A Living Review on the Usability of Interactive Theorem Provers

A thesis submitted in fulfilment of the requirements for the degree of Bachelor of Science.

Sam Nolan

Computer Science Undergraduate.

School of Computing Technologies

STEM College
Royal Melbourne Institute of Technology Melbourne, Victoria, Australia
School of Science.
College of Science, Engineering and Health.

October 2021

Declaration

I certify that except where due acknowledgment has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Signed: Sam Nolan

Date: 28 October 2021

Acknowledgments

I would like to acknowledge my supervisor Maria Spichkova for her guidance and high expectations for this project. I would also like to thank Flora Salim for inspiring me to further invest myself into research and never stopping in opening doors for me.

Abstract

Interactive Theorem Provers (ITPs) are tools that allows a user to both prove mathematical theorems, and also verify correctness properties of systems. Through proving sotware correct, ITPs reduce the amount of errors and give a high level of assurance of correctness. However, although the field is growing, the adoption of ITPs in mathematics and software development is far from widespread. It's been suggested that the reason for this is that ITPs are particulary difficult to use, and this could be a major reason why there is not a larger adoption of ITPs. This thesis attempts to uncover possible usability issues through a systematic literature review, and finds that the field is greatly lacking in empirical research. This thesis then contributes an in depth investigation to three usability issues, small mathematical scopes of ITPs support for counterexample generators, and math notation support. The results are presented in a living review, which updates semi-automatically to reflect the current state of the field. The living review doubles as a decision making tool for mathematicians, as some ITPs are better for different fields of mathematics, and tracks progress towards resolving this issue. Due to the updating nature of this review, a copy of this thesis can be found in html form at https://samnolan.me/thesis.

Contents

1 Introduction

This is a thesis about **Interactive Theorem Provers**, what they are, why they might be difficult to use, and whether you should explore using them in your next project.

An **Interactive Theorem Prover** (ITP) or **Proof Assistant** is a piece of software that helps a user prove mathematical theorems, or equivalently, prove correctness properties about software. Some of the more well known examples of ITPs include Coq and Isabelle.

An ITP is often used by either a mathematician or an engineer. From the side of a mathematician, ITPs allow users to specify a proposition that they would like to prove, and then specify to the ITP how to construct that proof of the proposition. The ITP can then check whether the given proof is valid. Usually, when a proof in mathematics is made, it must be checked for errors before being accepted. The use of an ITP automates this verification, and means that the mathematician only needs to trust that the ITP was implemented correctly in order to trust any proof that's been created from it. By formalizing mathematics within ITPs, mathematicians can create libraries of formal trustworthy proofs. This allows computers to aid with the proof process, and helps improve the trustworthiness of proofs.

ITPs have been used for formalizing mathematics in the past. For instance, the QED Manifesto [85] has proposed using ITPs to generate computer checkable proofs for mathematics. Further, they have been used to prove theorems such as the four colour theorem in Coq [32] and the Kepler Conjecture in a mixture of HOL Light and Isabelle [40].

An ITP can however also be used by a software engineer to create **certified software**. **Certified software** is software that's been proven to operate correctly, meaning that software is proven to operate as according to the specification. When developing software, eradicating bugs is usually done through procedures such as code reviews or software testing. However, these will often not identify 100% of the errors. ITPs allow you to prove that the implementation matches the specification 100%. Proving the software correct can give the assurance that the software implementation matches the specification. This is a very high level of assurance in terms of quality, and dramatically reduces the sources of errors in software. ITPs have been used to ensure correctness of code bases that require a large amount of trust, such as cryptographic libraries [47], microkernels [53] and compilers [54].

In computer science, often correctness proofs of algorithms (For example, Dijkstra's algorithm [60] in an undergraduate context) are described and proved to be correct on pen and paper. The use of an ITP is analogous to this type of activity. Except that an ITP aids the user in developing the proof, offering hints and/or automating portions of the proof along the way.

The task of creating verified software is split into two steps: first specification and then verification.

During specification, the user specifies what it is that they would like to prove, for example, that Dijkstra's algorithm always finds the shortest path between one node and all other nodes in a weighted graph, assuming positive weights. In this step, the user would create a specification for what is Dijkstra's algorithm, graphs, and shortest paths. Then state that Dijkstra's algorithm finds the shortest path. This step is far from trivial, as developing a precise form of specification for software requires having a detailed understanding of how exactly you want it to work. Often within the verification process, a user might realise that the specification itself is in error in some way, and might go back to correct the specification during the process.

This specification leaves the user with a **proof obligation**. A proof obligation is a onus on the user to prove that the specification of the software is correct. Then, during verification, it is then up to the user to provide to the ITP the reasoning as to why this proposition is correct. This can be done in several ways, sometimes through the use of automated software, or manipulating the proof by pointing and clicking, or writing down a **proof script** that describes the steps made to prove the theorem. Often this involves breaking down one proof obligation into many other simpler proof obligations that can be solved individually.

The user and the ITP work together until they have specified a proof of the statement they wish to claim. This is an iterative process where the user may specify what they want to prove, then

attempt to prove the proposition, find out the proposition or the specification that they are proving has a typo or error, fix the specification, continue developing the proof, realise the approach of the proof won't work, backtrack and start from an earlier state, realise they need to prove some auxiliary lemma, create the lemma and prove that, come back to the original proof etc. The ITP helps the user in the process by providing counterexamples for propositions that may be incorrectly specified, automatically solving some components of the proof, offering libraries of mathematical results and informing the user of any errors in the reasoning of the proof. Once that proof is made, assuming the ITP is functioning correctly and the specification is correct, the user can be assured that the proposition is true.

ITPs are often implemented by making use of a programming language, particularly statically typed functional programming languages. A functional programming language is a language that creates problems from the composition and definition of functions, such as Haskell or OCaml. Often a goal in creating statically typed functional programming languages is to create languages where its difficult or impossible to make a certain class of errors. For instance, Rust [57] is designed to allow systems level programming that's protected from memory errors, and Elm [27] is designed to create web programs that do not have runtime errors. This means that possible software implementation errors are caught as they are being typed checked, very early in the development process. ITPs can also be seen as an extension of this by having type systems and features that allow the user to go as far as proving arbitrary correctness properties about the software.

More widespread use of ITPs is valuable as it allows for errors to be caught much earlier in the development process. ITPs allow for errors in specification or implementation to be found as the software as the software is being developed. This prevents errors from arising when testing or even worse, in production software. Such errors can be much more expensive to fix [58].

1.1 Formal Methods

ITPs are not the only way to specify and verify software. They belong to a class of techniques named **Formal Methods**.

In essence, formal methods attempts to improve the process that users can prove the correctness of their software systems. Use of formal methods can be done without any tools at all, by simply proving properties manually, with paper and pencil, such as the Dijkstra example above [1].

However, when proving the correctness of larger systems, proving properties by hand quickly becomes intractable. Computers and tools have aided people in providing larger correction proofs for software systems. The tools used in Formal Methods can be roughly divided into three categories, Model Checkers, Automated Theorem Provers and Interactive Theorem Provers.

These three techniques are a trade off in three features. You can pick two but not all three.

Features of Formal Methods Tools:

Automation: Whether finding a proof is fully automated. That is, the user does not need to specify a proof manually for the proposition, the system simply attempts to find one automatically.

Termination: Whether the tool terminates in a practical amount of time when attempting to find a proof. For instance, does not have the possibility of running for several days when attempting to prove a proposition.

Scope: Whether the tool is able to prove any arbitrary property about a system. The properties that can be proven are not restricted to a subset of possible properties.

Formal Methods Tools:

Model Checkers are fully automated and if the specification is not particularly large, can find solutions in a reasonable amount of time. However, Model checkers can do this by restricting the scope of the systems that they can prove. They allow for a specification for a system in a (usually finite) state machine or automaton, and can prove properties about this state machine. This means that you can only prove with a model checker some simplification of the actual system, and often

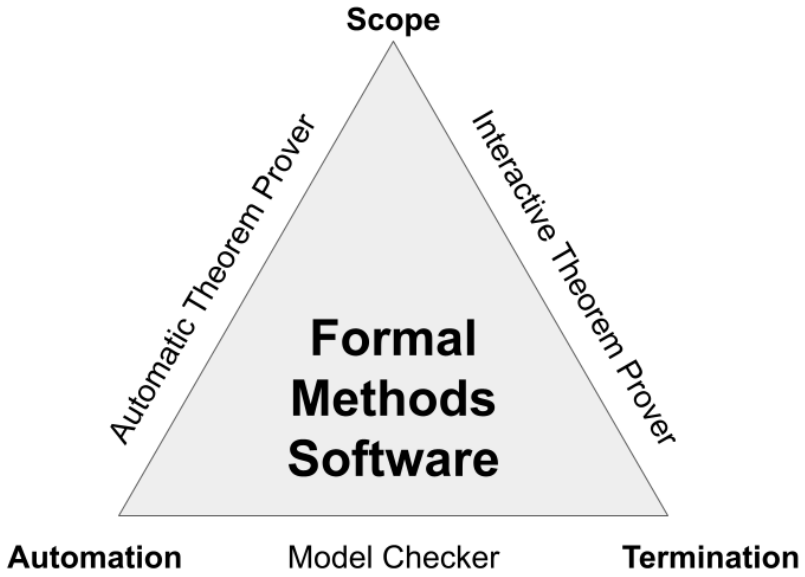

Figure 1: Three categories of Formal Methods

cannot prove more complicated properties such as those with numbers. This means that not all properties about the system can be verified.

Automated Theorem Provers (ATPs) are fully automated and can prove arbitrary theorems, however may not terminate in reasonable time. For larger systems or more complicated theorems, they may run forever and never identify a proof or disproof for the proposition.

Interactive Theorem Provers terminate in reasonable time and can prove arbitrary theorems. However, they are not fully automated, and require the user's input to guide the proof of the theorem. This is a major limitation as interacting with a human is often relatively slow and expensive. Especially if the propositions that are being proven are more trivial and could be solved quickly with an ATP.

The distinction between ATPs and ITPs is however not clear cut. ATPs can often include minor user interaction in order to correct it's path and find a proof. And ITPs often have automatic features and can even call external ATPs to help automatically prove goals.

ITPs were chosen for this investigation due their usage in creating fully certified software [77]. Model checkers can be used to check important properties about software, but can't get the last mile to make it fully certified due to not allowing the user to prove all the propositions that might be relevant to the code. As reliance on technology is only getting greater, it's important to consider making technology as trustworthy as possible, particularly if the software is safety/security-critical.

That being said, ATPs are often used beside ITPs, where an ITP might call an ATP to prove a proposition or discover a counterexample automatically. The primary goal in the development of ATPs is the speed at which it can prove a variety of theorems. As of such, development of ATPs directly influences development of ITPs by improving some of the automation related tools in ITPs.

1.2 ITP Interaction Paradigms

ITPs can go about proving and specifying properties about software in different ways. This section outlines a short history of interaction paradigms with ITPs, and possible developments.

Direct Manipulation ITPs such as KeY work by editing proof objects until the obligations have been resolved. These provers often have issues with tedious interactions, and work has even been done add textual elements to KeY [16]. The development of interfaces to Direct Manipulation provers often differs from textual ones.

Textual ITPs such as HOL, Isabelle, Coq and Matita work by writing a proof script that attempts to prove a proposition. Interacting with textual ITPs often involves a very simple read-evaluate-print-loop (REPL) for their interfaces. This is very similar to the example we went through in section Section 1.3. One very stark example of this is HOL-Light, which the user interacts with by opening up the OCaml REPL (a general purpose ML based functional programming language) and loading the HOL library. All OCaml is available alongside the HOL library. Although this is rather primitive, modern ITP interfaces such as Isabelle/jEdit and CoqIDE usually offer only a small layer of abstraction over a REPL for their own languages.

