)*, Peter Müller (Ed.), Vol. 12075. Springer, 29–55. https://doi.org/10.1007/978-3-030-44914-8_2

[3] Pınar Anapa and Hatice Şamkar. 2010. Investigation of undergraduate students' perceptions of mathematical proof. *Procedia - Social and Behavioral Sciences* 2 (2010), 2700–2706. Issue 2. https://doi.org/10.1016/j.sbspro.2010.03.399

[4] Danil Annenkov and Bas Spitters. 2019. Towards a Smart Contract Verification Framework in Coq. *CoRR* abs/1907.10674 (2019). arXiv:1907.10674 http://arxiv.org/abs/1907.10674

[5] Andrej Bauer, Gaëtan Gilbert, Philipp G Haselwarter, Matija Pretnar, and Christopher A Stone. 2018. Design and Implementation of the Andromeda Proof Assistant. *CoRR* abs/1802.06217 (2018). http://arxiv.org/abs/1802.06217

[6] Andrej Bauer, Philipp G. Haselwarter, and Anja Petkovic. 2020. Equality Checking for General Type Theories in Andromeda 2. In *Mathematical Software - ICMS 2020 - 7th International Conference, Braunschweig, Germany, July 13-16, 2020, Proceedings (Lecture Notes in Computer Science)*, Anna Maria Bigatti, Jacques Carette, James H. Davenport, Michael Joswig, and Timo de Wolff (Eds.), Vol. 12097. Springer, 253–259. https://doi.org/10.1007/978-3-030-52200-1_25

[7] Andrej Bauer and Matija Pretnar. 2012. Programming with Algebraic Effects and Handlers. (3 2012). https://doi.org/10.1016/j.jlamp.2014.02.001

[8] Yves Bertot. 2006. Coq in a Hurry. *CoRR* abs/cs/0603118 (2006). arXiv:cs/0603118 http://arxiv.org/abs/cs/0603118

[9] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *J. Funct. Program.* 30 (2020), e8. https://doi.org/10.1017/S0956796820000027

[10] Guillaume Claret, Lourdes Del Carmen González Huesca, Yann Régis-Gianas, and Beta Ziliani. 2013. Lightweight proof by reflection using a posteriori simulation of effectful computation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7998 LNCS, 67–83. https://doi.org/10.1007/978-3-642-39634-2_8

[11] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo bee doo bee doo. *J. Funct. Program.* 30 (2020), e9. https://doi.org/10.1017/S0956796820000039

[12] Alexander John Faithfull, Jesper Bengtson, Enrico Tassi, and Carst Tankink. 2018. Coqoon - An IDE for interactive proof development in Coq. *Int. J. Softw. Tools Technol. Transf.* 20, 2 (2018), 125–137. https://doi.org/10.1007/s10009-017-0457-2

[13] Maria Knobelsdorf and Christiane Frede. 2016. Analyzing student practices in theory of computation in light of distributed cognition theory. *ICER 2016 - Proceedings of the 2016 ACM Conference on International Computing Education Research*, 73–82. https://doi.org/10.1145/2960310.2960331

[14] Maria Knobelsdorf, Christiane Frede, Sebastian Böhne, and Christoph Kreitz. 2017. Theorem provers as a learning tool in theory of computation. *ICER 2017 - Proceedings of the 2017 ACM Conference on International Computing Education Research*, 83–92. https://doi.org/10.1145/3105726.3106184

[15] Daan Leijen and Daan. 2017. Structured asynchrony with algebraic effects. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development - TyDe 2017*, 16–29. https://doi.org/10.1145/3122975.3122977

[16] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof

Automation with SMT, Tactics, and Metaprograms. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11423 LNCS, 30–59. https://doi.org/10.1007/978-3-030-17184-1_2

[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT Solver. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4963 LNCS, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[18] Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, Andrew Kennedy and Amal Ahmed (Eds.). ACM, 1–2. https://doi.org/10.1145/1481861.1481862

[19] Nicolas Peltier and Mnacho Echenim. 2017. The Binomial Pricing Model in Finance: A Formalization in Isabelle. In *CADE 26 (26th International Conference on Automated Deduction)*, Leonardo de Moura (Ed.), Vol. 10395. Springer, Gothenburg, Sweden, 546–562. https://hal.archives-ouvertes.fr/hal-01562944

[20] Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied Categorical Structures* (2003). https://doi.org/10.1023/A:1023064908962

[21] Gordon D. Plotkin and Matija Pretnar. 2013. Handling algebraic effects. *Logical Methods in Computer Science* (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

