

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 many 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), 6 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 VERIFICATION FOR THE MASSES

The usability of proof assistants such as Coq, Agda, Isabelle/HOL and Twelf 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, and their programming style is sharply different from mainstream programming and even functional programming. Granted, they come with their own integrated development environment (IDE), 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.

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. Many Coq learning materials 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 university mathematics courses; and (b) proof by hand and automated proof employ fundamentally

Author’s address:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.

Manuscript submitted to ACM

Manuscript submitted to ACM

different paradigms, where the latter requires axioms and formulations to be encoded explicitly, commonly known as a “pedantic proof style”.

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.

In our approach, we propose an effects and handlers view of such proofs, based on prior work developed on the Andromeda proof assistant. 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, Eff, Frank and Effekt, 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 Juvix¹ 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:

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

- (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 minimal prototype in Agda that validates the feasibility of such system (Section 3);
- (3) a short technical account of the effort required to implement such a user interface for proof assistants (Section 3.1).

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 ¹citation, and their handlers were introduced later on by ²citation. 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.

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`.
- The Proof Search effect is used for library-level algorithms; users may choose to implement their own algorithms using a limited set of meta-programming³ constructs that are handled at top-level⁴.
- 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`, `??`, `??` are useful to users of all levels of expertise.

²Our 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 to 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.

³By 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 to use reflection since it has another meaning in terms of effectful computations.

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

- 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 registers⁵ completed sub-goals during the strategy’s attempt to complete the proof.

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 $\{A: \text{Type}\} \rightarrow x\ y: A \rightarrow \text{🧙} (x * y \equiv y * x)$.

3 AGDA’S WITCH

Why Agda? Agda provides higher-level meta-programming constructs, and its coding style is a reminiscent of the ML family of functional programming languages.

3.1 Technical Effort Required

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.

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. It follows the LCF tradition of theorem provers, where there exist two distinct languages. One inhabits 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.

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

*Meta-F**. F^* is a dependently typed language with (monadic) effects and liquid types. For tactics, F^* uses a library called *Meta-F**, 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* and later extended into *Mtac*. *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.

Understanding Witch



Algebraic Effects and Handlers — quick and dirty intro

- ✦ an effect contains a list of effectful operations, Σ ;
- ✦ each operation in Σ takes a list of params p^k , where k is the length of the list;
- ✦ a handler deals with $\Sigma + 1$, i.e all operations in Σ and returning pure values;
- ✦ internally, for each operation in Σ , a handler receives p^k and a continuation.
- ✦ a handler may choose to call the continuation zero, one or many times.

Syntax `operation { params } via handler`



Examples

```
data N : Type where
  0 : N
  s : N → N

data [_] : Type → Type where
  ∅ : ∀ {A: Type} → [A]
  _:: : ∀ {A : Type} → A → [A] → [A]
```

✦ regular Haskell-style datatype and function definitions

✦ Witch's effect wrapper is represented by

```
+comm : ∀ (x y : N) → ✦ x + y ≡ y + x
+comm 0 y = tactic { 0 + y ≡ y + 0 } via Ring
+comm (s x) y = search { x + (s y) ≡ suc (x + y) } via Backtracking

++assoc : ∀ {A : Set} (x y z : [A]) → ✦ (x ++ y) ++ z ≡ x ++ (ys ++ zs)
++assoc ∅ y z = solve { (∅ ++ y) ++ z ≡ ∅ ++ (ys ++ zs) } via SMT
++assoc (x :: xs) y z rewrite ++assoc xs y z = pure refl
```



Strategy as an Effect

```
Solver      > solve
Proof Search > search
Tactics     > tactic
Error       > throw; trace
```



Main Effects and their Operations

- ✦ Strategies may fail, so Error is a built-in effect; however, Error is only supposed to be called within a handler;
- ✦ Handlers may trace errors to provide better hints to the users;
- ✦ Solver models the concept of an external engine accessed via IO, while Proof Search, of an algorithm that can be implemented as a library.
- ✦ Tactics models strategies that take a step, but may not finish the proof.



Witch's Architecture, briefly