These interfaces have two main windows, the first has code and the second has proof state. The code can be evaluated up to a certain point, and the output from the REPL in terms of proof state are printed in the second window. The only major difference between this and a standard REPL is that the user can rewind to evaluate up to a previous line. This simple style of interface has consequences for usability. In particular, if any error is found either in proof or in syntax, execution stops until that error is resolved. Further, for larger projects, it can take a very long time for systems to recompile. It also means that the user can only reference things that have already been declared (it has to be a single pass). This is particularly an issue when automated tactics attempt to use lemmas above them to find solutions to theorems (such as Isabelle). This means that simply changing the order of lemmas in an Isabelle document, even if they never reference lemmas that are below them, could cause a lemma that was proven before to become unproven.

Developments in IDEs to allow asynchronous interfaces, reloading only parts needed and loading proofs out of order have been introduces to fix this problem. They are called "Prover IDEs", with two examples being Isabelle/PIDE [82] and Coq/PIDE [10]. These hopefully will resolve some of the issues cited above.

Although we have examples of large projects undertaken with ITPs, optimal interaction paradigms are still up for debate, and several novel interaction paradigms have surfaced. Including proving theorems and writing tactics with diagrams [38, 55, 76], or providing agent based interfaces [46].

We now move into the usability problems and solutions found in ITPs.

1.3 Using an ITP

To explain the typical terms and issues related to ITPs, let us present an small toy example of using an ITP using pseudocode syntax.

Our pseudocode syntax is based on textual ITPs such as Isabelle, Coq and HOL. The syntax is simplified to get the basic concepts of ITPs across without too many of the technical details.

To prove a property with an ITP, the user must start by having a goal that they wish to prove. You then manipulate and decompose that goal into simpler subgoals, until they've proven the proposition they wish to prove.

We start with a function:

$$f(x) = \begin{cases} f(x-1) + x, & x > 0 \\ 0, & x = 0 \end{cases}$$

This is the triangle number function. It adds a number to every number below that number up until 0. For instance, $f(5) = 5 + 4 + 3 + 2 + 1 + 0 = 15$.

However, there is a quicker way to calculate the triangle number, and that is that is that $f(x) = \frac{x(x+1)}{2}$.

This proof is an elementary induction proof. But we shall demonstrate that this statement is true.

We would like to prove that:

$$f(x) = \frac{x(x+1)}{2}$$

Textual ITPs are similar to interactive programming languages, such as R and Bash, where the main interaction is through a Read Evaluate Print Loop or REPL. You start by writing a line of code, and the ITP will return the state. These lines of code manipulate the state until the statement has been prover. After the proof, the user is left with a proof script, which is a listing of all the code used to prove the proposition.

To start with our proof, we first have to specify the property that we wish to prove. In pseudocode, we write Listing 1 into our ITP. This statement says that the user would like to `Prove` the statement `forall x, f(x) = x * (x + 1) / 2`. `Prove` here is a keyword that starts the proof. Everything between the `:` and the `.` represent the statement they wish to prove.

Listing 1: User Input: Statement of the proposition to prove

```
Prove: forall x, f(x) = x * (x + 1) / 2.
```

After writing this statement, the ITP will print the display shown in Listing 2. This will often appear in a window in the ITP interface, or if it is a command line prover, it will print it to console.

The state is separated into two sections, everything above the `---` is an assumption, that is, what we have assumed to be true. We can have multiple assumptions, but in this case, there are none. Then the statement below the `---` is the **goal**. This is the statement to prove. When starting a proof, whatever statement the user wishes to prove becomes the first goal, but both the assumptions and the goal will change as the user progress in the proof. It's also possible to split the goal into multiple subgoals as the proof is being developed.

Listing 2: ITP Output: Starting state

```

---
forall x, f(x) = x * (x + 1) / 2
```

Now we must give a series of **tactics** to the ITP to prove this proposition. A tactic is like a command that is sent to the ITP in order to prove the proposition that you wish to prove. Tactics are always typed, and sometimes they fail to work in situations where their usage would be invalid. These tactics form a language that you can use to specify your proof.

It should be noted that while attempting to prove a proposition, the user may wish to attempt to prove a goal or subgoal automatically. Most ITPs have the ability to automatically prove simple propositions, often by giving the `auto` tactic to the prover. If this succeeds, then the user is done and the proposition is proven. Otherwise, the user must continue to explore proof options. Furthermore, the ITP can assist the user by offering counterexamples to why the proposition might not actually be true. We will assume that the statement cannot be proven automatically.

This particular proof is often used as an introductory induction proof, so induction would be a good start to solving this. Entering the pseudocode in Listing 3 performs induction on the variable x. To perform induction, we must prove the base case, and then prove the inductive case. The ITP will ask to prove them one at a time, starting with the base case. After the tactic is executed, the state of the ITP will be modified, and display the state in Listing 4.

The state has now been split up into two *subgoals*. One for the base case and one for the induction case, which is labeled as (1/2). In our pseudocode ITP, all the tactics that we write will manipulate

the first goal only, but the ITP is indicating that there still is a second goal that needs to be proven after this first one. This means that the theorem prover is asking for a proof of the base case, that is, that the statement is true when $x = 0$.

Listing 3: Running the induction tactic

```
induction x
```

Listing 4: State after the induction tactic

```
---
f(0) = 0 * (0 + 1) / 2
(1/2)
---
f(x) = x * (x + 1) / 2 -> f(x + 1) = (x + 1) * (x + 2) / 2
(2/2)
```

Notice that the tactic modifies the state of the ITP. Only tactics that are valid at the time are allowed to be used, ensuring that all proof steps are valid and construct a correct proof.

The base case is very easy to solve, as simply evaluating the function on both sides ($f(0) = 0$ and $\frac{0 \cdot (0+1)}{2} = 0$) gives 0.

To evaluate this, we use the `simplify` tactic in Listing 5. This tactic attempts to try a list of rules that the prover guesses will simplify the current statement. In our pseudocode ITP, this includes evaluating statements with constants. The result is as we expect and shown in Listing 6. Indicating that after the simplification, both sides are equal to each other.

Listing 5: Running the simplify tactic

```
simplify
```

Listing 6: State after running the simplify tactic

```
---
0=0
(1/2)
---
f(x) = x * (x + 1) / 2 -> f(x + 1) = (x + 1) * (x + 2) / 2
(2/2)
---
```

Now the goal is to prove that `0=0`. This is trivially true because equality is reflexive. As of such, we can prove the current goal by indicating that it's reflexive. We can use the `reflexivity` tactic to prove any goal that is true because of reflexivity. This tactic is executed in Listing 7.

After we have solved the first goal, the base case, the pseudo-ITP is now asking us to prove the inductive case in Listing 8. The current goal is now to prove the inductive case. As it is with induction, the inductive case allows us to assume that the original proposition is true for any x, and that we need to prove that it is the case for x + 1. Therefore, the statement for proposition for (x + 1) is our goal.

Listing 7: Running the reflexivity tactic

```
reflexivity
```

Listing 8: State after running the reflexivity tactic

```
f(x) = x * (x + 1) / 2
---
f(x + 1) = (x + 1) * (x + 2) / 2
```

The first step would be to replace `f(x + 1)` with it's definition. This can be done with the `unfold` tactic, as shown in Listing 9. This tactic replaces a function with it's definition. This replacement is shown in Listing 10

Listing 9: Running the unfold tactic

```
unfold f
```

Listing 10: State after running the reflexivity tactic

```
f(x) = x * (x + 1) / 2
---
f(x) + (x + 1) = (x + 1) * (x + 2) / 2
```

Our current assumption and the goal are equivalent statements in different forms. We now have to re-arrange the goal to make it equal to the assumption. This might involve several tactics to manipulate the state of the equation. We will for the sake of brevity written these tactics in English like code in Listing 11. The intermediate goals are shown in Listing 12 to show the progress towards the desired goal, and the final state is shown in Listing 13.

Listing 11: Running re-arangement tactics

```
expand (x + 1) * (x + 2)
subtract both sides (x + 1)
replace (x + 1) with (2 * (x + 1) / 2)
combine fraction
simplify
factorise (x * x - x)
```

Listing 12: States after running each tactic

```
f(x) + (x + 1) = (x^2 + 3x + 2) / 2
f(x) = (x^2 + 3x + 2) / 2 - (x + 1)
f(x) = (x^2 x + 3x + 2) / 2 - ( 2 * (x + 1) / 2)
f(x) = (x^2 + 3x + 2 - 2 * (x + 1)) / 2
f(x) = (x^2 + x) / 2
f(x) = x * (x + 1) / 2
```

Listing 13: Final state after running each tactic

```
f(x) = x * (x + 1) / 2
---
f(x) = x * (x + 1) / 2
```

Because we know the inductive hypothesis is true, and the goal is exactly the same as the inductive hypothesis, we can simply indicate that we have proven the goal. We do this be the `assumption` tactic, and then close off the proof with the `QED` tactic Listing 14. This then accepts the proof as true Listing 15.

Listing 14: Running the assumption tactic

```
assumption
QED
```

Listing 15: State after running the assumption tactic

```
Proof accepted
```

It should be noted that the tactic we wrote out are akin to deduction rules, however, there is a problem with this approach, and the problem should become clear once we write down all the commands that have been put into the prover.

Listing 16: Final Proof script

```
Prove: f(x) = x * (x - 1) / 2
introduce
induction x
simplify
reflexivity
unfold f
expand (x + 1) * x
subtract both sides x
replace x with (2 * x / 2)
combine fraction
replace ( + x - 2 * x) with ( - x)
factorise (x * x - x)
assumption
QED
```

These proof scripts are very difficult to understand statically, that is it's difficult to formulate the reasoning behind the proof simply by looking at the list of tactics that were used to prove the proposition. Our understanding of the tactic were aided due to our knowledge of the current state, and understanding However, when looking at the script without the context of the state, they are often very difficult to follow. Especially if the scripts are more complicated than this one.

1.4 Cognitive Dimensions

Usability problems with ITPs have been noted in the past. In particular, it was completed as part of Kadoda's PhD thesis in 1999 [49]. In their thesis, Kodada used Cognitive Dimensions of Notation to classify and sort usability issues. With ITPs.

Cognitive Dimensions of Notation is a framework proposed by Green [37], as a way of discussing the design trade-offs of visual programming languages, but has been applied elsewhere for a variety of notations. These dimensions are not an evaluation framework for notations, as often increasing one dimension will also change other dimensions, and different tasks may require different dimensions. For instance, in textual ITPs, dependencies are not shown between theorems, and doing so would increase the Diffuseness of the notation, allowing less to be shown and understood on a single screen. However, debugging why some theorem might fail given a change in other theorems would aid from a more diffuse representation showing the hidden dependencies.

Cognitive Dimensions focus mainly on the way that users understand and work with the meaning of the program. Cognitive Dimensions make an important distinction between difficulties of understanding and working with the notation vs. difficulties with the actual problem itself. Because proving theorems is a very cognitively demanding task, and a lot of that difficulty is inherit to the problem of proving theorems, and no amount of notation will solve it. As of such, we need to focus on ways that the notation can be improved to remove as many difficulties as we can that are not inherit.

Cognitive Dimensions of Notations has been adopted as a way of evaluating the usability of ITPs in Kadoda PhD thesis [50]. In their thesis, they simply use the dimensions as a framework for constructing a questionnaire about possible usability issues with ITPs. We derive our definitions of Cognitive Dimension as applied to ITPs from her thesis.

This thesis will be using Cognitive Dimensions of Notations framework to classify different usability problems. The Cognitive Dimensions of Notations and their definitions are presented:

Abstraction Gradient: Does the ITP offer ways of abstracting components? Abstraction here refers to methods, classes and encapsulation. Green classifies notations as either being abstraction-hating, abstraction-tolerant or abstraction-hungry. An abstraction-hating ITP would be one that forces the user to work with low level constructs often. An abstraction-tolerant ITP would be one that gives some methods for abstraction, but still nevertheless requires constant low level interaction. An abstraction-hungry ITP would offer many methods of abstraction, that could even in the end obscure what is actually happening behind the scenes.