[22] Kenneth Roe and Scott F. Smith. 2016. CoqPIE: An IDE Aimed at Improving Proof Development Productivity - (Rough Diamond). In *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings (Lecture Notes in Computer Science)*, Jasmin Christian Blanchette and Stephan Merz (Eds.), Vol. 9807. Springer, 491–499. https://doi.org/10.1007/978-3-319-43144-4_32

[23] Freek Wiedijk. 2003. Comparing mathematical provers. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2594 (2003), 188–202. https://doi.org/10.1007/3-540-36469-2_15

[24] Artem Yushkovskiy and Stavros Tripakis. 2018. Comparison of Two Theorem Provers: Isabelle/HOL and Coq. *CoRR* abs/1808.09701 (2018). arXiv:1808.09701 http://arxiv.org/abs/1808.09701

[25] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2015. Mtac: A monad for typed tactic programming in Coq. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000118

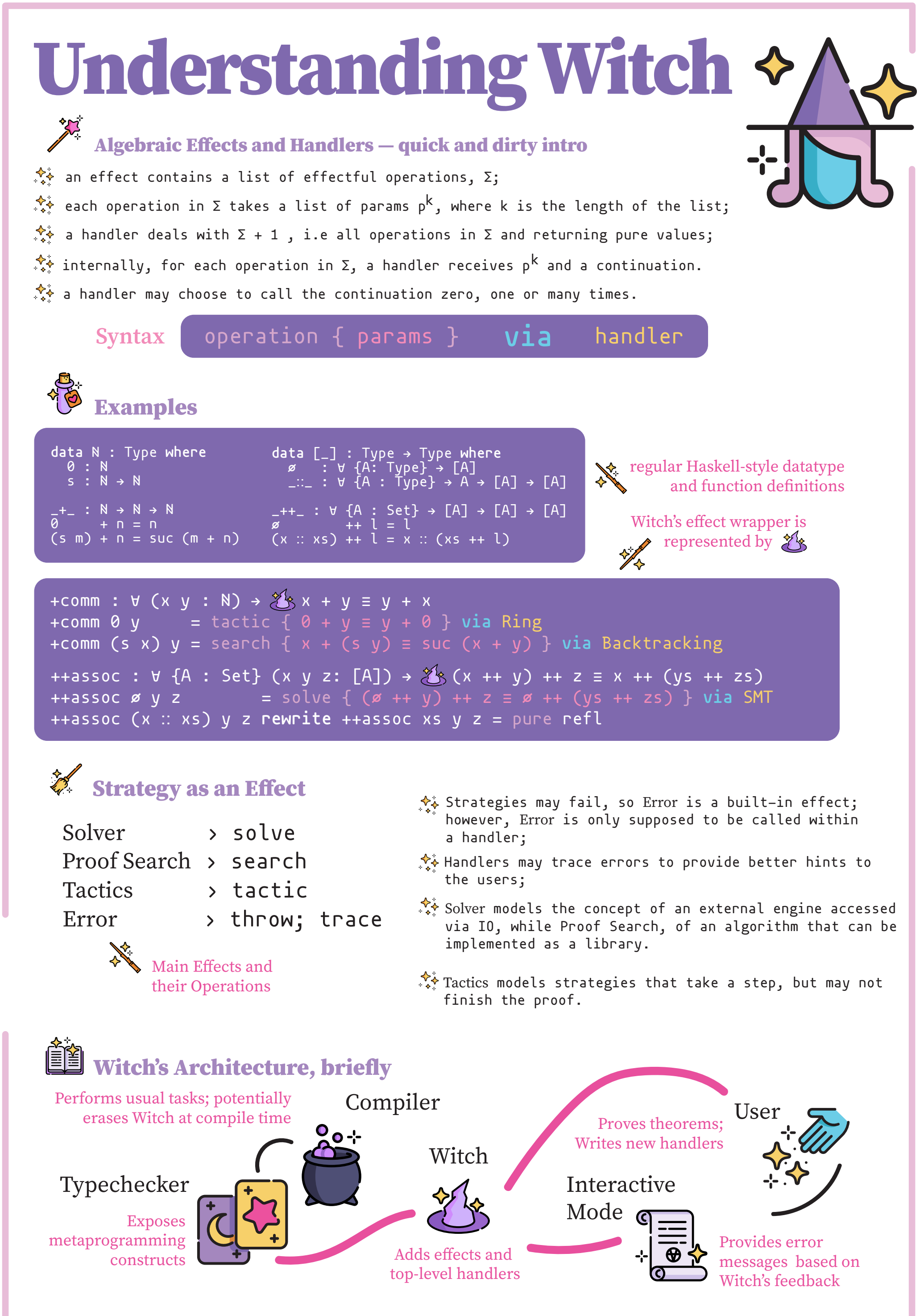

Fig. 1. Summary of Witch pertaining to strategies, examples and the architecture.