Closeness of Mapping: Closeness of Mapping is how similar the notation is to the problem domain. At some point, a representation of the problem has to be put into notation suitable for the ITP. The easier this is to do the better the closeness of mapping, or how close the proof state is represented vs what the user would expect.

Consistency: Once the user know the basics of an ITP, how much of the rest can be inferred? A notation that is not consistent would require constant lookup of features in the theorem prover. Consistency is particularly important for learning ITPs. Consistency can become an issue when there are a large amount of abstractions.

Error-Proneness: Is it easy to make careless mistakes? A common "careless mistake" in theorem provers is trying to apply a tactic in a textual theorem prover that is not valid for the current proof state.

Diffuseness: How much does it represent a proof relative to the complexity of the proposition proven? This is an easier Cognitive Dimension to measure, and represents the verbosity of the notation. ITPs with high diffuseness often have lower abstractions and are easier to understand, but more difficult to change.

Hard Mental Operations: Are there parts of the notation that require getting out a pencil and paper to understand what it means? The domain of ITPs by it's very nature requires Hard Mental Operations, so it's important to separate the inherit difficulty vs difficulty created by the notation. Hard Mental Operations may arise out of particularly complicated tactics, especially since tactics can be composed together. An ITP with a consistent set of tactics would reduce Hard Mental Operations.

Hidden Dependencies: Are there dependencies in the notation that are not presented? In ITPs and programming languages, it's usually possible to find what a function/lemma references, but is difficult to find what lemmas/functions reference the one we are working in. Furthermore, automation often uses lemmas in the context without indicating at all that it is using them. This makes the issue of hidden dependencies even more difficult. An ITP with low hidden dependencies makes these dependencies between parts of the program explicit

Perceptual cues: Does the notation force the user to make decisions before they have the information they need to make it? Especially for novice users, ITPs need to allow the user to explore different paths for proving a statement. This often represents a premature commitment as the user has to commit to a strategy before evaluating whether that strategy would work. ITPs that offer undo features and allow postponing making decisions require less premature commitment.

Premature commitment: Does the notation force the user to make decisions before they have the information they need to make it? Especially for novice users, ITPs need to allow the user to explore different paths for proving a statement. This often represents a premature commitment as the user has to commit to a strategy before evaluating whether that strategy would work. ITPs that offer undo features and allow postponing making decisions require less premature commitment.

Progressive Evaluation: Does the system give adequate feedback? Error messages, display of current proof state and ability to work with incomplete proofs are all features of progressive evaluation. Getting feedback from the system is absolutely essential for learning the ITPs.

Role Expressiveness: Is it easy to identify what the purpose of a component is? Lack of role

expressiveness, particularly within the proofs of textual ITPs, was one of the main motivations of this study. It is often very difficult on retrospect to identify how the components of a proof relate to each other. An ITP with high Role Expressiveness would make it clear how a lemma or component of a proof contributes to the proof.

Secondary Notation: Are there avenues for comments, colours and representation of the code that helps with comprehension? A Secondary Notation is a way of representing understanding by not changing the actual meaning of the notation. ITPs that offer comments, colours and whitespace grouping help with representing secondary notation.

Viscosity: Is it easy to make a change in the system? ITPs with low abstraction make it difficult to make changes. Sometimes a small difference to what the user is wanting to prove requires a disproportionate change of the proof. ITPs with high viscosity make it difficult to change.

Visibility and Juxtaposability: How easy is to get a particular piece of desired information? How easy is it to compare part of the proof with proofs elsewhere? Sometimes critical information is difficult to obtain when creating or understanding a proof state. A common example is being able to inspect intermediate proof steps. When a proof relies heavily on automation, it is sometimes difficult to understand how the automated tactic managed to get in a particular proof state. Having this information helps understand the proof and how to move forward. ITPs with low visibility make it difficult to find such information. Juxtaposability is showing two parts of the system side by side. This is important as often a proof might only be a refinement of a previous proof, and might need to be understood in context.

1.5 Research Questions

This thesis was inspired by the difficulty in reading static proof scripts. Some provers have already moved to attempt to fix this problem. For instance, Isabelle's Isar language offers a different way of proving propositions that embeds more state, and helps the user understand the proof. It would be therefore valuable to determine what usability issues have been reported, what has been done to fix them, and whether solutions from some ITPs could be used to help other provers.

And finally, the secondary motivation of this thesis is to encourage further use and development of ITPs. So it would be valuable to create a decision tool that helps a user determine which ITP, if any, should be used for a particular project.

Hence, the research questions for this thesis are:

RQ1 *What usability issues and solutions have been mentioned in literature regarding ITPs?*

This research question was chosen to better understand the context of the study. This was answered by a systematic literature review in Section 3.

RQ2 *To what extent to these usability issues exist at the latest versions of ITPs?*

This research question was chosen due to the continual development of ITPs. It is answered by a living review in Section 4.

RQ3 *What, if any, ITP should be used for a specific project?*

Finally, this research question was chosen as being important for the aim of encouraging more development in the field of ITPs and the creation of more verified software. It is answered by a living review in Section 4.

2 Methodology

In this section, we go over the methodology for answering our research questions. We will structure our systematic literature review, and outline our contribution of a separate living review.

The natural method for answering RQ1 is to perform a systematic literature review. This literature review intends to identify and categorize usability issues related to different theorem provers. We outline the methodology for this review in Section 2.1.

Answering RQ2 requires going through the theorem provers and identifying the issues. However, ITPs are continually in development, and any issue that arises could be solved at a future date. As of such, we created a **living review** to answer this question.

A living review is a literature review of a field that updates periodically to reflect the current state of the research. These reviews are often published electronically, such as to a website. The goal of a living review is to ensure that the review never goes stale, and can be used as a reference years to come.

Our review looks through more than just literature, and attempts to directly track progress towards fixing usability problems in ITPs. The methodology for the living review is detailed in Section 2.2. This living review will gather information about ITPs directly semi-automatically in order to keep up to date. This living review is the main contribution in this thesis.

It differentiates itself from a normal review by updating automatically over time, being an interactive software product, and existing as a decision tool for users.

The creation of this living review will answer RQ3 and RQ2, and provide descriptions of the current state of the field for ITPs.

2.1 Systematic Literature Review Methodology

In this section is to outline the methodology for our systematic literature review. The goal is to answer RQ1, and identify usability issues in literature.

To start, A preliminary ad-hoc literature review will be done in order to survey what usability problems exist in ITPs. This preliminary review came from a search for “usability interactive theorem provers" on the ACM digital library and Google Scholar. The review found several papers on the topic. This will be done to better understand the keywords that people use when discussing usability problems. We then attempted to construct a query that would match these papers and also other papers in the field.

The query constructed from this process was as follows:

```
    ("Interactive" OR "Deductive")
AND ("prover" OR "provers" OR "proving" OR "verifier")
AND ("usability" OR "user" OR "users")`
```

This query has a large amount of quotes and exact text. The reason for this is that both in many databases, looking up “prover” will return you papers that include the much more common and less relevant term “prove.” Further, not quoting “usability” has the unfortunate consequence of returning papers that contain the word “use.” As of such, we have a very specific query.

We searched the following databases using this query string:

- Scopus
- DBLP
- Springer Link
- Science Direct
- ACM
- IEEE
- Xplore

From the papers discovered in this way, we went through the abstracts and discerned whether the paper was relevant to the research question.

Our inclusion criteria for the papers inclusion and exclusion criteria in the systematic literature review were the following:

LIC1 A peer review published paper.
LIC2 Notes particular usability issues with theorem provers OR
Offers direct recommendations to the improvement of the usability of interactive theorem provers.
LEC1 The paper is written in a lanugage other than english
LEC2 The paper is of a tutorial, workshop, extended abstract.
LEC3 the item is not peer reviewed.

From the papers that were deemed relevant to the research question, we found papers that cited the papers discovered. That is, we applied forward snowballing [86].

We then tried to discover whether these papers were relevant to the research question, and repeated the process of forward snowballing until there were no more papers discovered.

We then analysed the identified papers to discover:

- The problems and/or solution to usability of interactive theorem provers
- Which theorem prover the issue is relevant to
- Evidence behind issues and proposed solutions

The issues were then categorized by Green's Cognitive Dimensions of Notations [37]. This literature review therefore will answer RQ1.

2.2 Living Review Methodology

The living review has the end goal of determining whether usability issues still exist, and then further offering a tool to help decision about ITPs (RQ3) but also answer whether usability issues still exist in the newest versions of ITPs (RQ2).

The choice of what topics should be covered in the living review were informed by the systematic literature review. All of these options for the scope of the living review will be justified later. We chose to focus on three things:

Mathematical Scope. Does the library implement enough mathematical fundamentals? Often it becomes difficult to work with theorem provers with small libraries, as you may always have to go back to prove trivial things. Math scope is therefore an important usability issue. This living review collects math modules from the theorem provers and then classifies them. This allows for a good comparison between the scope of different ITPs. This is the primary contribution of this thesis.

We further included a discussion of counterexample generators and math notation as a minor contribution on top of this. Counterexample generators are software that attempts to disprove the proposition you are trying to prove by providing a counterexample. It does this so that you can identify if there are any problems with your specification. Counterexample generators therefore help the user better understand the proposition they wish to prove. This living review discusses the support for counterexample generators among ITPs.

Finally, because proving software correct often requires the use of mathematics, often being allowed to use mathematical notation can be useful to make it easier to read and understand the items they are being asked to prove. The review includes a discussion on support for mathematical notation.

The following sections describes the methodology in detail. First, we discuss the choice of Interactive Theorem Provers included within the review in Section 2.2.1. Then, we discuss the methodology used in evaluating the scope of the library in Section 2.2.2. Then, we discuss the methodology for evaluating support for counterexample generators in Section 2.2.3. Finally, we discuss the methodology for evaluating support for math notation in Section 2.2.4.

2.2.1 Choosing Interactive Theorem Provers to Cover

In this section, we discuss the choice of which ITPs to cover as part of the living review.

A reasonable approach to this would be to reference a past review and borrow their scope of ITPs. This was the approach that we took. The newest review in the field is currently [63]. The ITPs in this review include Isabelle, Coq, HOL, Agda, PVS, LEO-II, Watson, Yarrow, Atelier B, Metamath, Twelf, Mizar, RedPRL, JAPE, LEO-II, Getfol, and Z/EVES.

However, we noted a couple of issues with this consideration of ITPs. In particular, it seemed to favour reviews of older ITPs, and didn't include successful ITPs that have come out more recently.

One clearly missing consideration was that of Lean [61], a highly popular ITP that's more often used by mathematicians. Lean is mentioned within the review as a 'newer ITP' but is not considered as part of the review. We chose to add Lean, particularly because of its large community based mathematical library. It would be difficult to make a comparison between mathematical libraries, as we intend to do, without including this one.

Secondly, there were some provers that were included within this review that are currently abandoned. Those two provers were Yarrow, and Watson. These two provers had their last releases in 1999 and 2001 respectively. Considering the rate of development of ITPs, these were removed due to meeting a very conservative definition of being abandoned.

Finally, the last modification that was made was that the survey referenced "HOL" as a theorem prover. HOL is not a single ITP but has several implementations, including HOL4 [78], HOL Light [41], ProofPower [70] and HOL Zero [2]. For the sake of their review, keeping these separate was appropriate as only properties of different theorem provers were discussed, and these properties remained largely the same for each implementation. However, because this living review covers mathematical libraries, which will change depending on the implementation, it will require comparing between implementations. As of such, "HOL" was split into its two most popular interventions, HOL4 and HOL Light. These two were chosen as they have larger mathematical libraries, whereas HOL Zero and ProofPower do not. Including HOL Zero or ProofPower would therefore not add much value to the living review.

This left the following ITPS in scope for the review: ACL2 [51], Agda [64], Atelier B [25], Coq [19], Getfol [33], HOL Light [41], HOL4 [78], Isabelle [68], JAPE [21], LEO-II [17], Lean [61], Metamath [65], Mizar [34], PVS [73], RedPRL [5], Twelf [69], and Z/EVES [74].

2.2.2 Scope of Library

This section reviews the methodology for investigating the scope of libraries within the living review. We have the goal of determining whether mathematical foundations were covered for any given subject area, and by which provers they are covered by.

On a high level, the mathematical libraries for different ITPs will be decomposed into modules, and each of these modules are classified according to which section of mathematics they cover. This way we can identify which ITPs cover which mathematical topics, and make meaningful comparisons between them.

The flowchart shown in Figure 2 has an overview of the methodology to be used used to evaluate the scope of the mathematical libraries.

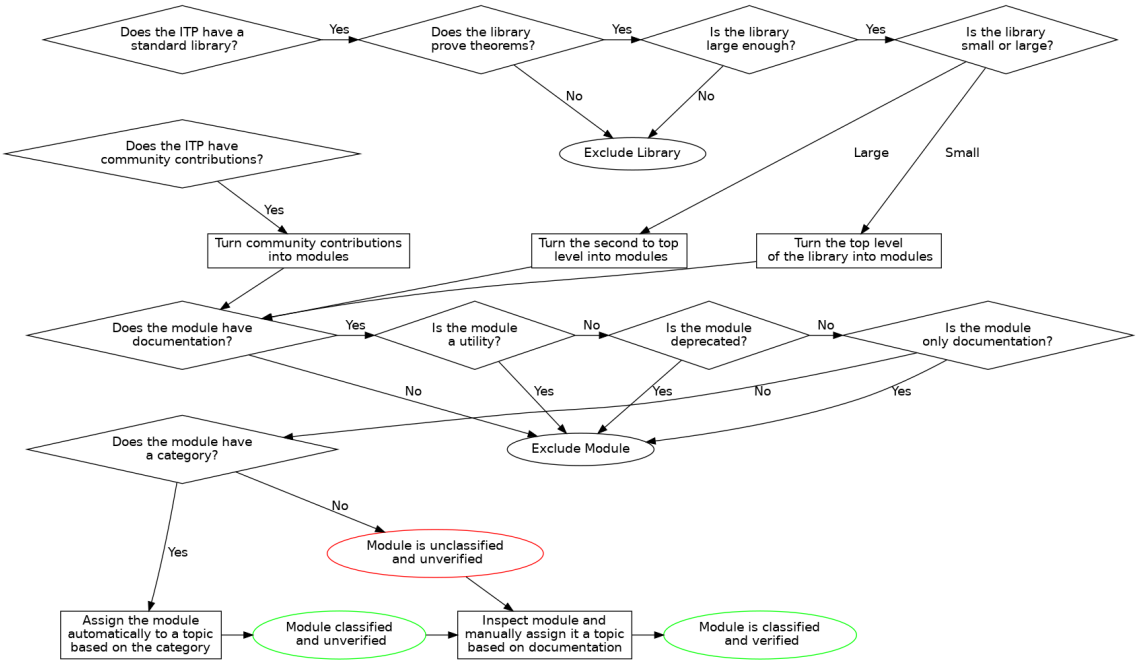

Figure 2: Flow chart for classifying mathematical libraries

2.2.2.1 Step 1: Identifying Mathematical Libraries We start by identifying mathematical libraries. For the sake of this analysis, we also include community contributions, such as package management systems, as part of the scope. As of such, a "library" will refer to both a library that comes along with an ITP (often titled a "standard library") as well as a collection of community contributions, such as package management system.

For each ITP, it will be determined whether they first had a body of code similar to a library. If it did, then it was determined whether the library simply meets the following inclusion criteria:

IC1: The library contains mathematical theorems. Libraries that do not prove theorems are often libraries that are about simple constructions of data types.

IC2: The library contains more than 15000 lines of proof code worth of math theorems. This, as identified later, corresponds to about 10 modules. If the library doesn't offer that many modules, then we consider this to not be of interest to a mathematician.

For each ITP, we look for two different things. Firstly, whether the ITP has a mathematical library, and secondly, if the ITP has something resembling a package management system. If the ITP had either or both of these things, then they were decomposed into mathematical modules, as per the next section.

2.2.2.2 Step 2: Collecting Modules The next step is to determine what constituted a module for each of the libraries. We define a **module** as a single unit of work contributed to an ITP. This is used to reason about contributions and the scope of the ITP. We define this concept as in our analysis we compare components of standard libraries with packages in package management systems, so we refer to a contribution under the same name.

A collection of resources can be of three types: "package," "small" or "large." These are chosen to give a fair comparison between modules.

If the module came from a community contribution, such as Isabelle's Archive of Formal Proofs or Coq's package management system, then it is considered a "package" type. Meaning a module is considered to be whatever a unit of contribution is. For instance, a package is considered a module in Coq's package management system, or a submission in the Archive of Formal Proofs. We consider this to be a natural mapping because there are very little other options in defining

a module in a package management system. It would both be unusual to split up packages into their smaller components due to having non standardised structure and also unusual to define a module as a collection of packages. It also means that the module is understandable, linkable and explorable in the review.

If the modules came from a "standard library," we attempted to divide the standard library into "package sized" sections, in order to make a fair comparison between contributions. Because a package, as above, is a natural mapping to modules, and standard libraries are often much larger than a single package, we need to split the library up in a way that makes each component to a package. To do this, we exploited that most libraries are structured in a hierarchy, so we split the library as determined by how far down the hierarchy we need to go until we get "package sized" sections. If only one level of the hierarchy is needed, it is classified as "small," if two levels are needed, it is classified as "large."

To determine whether a library is small or large, we first have a target module size, which we define to be 1500 lines of proof code. Then, the mean module size is calculated, and it is determined whether a small or large library is closer to 1500 lines on a log scale. Whatever is closer we assign that size to the library.

These modules are then collected from the internet automatically from the webpage using a python script. The script is designed so that it can update the review to newest modules automatically, ensuring that the users are always viewing the most up to date version of these libraries.

This script will be run monthly to ensure that the review is up to date.

From the modules, a number of elements were collected. The name of the module, a description if it was available, authors if it was available, and a URL to the module. These will all be presented to the user when discovering different modules supported by different ITPs.

2.2.2.3 Step 3: Classifying modules Our next step is to classify or exclude each module that we have collected.

There are a number of exclusion criteria for this classification. These exclusion criteria exist for modules that would be difficult or impossible to classify as a part of a mathematical topic. Those possible exclusion criteria include:

EC1: Utility, The module only offers a utility that is only relevant to this ITP. For instance, many ITPs double as programming languages, so any modules that allow for programming that works with the system such as file operations are excluded. This also includes modules that are integrations with other tools, or simply initialization libraries that only reference other modules.

EC2: No Documentation, If the module doesn't have any documentation or description of purpose, it is excluded from the classification. It should be noted that the definition of documentation is very minimal in this case. The module doesn't need to have much documentation, one sentence of its purpose is enough. However, if it is missing, the module is difficult to classify and is excluded.

EC3: Only Documentation: If the module is simply documentation and doesn't provide any new resources, it is also excluded.

EC4: Deprecated: If the module has been marked or considered deprecated, then it is also excluded.

Excluded modules will be marked as excluded, and the reason for exclusion placed in the living review.

If a module doesn't fall under any exclusion criteria, then the modules is ready to be classified. Each module is then assigned a mathematical subject that it is relevant to.

The topics these modules were classified under come from the Mathematical Subject Classification 2020 (MSC2020) [9]. MSC2020 is a classification of mathematical subjects that is often used for mathematics papers. Each topic is given a code representing its classification, One such example of a code is 68P05, the code for work about data structures. The code is split up into three sections, the top level is represented by a top level mathematical field, For instance, the 68 in 68P05 means

that it's in the field of Computer Science. To refer to the general field of Computer Science, it is referred to is 68-XX, with the -XX meaning that it could be anything under this field. The next level of classification is represented by a letter. In this case, the P in 68P05 refers to subclassification of "Theory of Data" under "Computer Science." If one wishes to refer to the mid level classification, then one can do so like 68Pxx, by replacing the final two numbers with "x"s. Finally, the last two numbers represents the final level of classification. In this case, the 05 refers to "Data Structures" under "Theory of Data" under "Computer Science."

Some further examples of MSC Classifications are:

68Nxx: Theory of Software

68N15: Theory of Programming Languages

13-XX: Commutative Algebra

13Axx: General commutative ring theory

13A50: Actions of groups on commutative rings; invariant theory

This classification is chosen in order to be familiar to mathematicians, and ensure that modules can be classified with enough specificity.

If the module also comes with some form of category (for instance, Coq's package management system allows adding categories to a package, indicating the subject it covers, or large library modules use the top level module as a category), then the classification of the package is "guessed." This is done by mapping the category to a classification. The module can be browsed and seen in the tree, but however is marked as "unverified." If the module doesn't come with a category, then the module is marked as "unclassified."

Then, a module's description (for instance, abstracts, package descriptions, names) is inspected manually and a classification is given to it. The package is then considered to be "verified."

Finally, there is a limitation in our study, in that a very large number of mathematical topics have been covered by ITPs. As of such, there are some subjects that would require a very wide and professional understanding of the many facets and areas of mathematics in order to classify correctly. Any package that is therefore outside of the ability of the author to classify, is labeled as "needs professional" during verification. However, these modules are at the very least categorized vaguely into a top level MSC classification. For example, the author doesn't have a strong understanding of partial differential equations, so any package that is from the field of partial differential equations is marked as so, but not classified with any specificity beyond that.

This means that modules can be placed into five different states:

Unclassified: The module is included in the system, but the system does not have any idea of what math classification it falls under, nor has it been verified.

Unverified: The module's classification has been guessed automatically based on a category provided by the resource providing the module. For instance, Isabelle's Archive of Formal Proofs categorizes modules by topic in their index, and the classification of the module is guessed based on this.

Excluded: The module has been inspected and meets one of the exclusion criteria. This excluded module is not classified and removed from the study.

Needs Professional: The module has been verified but classified to a top level. It requires a professional mathematician to classify more specifically.

Verified: The module has been verified manually to be in the correct classification. This is the final desired state of a module.

States that are **Unclassified** or **Unverified** represents states that are incomplete. As in, at some later point, manual intervention is required to "complete" the classification. The reason why we discuss and include unclassified and unverified modules in the analysis is that we would prefer to

present all the resources available with a prover to a viewer, even the newest resources, before a human goes into the system to verify a package. The goal however, is to have no or little packages in the unclassified and unverified state.

A static snapshot of the work presented in Section 4 contain no Unclassified on Unverified newer, but the living version of it may, as it continues to collect newer module.

2.2.3 Counterexample Generator Methodology

When proving a theorem, a counterexample generator attempts to find an example for which the theorem does not hold. This helps the user better understand when the theorem and the statement that they wish to prove.

A literature review was performed in order to identify counter example generators, and the ITPs systems that they have support.

This is a very minor contribution, but simply attempts to give a very brief overview of support for counterexample generators for theorem proving.

2.2.4 Math Notation Methodology

Finally, as another minor contribution, we include details about which theorem provers use mathematical notation in proving theorems.

Determining whether an ITP uses mathematical notation in proofs can easily be done by examining example ITP code, and determining whether they use math notation. Which ITPs support mathematical notation was further included as a minor contribution to the living review.

2.2.5 General features of about ITPs

To compare between different ITPs, we must first start by collecting general features about them. This is tied in RQ3, as working out whether an ITP should be used on a project will require having some general information about it.

We decided to use a 2019 Systematic Literature Reviews on Theorem Provers as our starting dataset [63]. This review went through 27 theorem provers and described the features that each prover had. This was converted into a dataset and used to compare general features.

This dataset contained several properties about ITPs, and properties to compare between them.

The full set of properties we used from the dataset are:

- What the ITP is based on (Syllogism, Type Theory, Etc)
- The logic of the ITP
- The Truth value of the ITP
- Whether it supports Set theory
- Whether it has a library
- What the ITPs calculus is
- Whether the architecture is modular or monolithic
- The programming language it's implemented in
- Whether it has a GUI or CLI interfaces
- The Platforms its supported on (Windows, Mac, Linux)
- Whether it has an IDE
- When it was first released.
- Its latest release

A lot of these features (such as the logic, truth value etc) require explanations as to what they refer to. These explanations will be included within the tool. This allows people unfamiliar to the field to learn what the components of ITPs are and why they are important.

Some newer ITPs were not included in the dataset, such as Lean, F* and Idris. These were added manually to the dataset.

The systematic literature review we source our data from however, is already out of date for the latest release of its provers. As of such, we have a python script that automatically retrieves whether any newer versions of a prover have been released by checking GitHub Tags. It gets the latest tag to be published and adds that as the latest release on the living review, ensuring that the review doesn't go out of date by having newer releases.

3 Literature Review

The amount of papers found in each section of the review are shown in Table 1. This totals to 36 papers found on the topic.

There was a surprisingly small amount of papers caught by query in comparison to snowballing. This is because many papers that discuss usability issues about ITPs do not tackle the problem directly (28/38 of the papers), but rather showcase a feature that has been coded into an ITP interface. These features do indeed solve a usability problem implicitly, and represent the bulk of the work on improving interfaces for ITPs. However, comparatively little research has been done identifying the issues with ITP interfaces and empirically comparing these user interface modifications for merit (10/38 of the papers).

Table 1: Literature review papers

Round	Found	Relevant
Query	45	14
Snowball 1	121	14
Snowball 2	191	5
Snowball 3	99	2
Snowball 4	44	1
Snowball 5	4	0
Total	504	36

When going through papers, it was interesting to find a large amount of papers proposing user interface models, but not actually identifying the problems that they solve, nor evaluating their effectiveness. In fact, out of the 28 papers that showcased user interface improvements, only 2 papers evaluated the their interface improvement to without the improvement [18, 44]. That there is not enough empirical studies verifying usability issues has been cited as an issue that needs to addressed in the past [39].

The following sections outline usability issues and solutions to those issues. Tables are included outlining the usability issues mentioned. If the same problem is mentioned in two papers, it is given two rows.

The theorem prover column refers to the theorem prover the usability issue was found in. If the problem is a general comment “General" is written.”Textual" means a theorem prover that uses proof script to solve theorems, such as Isabelle/HOL, Coq, Agda. “Direct Manipulation" means a theorem prover that uses direct manipulation to solve theorems, such as KeY.

The discovered column indicates the evidence that that problem exists. "Suggested" simply means that problem or solution was simply inferred or has not actually been evaluated as effective. Other values indicate the type of study that the paper used to observe or evaluate this problem or solution.

3.1 Theorem Provers

First of all, a brief overview of the theorem provers is in order.

3.1.0.1 KeY KeY is a Direct Manipulation theorem prover, meaning that unlike Textual theorem provers, does not prove theorems by writing proof scripts, but instead works by modifying a proof object directly until all proof obligations have been solved. KeY works by annotating Java programs with preconditions and postconditions. These conditions are then fed into KeY as proof obligations. KeY can act as a fully automatic prover, but also allows the user to attempt to find a proof if the prover fails. KeY has also formed the basis of KeYmaera and KeYmaera X, which are for proving properties of hybrid systems.

3.1.0.2 HOL HOL is actually a family of theorem provers. Notably HOL4, ProofPower, HOL Light and HOL Zero. HOL is one of the oldest provers in this list, and HOL Light is known to be used as a lightweight prover, with a very easily checkable kernel. HOL provers are textual, and have a simple type system and use tactics to prove propositions.

3.1.0.3 Isabelle Isabelle (also known as Isabelle/HOL, but for this paper will remain as Isabelle to prevent confusion) is one of the most popular theorem provers. The prover has been used for the verification of the SeL4 prover, and exists as the state-of-the-art of ITPs. Isabelle like HOL has a simple type system and is based of the Logic for Computable Functions.

3.1.0.4 Coq Coq is another popular ITP that also supports a dependent type system. It's based on the Calculus of (Co)Inductive Constructions, which was designed specifically for Coq. Coq has been used to prove the four colour theorem, and create the CompCert certified C compiler.

3.1.0.5 Matita Matita is a theorem prover based on Coq's Calculus of (Co)Inductive Constructions, and was designed to address many of the pain points in working with Coq. Matita is a much simpler prover that aims to present the theorem prover as editing a mathematical library. As of such, Matita's solutions to problems are often pain points in Coq (mathematical notation, Tinycals etc).

There are a few other provers also in this review, such as iCon, CardiZ and Dafny. These ITPs are often either coded as proofs of concepts (such as iCon), or are no longer maintained (in the case of CardiZ or Dafny). The problems raised by them however, are often relevant for current day ITPs.

This is by no means a complete list of modern ITPs. Such compilations have been done [63]. These are only the ITPs discussed in the papers found in the review. There are notable ITPs that are missing from this list that caught us by surprise, including Agda, Lean and Mizar.

3.2 Abstraction Gradient

The problems classified under abstraction gradient are shown in Table 2.

Table 2: Abstraction Gradient Problems

Theorem prover	Problems	Discovered	citation
KeY	Interaction with low level logic	Focus Groups	[15]
Isabelle	Missing Library	Focus Groups	[15]

The abstraction gradient Dimension concerns itself with the highest and lowest levels of abstraction that are presented. Are they at the appropriate level of abstraction?

Issues in this dimension were uncommon. KeY was found to require interacting on low level logic formulas consistently. Similar issues with tedious interactions with KeY are mentioned in the viscosity's section. No solutions were found or suggested to this problem, and it has not been

empirically tested.

Focus Groups [15] found that Isabelle's library lacks the appropriate mathematical foundations. Interestingly, this is the only issue of this class and is not mentioned elsewhere, often library issues are more about managing and searching large libraries, which Matita attempts to handle, and correct documentation of libraries. This has not been tested empirically. Other than the implicit solution of providing better library support for theorem provers, no solution has been provided for this problem.

3.3 Closeness of Mapping

The problems classified under Closeness of Mapping are shown in Table 3.

Table 3: Closeness of Mapping Problems

Theorem prover	Problems	Discovered	citation
KeY	Unintuitive mapping between formula and program	Focus Groups	[15]
CardiZ	Cannot sketch out proofs	Questionnaire	[48]
Coq	Cannot use mathematical notation	Survey	[18]
Coq	Cannot use mathematical notation	Suggested	[7, 87]

The dimension of Closeness of Mapping is whether the interface maps well to the problem world. For interactive theorem provers, it has to do with how well the proof state is understood in comparison to the actual problem.

Focus groups [15] found that because KeY attempts to prove properties through annotations and java source code, it can sometimes be difficult to see how this proof state maps to the program [15]. This issue is not mentioned in any other source and not tested empirically. No solutions have been suggested for this.

CardiZ, an ITP that can be used to prove properties of Z specifications, found that the user could not sketch out proofs before an attempt [48]. This is the only paper on CardiZ, as CardiZ is not a popular prover. No solutions have been suggested for this.

A common issue that came up with Coq was the inability to use mathematical notation [7, 18, 87]. Notation issues are problematic in ITPs. One one hand, theorem provers such as Isabelle and Agda allow using mathematical notation in their theorems. This helps the user understand the theorem in a terse syntax. On the other hand, mathematical notation can often be ambiguous and difficult to type. Isabelle allows using LaTeX style commands such as `rightarrow` to render math notation, whereas Agda allows Unicode in source files. In order to avoid ambiguity, Coq has no support for math notation, and in response to this, Matita has LaTeX style mathematical notation [7, 87]. This issue came up in three different sources.

3.4 Consistency

The problems classified under Consistency are shown in Table 4.

Table 4: Consistency Problems

Theorem prover	Problems	Discovered	citation
Isabelle	Difficult to know what tactics and lemmas to use	Focus Groups	[15, 16]
HOL	Difficult to know what tactic to apply next	Suggested	[3]
Isabelle	Hard to remember prover specific details	Suggested	[62]
KeY	Difficult to know what tactic to apply next	Suggested	[59]
Isabelle,HOL	Difficult to remember names of theorems	Observational	[4]
Coq	Difficult to find relevant lemmas	Survey	[18]
Coq	Difficult to find arguments for tactics	Observational	[71]

Theorem prover	Problems	Discovered	citation
Coq	Bad Library, inconsistent naming	Survey	[18]

Consistency is the Cognitive Dimension of whether, once learning part of the notation, the user is able to infer the rest of the notation.

In textual theorem provers, it is often difficult to remember the name of the next tactic, theorems or lemmas should be applied in any situation. This has been bought up in focus groups [15], observational studies [4], surveys [18] and suggested as a problem from various other sources.

Solutions to this problem often include choosing applicable tactics by menu [3, 4] Which has been implemented in Coq through Proof Previews [18] and in KeYmaera X [59]. Machine learning for choosing appropriate recommendations has been suggested for this problem [71], and has also been implemented through the PaMpeR tool in Isabelle [62]. A second way of tackling this problem is to improve library searching, which was suggested [4] and is a focus in Matita [80]. Improving these tools is a promising area for improving the usability of ITPs

3.5 Diffuseness / Terseness

The problems classified under Diffuseness/Terseness are shown in Table 5.

Table 5: Diffuseness / Terseness Problems

Theorem prover	Problems	Discovered	citation
Isabelle	Bloated Formulas	Focus Groups	[15]
Isabelle	Large proofs correspond to large effort	Observational	[22]

Diffuseness is the Cognitive Dimension of the tersity/verbosity of the syntax. Bloated formulas were mentioned in Isabelle in Focus Groups, and projects with more lines of code were strongly correlated with more effort. No solutions have been suggested to reducing the size of code bases or formulas.

3.6 Error Proneness

The problems classified under Error Proneness are shown in Table 6.

Table 6: Error Proneness Problems

Theorem prover	Problems	Discovered	citation
Isabelle,HOL	Easy to get errors in Object Level Constructions	Observational	[4]
Isabelle,HOL	Incorrect predictions made about tactics	Observational	[4]
Isabelle	Difficult to manage namespace	Suggested	[22]

Error proneness is the Cognitive Dimension of whether a system allows its users to make errors.

Observational studies have found that in Isabelle and HOL it is easy to make errors in syntax. Considering the frequency of syntax errors, this issue came up surprisingly little other sources. This could be because syntax errors are relatively easy to fix and also decrease with usage. It's been suggested that this problem could be solved my improving feedback in Object Level syntax input [4]. A more interesting solution has been implemented for Coq is a structure editor based off keyboard cards [18]. This uses rolling chords (like *vi*) keyboard interfaces to interact with the theorem prover. This means that it is only possible to enter syntactically valid statements. This current solution only works on a subset of Coq's syntax. It was found to be slightly quicker than

using dropdown menus.

Sometimes when applying a tactic, an unexpected result would occur, causing the user to back up and try to understand the current state. This issue could be solved by Proof Previews [18], which allow the user to see a proof state when selecting tactics from a menu without actually applying the tactic. That way the user can cheaply explore tactics to continue in the proof.

Finally, for large verification projects such as SeL4, there is an issue with managing the namespaces of large amounts of theorems and lemmas. No verification of this problem nor solution has been suggested.

3.7 Hard Mental Operations

The problems classified under Hard Mental Operations are shown in Table 7.

Table 7: Hard Mental Operations Problems

Theorem prover	Problems	Discovered	citation
Coq	Hard to understand proof scripts statically	Suggested	[87]
General	Difficult to understand tacticals	Suggested	[38, 55]
General	Proof scripts can become complicated	Suggested	[8]

The Dimension of Hard Mental Operations refer to the difficulty in understanding and using the interface on a syntax level. This type of issue is common with ITPs, as the actual domain is complicated, so this is reflected with difficult syntax.

Proofs are hard enough to understand while viewing the dynamic nature of the proof, investigating proof state bit by bit. They are often near impossible to understand statically [87]. This issue has not been investigated empirically, but solutions often involve changing the syntax around proofs. One notable example of this is Isar for Isabelle [84], which attempts to mirror how a pen and paper proof is structured.

It is often difficult to understand tacticals, and the problem is made even worse when it is not possible to view the state of a tactical mid way through interaction. This problem has been suggested in several sources [38, 55, 87] but never empirically investigated. Solutions to this include representing tacticals as graphs [38, 55]. This solution has not been tested with users.

Proof scripts can also get very complicated for larger propositions. Keeping track of this complexity has been suggested with proof metrics, which are similar to classic code complexity metrics [8].

3.8 Hidden Dependencies

The problems classified under Hidden Dependencies are shown in Table 8.

Table 8: Hidden Dependencies Problems

Theorem prover	Problems	Discovered	citation
Isabelle	Hard to see dependencies between proofs	Suggested	[79]
KeY	Difficult to patch proofs that have slightly changed	Suggested	[13]
Isabelle	Hidden automation dependencies	Suggested	[22]
Isabelle	Difficult to patch proofs when dependencies change	Suggested	[22]
Isabelle	Hard to see dependencies between proofs	Suggested	[8]

Hidden Dependencies represent dependencies between components that are not shown explicitly. Hidden dependencies are everywhere in theorem provers. Like functions in many programming languages, lemmas can reference the lemmas that they use, but it is difficult to find where a

particular lemma has been used. Automation makes this problem even worse, where in Isabelle, an automatic tactic will try lemmas that are above it in the theory. This makes moving lemmas around a theory document difficult. Moving a lemma around a document, even if all the other lemmas it is references are above it, may cause it to fail due to it using a lemma by automation. Monitoring dependencies has been suggested as part of formal proof metrics. It's been suggested and implemented within CoqPIE to show these dependencies within the IDE [72]. Tools have also been built to analyse dependencies between Isabelle proofs [79]. For automated tactics, Isabelle's sledgehammer offers a unique solution to showing dependencies. The automatic tactic, after execution, simply prints a set of manual tactics that were used to prove the theorem into the document. That way, all the lemmas that were used in the automated tactic are made explicit [22]. None of these solutions have been empirically tested for validity.

Furthermore, often changing a definition or proof slightly requires changing the proof in order to match the new definitions. This is a tedious process.

None of these issues have been tested empirically.

3.9 Perceptual Cues

The problems classified under Perceptual Cues are shown in Table 9.

Table 9: Perceptual Cues Problems

Theorem prover	Problems	Discovered	citation
HOL	Difficult to understand proof state	Suggested	[48]
KeY	Difficult to understand proof state	Suggested	[43]
General	Difficult to understand proof state	Suggested	[30]

Perceptual Cues is how easy it is to understand what is being represented. Understanding proof state is an enormous part of theorem proving. The normal solutions to understanding proof state are to offer more ways of viewing it, and ensuring easy access to these views. As of such, solutions are found in the visibility section.

3.10 Premature Commitment

The problems classified under Premature Commitment are shown in Table 10.

Table 10: Premature Commitment Problems

Theorem prover	Problems	Discovered	citation
HOL	Need to redesign model if proof attempt fails	Suggested	[15]
Coq	Have to apply tactics before understanding what they do	Suggested	[18]

When an attempt to prove a theorem fails, either one of two things has happened. First, the proof the user are attempting to perform is incorrect, or the model itself is in error. The model is often in error, and as of such there is a Premature Commitment to a model before having a full understanding. Counterexample generators such as Quick Check and nitpick [15, 16] for Isabelle help prevent the user from trying to prove improvable lemmas by providing the user with a counterexamples to show why their lemmas can't be true.

Furthermore, tactics often need to be applied to discover what they do. Typing out this tactic is part of the exploration, and represents another premature commitment. A cheaper way to explore tactic applications were trialed with Proof previews in Coq [18]. These proof previews allowed the selection of the next tactic by menu, and hovering over a tactic previewed it's application in a separate window. This was found to be helpful with users.

3.11 Progressive Evaluation

The problems classified under Progressive Evaluation are shown in Table 11.

Table 11: Progressive Evaluation Problems

Theorem prover	Problems	Discovered	citation
General	Hard to understand why proof state fails	Suggested	[45]
KeY	Hard to understand why proof state fails	Suggested	[14–16]
Dafny	Hard to understand why proof state fails	Suggested	[35]
KeY	Hard to understand why proof state fails	Suggested	[55]
General	Hard to understand automated tactics	Suggested	[59]
e General	Not enough feedback hinders learning	Suggested	[59]
Coq	Lack of background automation	Survey	[18]
General	Lack of background automation	Suggested	[46]
KeY	Bad feedback hinders learning	Survey	[13]
Coq	Don't know whether an automated tactic would prove a goal	Survey	[18]
Isabelle	Non reactive interfaces	Suggested	[15]
Isabelle	Hard to understand errors from bad inferences of types	Suggested	[15]
Isabelle	Performance of automatic strategy	Suggested	[15]
Isabelle,KeY	Difficult to understand automated strategy	Suggested	[15]

Progressive Evaluation is the Dimension of getting appropriate feedback from the system.

Not understanding why proof attempts fails in a widely cited example of this. This becomes especially true when automation is added to the mix. Insight to the operation of automated tactics is missing in many ITPs. This has been suggested to be one of the largest issues with the usability of ITPs in Focus Groups [15]. Although there is little empirical evidence of this issue, the fact that it is so widely cited indicates importance. Solutions to this issue often resolve around providing better visibility, and are covered there.

Other issues include that systems with low feedback make it difficult to teach using ITPs, and that ITPs do not effectively use background processing to provide the user with feedback. One way of improving feedback is using a cache of proof state [18, 22]. Another more novel way is to provide an agent based interaction model [46], where the user interfaces have a "Personal assistant", who then negotiates with proof agents to help solve a particular proof. This makes best use of background processing while the user is trying to solve a problem. Neither of these have been tested with users.

Finally, it was mentioned in a survey of Coq users that it would be nice to know in advance whether an automated tactic could prove a goal. This would prevent further unnecessary work.

3.12 Secondary Notation

Secondary Notation is the realm of comments, documentation, and even use of visual placement to convey meaning. Problems due to lack of secondary notation are usually simply because of missing features, and are therefore more naturally discussed as solutions.

The problems classified under Secondary Notation are shown in Table 12.

Table 12: Secondary Notation Solutions

Theorem prover	Intervention	Discovered	citation
HOL	Allow notes in tree contexts	Suggested	[3]
Isabelle,HOL	Allow adding notes to proof context	Suggested	[13]
Isabelle	Add document orientated features	Suggested	[83]
Isabelle	Gravity for automated lemma placement	Suggested	[22]
Isabelle	Deprecation tags	Suggested	[22]

Theorem prover	Intervention	Discovered	citation
Coq	Doc comments	Survey	[18]
HOL/CardiZ	Colour and low secondary notation	Survey	[48]
Coq	Lack of good tutorials and documentation	Survey	[18]
KeY	Poor documentation	Suggested	[13]
Isabelle	Better libraries and documentation	Suggested	[18]

One improvement on secondary notations is the ability to note and label parts of proof context. Usually, proof context is boxed off and cannot be documented other than basic comments. Interestingly, although this feature is cited multiple times. It remains, to the best of my knowledge, unimplemented, and as with the rest of the solutions in this category, untested.

Document oriented proof organization tries to make each theory readable both to a human and to a computer, and involves allowing linking to external theories, websites, diagrams and other features all in the prover editor. This is commonly done with web interfaces controlled by ITP code. This method has not been investigated as being beneficial, but Matita itself was built to support this style of interaction.

Features such as deprecation tags, doc comments and automatic naming of lemmas frequently showed up. These indicate that it is important for the user to break out and write hints to help themselves and others navigate their code. These have been implemented in some provers, but again, have not been tested.

Finally, a very common issue with ITPs is the lack of tutorials and documentation, particularly around library functionality. This is remarkably important, regardless of what the prover is.

3.13 Viscosity

The problems classified under Viscosity are shown in Table 13.

Table 13: Viscosity Problems

Theorem prover	Problems	Discovered	citation
Isabelle	Messy Downwards compatibility	Focus Groups	[15]
Isabelle	No support for proof refactoring	Focus Groups	[15]
Direct Manipulation	Tedious Interactions	Suggested	[15, 36]
General	Hard to make effective use of large library	Suggested	[6, 80]
Isabelle	Tacticals difficult to write	Suggested	[12]
Coq	Have to update proofs once definition changes	Suggested	[71]
Coq	Proof renaming and refactoring is tedious	Suggested	[71]
Coq	Large proof scripts require too long to recompile	Suggested	[10]
Isabelle	Large proof scripts require too long to recompile	Suggested	[82]
Coq	Difficult to select terms	Survey	[18]
Coq	Unnecessary re-running of proofs	Survey	[18]
Coq	Slow for large proofs	Suggested	[72]
Coq	Change in lemma requires change in proof	Suggested	[72]
KeY	Bad change management	Suggested	[13]
KeY	Bad Automated proof performance	Suggested	[13]
KeY	Hard to decompose proof	Suggested	[14]

Viscosity is the Cognitive Dimension of the ease of changing the state of the programs.

One source of viscosity is simply performance. As automatic strategies get more complicated, their performance becomes important for them to be useful to the user. This has been suggested by focus groups and in surveys. This becomes a particularly difficult problem especially for larger systems.

Attempts to improve performance have been done by asynchronously loading only required parts of the proof in Coq [10] and Isabelle [82]. Improvements to performance of automatic strategies will always be an improvement [22], including making better use of the library [6, 80].

A second source is the need to make trivial interactions when making small changes. For instance, the renaming of a lemma require needing to go through several files to find where to change the identifier. Messy downwards compatibility, changing definitions, and lack of refactoring are all examples of this. These are usually addressed by refactoring tools that have been suggested as necessary [22, 71] and implemented in some IDEs such as CoqPIE [72]. These solutions have not been tested with users.

The third is simply interactions that are tedious and error prone. This is more common in direct manipulation theorem provers such as KeY. No solutions have been suggested for this problem.

Finally, the fourth source of viscosity is clunky syntax, such as the need to explain selections to the theorem prover. Selections are a common issue when describing the part of the goal that the user wants to rewrite. This part might be complicated, but has to be represented textually. This has served as a challenge for ITP designers. Selections using patterns has been implemented in Matita [7, 87] to address this pain point.

3.14 Visibility

The problems classified under Visibility are shown in Table 14.

Table 14: Visibility Problems

Theorem prover	Problems	Discovered	citation
KeY	Proof tree too detailed	Focus Groups	[15]
Textual	Limited insight to automated tactics	Suggested	[36]
HOL	Hard to understand proof tree	Suggested	[3]
HOL	Allow showing and hiding of proof contexts	Suggested	[3]
Coq	Cannot see intermediate proof states	Suggested	[87]
Coq	Difficult to see structure of proof tree	Suggested	[18]
Coq	Cannot see the relation between subgoals	Survey	[18]
Coq	Cannot quickly see type or simplification of term	Survey	[18]
KeY	Bad presentation of incomplete proofs	Suggested	[13]

Visibility was a commonly cited issue with interactive theorem provers.

The Cognitive Dimension of visibility has to do with being able to how information can be identified and accessible to the user. For the case of interactive theorem provers, there were cases where theorem provers show too much or too little information.

Direct manipulation theorem provers such as KeY were found to show too much information in the proof tree, which overwhelms the user trying to work out why a proof attempt has failed. The simplest solution to this is to only show information needed [30] and allow the opening and closing of views [3]. However, some IDEs (such as CoqIDE) come without a proof tree. These have been considered helpful [3, 18] and have been implemented with Traf [52]

In contrast, it's also been claimed that there is a lack of visibility of the proof state, particularly intermediate proof states within textual theorem provers. The lack of visibility is often to do with intermediate proof states. An intermediate proof state is the state that a proof is in before the full completion of a tactic, and can be used to determine how a tactic got to a particular proof state. Understanding these intermediate proof states is important in understanding the process of automatic theorem provers and the current state. Viewing intermediate proof states is not possible with Isabelle/HOL or Coq. In fact, it is not even possible to investigate the inside of tacticals making it even more difficult to understand intuitively a proof. The tactical problem has been

resolved by only using a subset of tacticals with Matita's Tinycals [7, 87]. KeYmaera X also offers traceability with automatic tactics, allowing insight to the operations they performed [59].

In one of the only empirical tests of two different user interfaces, an interface akin to a symbolic debugger is compared against the standard interface of KeY [43–45]. The symbol debugger was found to be easier to use. The interfaces are very different, and it could be for a variety of reasons. One such reason is that the interface of a symbolic debugger maps better onto the context of source code, and offers visibility of that connection. Offering different ways of viewing and interacting with proof state has been suggested as a way forward in the usability of ITPs [30, 35].

Diagrammatic representations of proof is an alternative way of proving theorems, as demonstrated with iCon [76] and Proof Transitions in CoqEdit [18]. This has not been tested empirically against other ITPs

3.15 Analysis

Many problems were identified. A summary of the problem is tabulated in Figure 3.

Cognitive Dimension	Problem	Identifier	Isabelle	KeY	Coq	HOL
Abstraction Gradient	Quality of library content	AG1	Focus Group			
Abstraction Gradient	Bad abstraction	AG2		Focus Group		
Closeness of Mapping	Notation	CM1			Survey	
Closeness of Mapping	Bad mapping to program	CM2		Focus Group		
Consistency	Hard to find tactics and lemmas applicable	C1	Observational		Survey	Observational
Diffuseness	Bloated representations	D1	Focus Group			
Error proneness	Difficult namespace	EP1	Suggested			
Error proneness	Difficult syntax	EP2	Observational			Observational
Error proneness	Unexpected results from tactics	EP3	Observational	Suggested		Observational
Hard Mental Operation	Hard to understand proof scripts	HMO1	Suggested		Suggested	
Hidden Dependencies	Cannot see dependencies	HD1	Suggested		Suggested	
Hidden Dependencies	Difficult to patch proofs	HD2	Suggested	Suggested	Survey	
Hidden Dependencies	Difficult to move lemmas	HD3	Suggested			
Perceptual Cues	Difficult to understand proof state	PeC1	Suggested	Suggested	Suggested	Suggested
Premature Committment	Countexamples	PrC1		Focus Group		
Premature Committment	Constant Redesign	PrC2	Suggested	Suggested	Suggested	Suggested
Progressive Evaluation	Error messages	PE1	Suggested			
Progressive Evaluation	Difficult reason as for proof failure	PE2	Focus Group	Suggested		
Viscosity	Proof Refactoring	V1	Focus Group		Survey	
Viscosity	Performance	V2	Focus Group		Survey	
Visibility and Juxtaposability	Too much detail	VJ1		Focus Group		Suggested
Visibility and Juxtaposability	Missing required information	VJ2			Survey	
Visibility and Juxtaposability	Insight into automated tactics	VJ3	Suggested	Suggested		

Figure 3: Identified Usability Issues

This analysis finds that although many issues were identified, there is very little empirical research on these problems. This is probably due to the difficulty in recruiting expert participants to these studies, and the small size of the field.

An empirical analysis of all of these problems is well and truly outside the scope of this thesis. The task at hand is to now select problems that can be addressed.

The first thing to consider is that we are creating a living review. Many of the usability issues that arose have a strong human component. For instance, "Hard to predict the results of tactics" would be very difficult to evaluate without performing a usability test. If we were to include a measure within the living review that required the conducting of a usability test, the usability test would need to be run on a periodic basis to keep it up to date with the current state of technology. This is highly undesirable, as such a project would be extremely time consuming and expensive, and would require a time and money investment for years after this thesis is published.

To address this, we restrict this thesis to usability issues that can be determined to exist without the highly expensive intervention of a user. This leaves the following options:

- Scope of Library
- Math Notation support
- Counterexamples

- Performance

All these issues beside performance are within the scope of our living review. Although creating a living review for ITP performance that automatically updates is technically feasible, we determined that this was not a usability problem of interest in comparison to the large amount of effort and money (setting up a list of standard activities, setting up of standardised hardware, running programs in test harnesses) required to include performance the review's scope.

4 Results

In this section, we discuss in detail the living review that we have contributed. This section is only a static snapshot of the review at its current stage. The full living review is available online and will be fully up to date. Therefore, it should be noted that the living review itself contains all the information found in this section and more. If you wish to view the results as it is up to date, then we invite you to look through the online widget.

The widget can be explored from the following: https://samnolan.me/thesis/review.html

The source code for this thesis, the widget, and all the code behind it are also available on GitHub at https://github.com/Hazelfire/thesis. What follows is a snapshot of the findings in the review.

The following 17 ITPs were included in the review: ACL2 [51], Agda [64], Atelier B [25], Coq [19], Getfol [33], HOL Light [41], HOL4 [78], Isabelle [68], JAPE [21], LEO-II [17], Lean [61], Metamath [65], Mizar [34], PVS [73], RedPRL [5], Twelf [69], and Z/EVES [74].

The results are split into three sections. In Section 4.1, results about the state and scope of mathematical libraries of ITPs are discussed. In Section 4.2 results about the support of Counterexample generators are covered. Finally, in Section 4.3 and results about mathematical notation support are covered.

4.1 Mathematical Libraries

This section details results about the distribution of mathematical topics currently covered by ITPs, as of 28 October 2021. All findings, including the dataset of classified modules, can be viewed and downloaded in the up to date form by viewing the review online.

The methodology for this section has been laid out in Section 2.2.2.

4.1.1 Step 1: Identifying Libraries

This section corresponds to the section laid out in Section 2.2.2.1

13 mathematical libraries were covered in this analysis. They are detailed in Table 15.

Twelf was excluded due to not proving any mathematical theorems within its library, failing IC1.

Getfol and RedPRL failed to meet IC2, in having small libraries with only 1372 and 2680 lines of proof code respectively.

LEO-II and Z/EVES do not have standard libraries.

4.1.2 Step 2: Collecting modules

As per our methodology in section Section 2.2.2.2, we must now chose what we mean by a module for comparison. What we chose as a module is listed in Table 15 for each ITP.

We chose the library sizes based on the methodology in Section 2.2.2.2.

Table 15: Libraries covered in the living review

Name	Library	Type	Module Definition
ACL2	Community Books	packages	a community book, such as acl2ls or data-structures
Agda	Community Libraries	packages	a submission such as AoPA or DTGP
Agda	Standard Library	small	a top level module, such as Data or Algebra
Coq	Standard Library	small	a top level module, such as Init or Arith
Coq	Packages	packages	a package
HOL Light	HOL Light Library	large	a second level module, such as Library/analysis.ml or Multivariate/clifford.ml
HOL4	HOL4 Library	small	a top level module, such as bool or topology
Isabelle	Core Libraries	large	a session, such as HOL/HOL-Algebra
Isabelle	Archive of Formal Proofs	packages	a submission
Lean	Lean Mathematical Library	large	a second level module, such as algebra.algebra or probability.distribution
Metamath	Metamath Library	small	a "part", such as "ZF SET THEORY", or one of the smaller theories, such as "Higher Order Logic"
Mizar	Mizar Mathematical Library	package	a submission, such as abian of aff_2
PVS	NASA PVS Library of Formal Developments	packages	a module, such as algebra or analysis

4.1.3 Step 3: Classifying modules

From these libraries, 5581 math modules were collected libraries. All of these modules were classified according to MSC2020. They are presented in Figure 4.

In descending order of the amount of total mathematical modules, a description of the classification for each ITP is given:

Mizar - Mizar Mathematical Library: Has a total of 1383 modules. Of these modules, 52 modules were excluded. Including 16 modules were excluded for being a utility (EC1). 915 modules were verified into categories and 416 require a professional mathematician to properly categorise.

ACL2 - Community Books: Has a total of 873 modules. Of these modules, 616 modules were

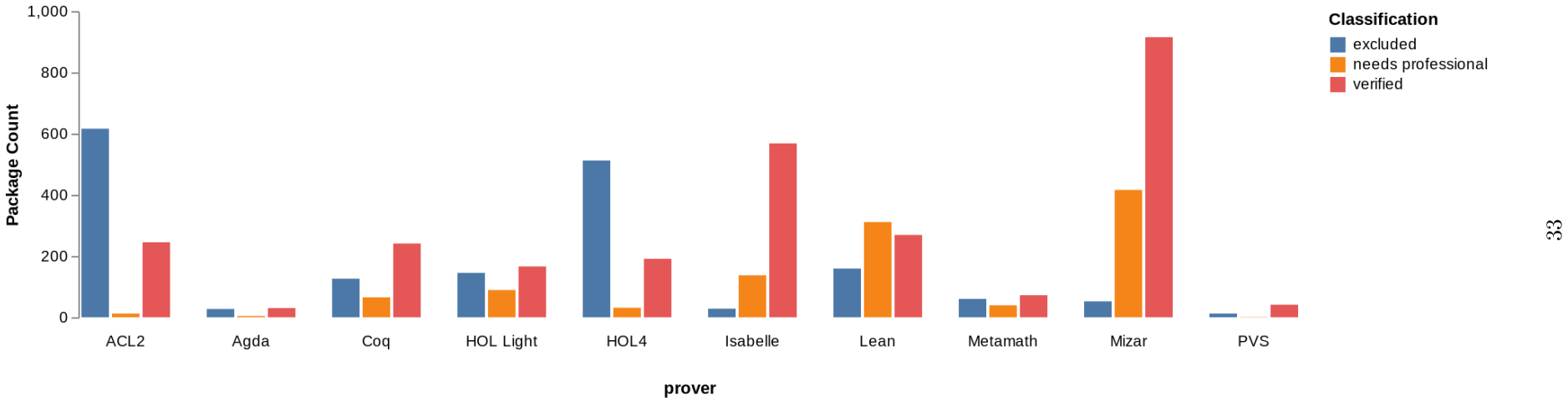

Figure 4: Amount of modules found in each ITP

excluded. Including 448 modules were excluded for being a utility (EC1), 53 modules were excluded for being only documentation (EC3), and 26 modules were excluded for being deprecated (EC4). 245 modules were verified into categories and 12 require a professional mathematician to properly categorise.

Lean - Lean Mathematical Library: Has a total of 739 modules. Of these modules, 159 modules were excluded. Including 150 modules were excluded for being a utility (EC1), one module was excluded for being only documentation (EC3), and 6 modules were excluded for being deprecated (EC4). 269 modules were verified into categories and 311 require a professional mathematician to properly categorise.

HOL4 - HOL4 Library: Has a total of 734 modules. Of these modules, 512 modules were excluded. Including 294 modules were excluded for being a utility (EC1) and one module was excluded for being only documentation (EC3). 191 modules were verified into categories and 31 require a professional mathematician to properly categorise.

Isabelle - Archive of Formal Proofs: Has a total of 611 modules. Of these modules, 3 modules were excluded. Including 3 modules were excluded for being a utility (EC1). 495 modules were verified into categories and 113 require a professional mathematician to properly categorise.

HOL Light - HOL Light Library: Has a total of 400 modules. Of these modules, 145 modules were excluded. Including 71 modules were excluded for being a utility (EC1) and 25 modules were excluded for being only documentation (EC3). 166 modules were verified into categories and 89 require a professional mathematician to properly categorise.

Coq - Packages: Has a total of 396 modules. Of these modules, 119 modules were excluded. Including 114 modules were excluded for being a utility (EC1), one module was excluded for being only documentation (EC3), and 3 modules were excluded for being deprecated (EC4). 214 modules were verified into categories and 63 require a professional mathematician to properly categorise.

Metamath - Metamath Library: Has a total of 171 modules. Of these modules, 60 modules were excluded. Including 42 modules were excluded for being a utility (EC1), 2 modules were excluded for being only documentation (EC3), and 16 modules were excluded for being deprecated (EC4). 72 modules were verified into categories and 39 require a professional mathematician to properly categorise.

Isabelle - Core Libraries: Has a total of 123 modules. Of these modules, 25 modules were excluded. Including one module was excluded for being a utility (EC1) and 18 modules were excluded for being only documentation (EC3). 73 modules were verified into categories and 24 require a professional mathematician to properly categorise.

PVS - NASA PVS Library of Formal Developments: Has a total of 54 modules. Of these modules, 12 modules were excluded. Including 11 modules were excluded for being a utility (EC1) and one module was excluded for being only documentation (EC3). 41 modules were verified into categories and 1 require a professional mathematician to properly categorise.

Agda - Community Libraries: Has a total of 43 modules. Of these modules, 19 modules were excluded. Including 13 modules were excluded for being a utility (EC1) and 2 modules were excluded for being only documentation (EC3). 21 modules were verified into categories and 3 require a professional mathematician to properly categorise.

Coq - Standard Library: Has a total of 36 modules. Of these modules, 7 modules were excluded. Including 7 modules were excluded for being a utility (EC1). 27 modules were verified into categories and 2 require a professional mathematician to properly categorise.

Agda - Standard Library: Has a total of 18 modules. Of these modules, 8 modules were excluded. Including 8 modules were excluded for being a utility (EC1). 9 modules were verified into categories and 1 require a professional mathematician to properly categorise.

It was found that some libraries were clear outliers in mathematical scope covered. Those libraries were Coq, HOL Light, Isabelle, Lean, and Mizar. It would be difficult to justify use of other theorem provers as a mathematician getting into the field.

HOL4, HOL Light and ACL2 have a lot of excluded modules due to having many utilities available for the user. For instance, in HOL Light allows you to write proofs in a declarative Mizar style, and has a module for this. This is not relevant for determining mathematical scope.

The chart in Figure 5 shows which verified top level MSC Classifications these modules were sorted into as of 28 October 2021. Each column represents a top level classification of a mathematical topic from MSC2020, sorted by by the amount of total modules in each classification. Each colour represents modules from a different ITP.

This chart was created using Vega-Lite [75] a chart visualisation framework. It is reproduced in the living review where it is fully up to date and is further interactive, showing the exact modules counts when hovering with a mouse.

The chart represents the mathematical scope of each of the libraries. If a particular field has more modules, it represents a larger availability of prior work to build upon when developing new proofs, therefore less working from the ground up.

Computer science (68-XX): Had a total of 978 modules. The ITPs with the most packages in this category were *Isabelle* with 323 modules, *HOL4* with 158 modules, and *Coq* with 146 modules. The most popular second level MSC classifications were *Theory of data* (68Pxx) with 320 modules, *Theory of computing* (68Qxx) with 252 modules, and *Theory of software* (68Nxx) with 151 modules. Computer Science was clearly the most popular category, mainly because ITPs like ACL2, Isabelle, HOL4 and Coq are all mainly built for the purpose of verifying software. There was a large amount of data structures of all kinds created to reason about programs. Even Mizar, a usually mathematical ITP, has several modules dedicated to the verification of software.

Mathematical logic and foundations (03-XX): Had a total of 548 modules. The ITPs with the most packages in this category were *Mizar* with 188 modules, *Isabelle* with 113 modules, and *Coq* with 53 modules. The most popular second level MSC classifications were *Set theory* (03Exx) with 183 modules, *General logic* (03Bxx) with 180 modules, and *Proof theory and constructive mathematics* (03Fxx) with 56 modules. Mathematical Logic and Foundations was common because often ITPs would start developing their libraries through laying the foundations. However, some ITPs such as Mizar have large amounts of contributions on topics such as fuzzy logic, which does not neccesarily make up its foundation but is still within this category.

Number theory (11-XX): Had a total of 209 modules. The ITPs with the most packages in this category were *Mizar* with 63 modules, *HOL Light* with 46 modules, and *Isabelle* with 38 modules. The most popular second level MSC classifications were *Elementary number theory* (11Axx) with 72 modules and *Sequences and sets* (11Bxx) with 19 modules. Number Theory consistently had a large amount of modules from most ITPs. Elementary Number theory made up the majority of this category, mainly modular arithmetic, distribution of primes and primality checking.

Real functions (26-XX): Had a total of 186 modules. The ITPs with the most packages in this category were *Mizar* with 89 modules, *HOL Light* with 36 modules, and *Metamath* with 21 modules. The most popular second level MSC classifications were *Functions of one variable* (26Axx) with 90 modules and *Functions of several variables* (26Bxx) with 34 modules. Real Functions covers topics often considered to be part of real analysis. This classification has a strong presence in Mizar, where a large amount of real analysis is covered. But also HOL Light, which sports a strong multivariate library.

General topology (54-XX): Had a total of 185 modules. The ITPs with the most packages in this category were *Mizar* with 121 modules, *Lean* with 46 modules, and *HOL Light* with 6 modules. The most popular second level MSC classifications were *Topological spaces with richer structures* (54Exx) with 24 modules and *Fairly general properties of topological spaces* (54Dxx) with 20 modules. Topology made up a large amount of modules. Mizar definitely dominated this space, and discusses topology widely.

Order, lattices, ordered algebraic structures (06-XX): Had a total of 179 modules. The ITPs with the most packages in this category were *Mizar* with 101 modules, *Lean* with 50 modules, and *Isabelle* with 12 modules. The most popular second level MSC classifications were *Lattices*

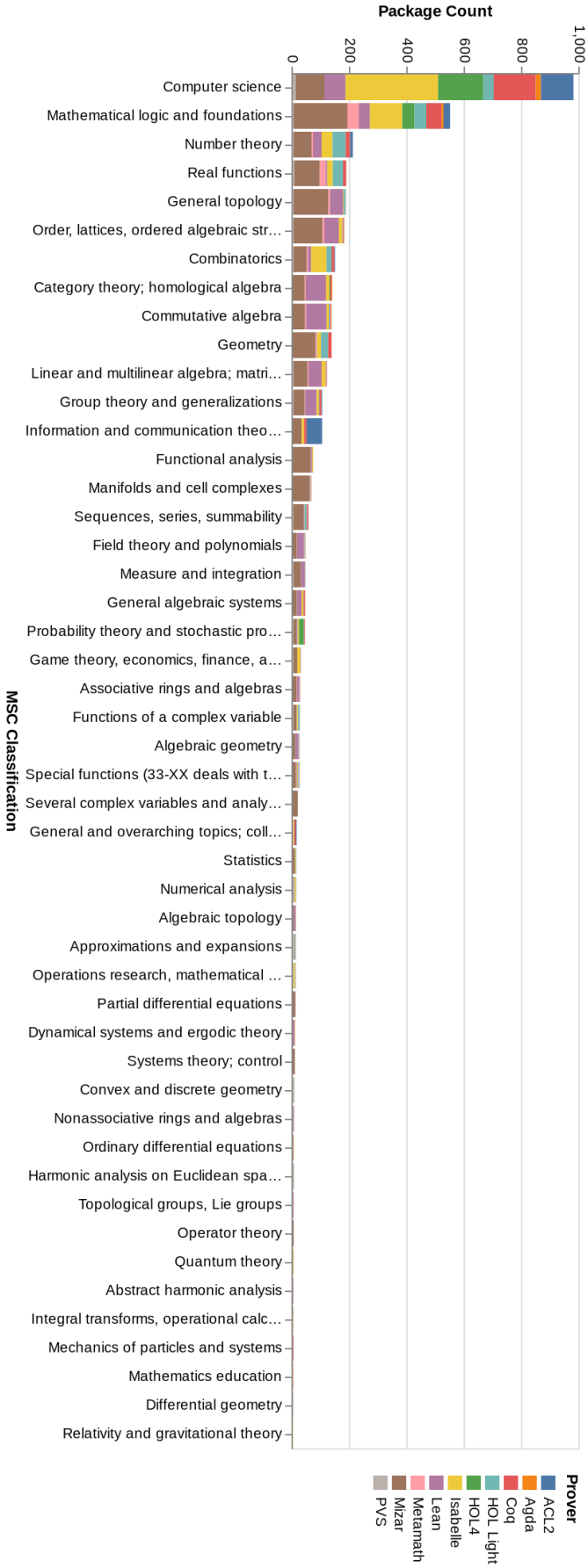

Figure 5: Math Package classifications, as of 28 October 2021

(06Bxx) with 60 modules and *Ordered structures* (06Fxx) with 19 modules. Orders was covered widely, mainly in discussion with lattices. Mizar has a large amount of modules dedicated to formalising continuous lattices, which are then used in the context of Domain Theory.

Combinatorics (05-XX): Had a total of 147 modules. The ITPs with the most packages in this category were *Isabelle* with 54 modules, *Mizar* with 46 modules, and *HOL Light* with 18 modules. The most popular second level MSC classifications were *Graph theory* (05Cxx) with 93 modules and *Enumerative combinatorics* (05Axx) with 31 modules. There were a large amount of combinatorics modules, mainly from graph theory. Isabelle here has the most packages, using graph theory mainly for the purpose of verifying graph algorithms.

Category theory; homological algebra (18-XX): Had a total of 137 modules. The ITPs with the most packages in this category were *Lean* with 70 modules, *Mizar* with 41 modules, and *Isabelle* with 12 modules. The most popular second level MSC classifications were *General theory of categories and functors* (18Axx) with 38 modules and *Categories and theories* (18Cxx) with 9 modules. Category theory is often in the context of functional programming and controlling effects. Haskell has popularized the use of monads for controlling effects. ITPs such as Lean reimplement those concepts in their ITPs.

Commutative algebra (13-XX): Had a total of 135 modules. The ITPs with the most packages in this category were *Lean* with 71 modules, *Mizar* with 42 modules, and *Isabelle* with 8 modules. The most popular second level MSC classifications were *Theory of modules and ideals in commutative rings* (13Cxx) with 22 modules and *General commutative ring theory* (13Axx) with 18 modules. Discussion of Commutative Algebra was mainly restricted to ITPs interested in proving math theorems, such as Mizar or Lean. Lean has a top level module entirely on ring theory.

Geometry (51-XX): Had a total of 135 modules. The ITPs with the most packages in this category were *Mizar* with 79 modules, *HOL Light* with 25 modules, and *Isabelle* with 14 modules. The most popular second level MSC classifications were *Real and complex geometry* (51Mxx) with 42 modules and *Analytic and descriptive geometry* (51Nxx) with 24 modules. Geometry was common among most theroem provers, with Mizar having a large amount of module about Affine Geometry.

4.2 Counterexample generators

Counterexample generators have been suggested to be beneficial in helping users understand the proof state they they were in [15].

This living review covers 3 counter example generators. These counterexamples generators were;

Nitpick [20]: Nitpick is a counterexample generator for Isabelle/HOL. It is fundementally a model finder and works by using the relational model finder KodKod [81] as a backend. Model finders are akin to SAT solvers, in that they work by attempting to find a collection of values that satisfy a given statement. Nitpick uses model finders in order to find possible counterexamples. Nitpick also outperforms QuickCheck. It has support for Isabelle.

Nunchaku [26, 67]: Nunchaku is a counterexample generator intended to be the successor of Nitpick. One of its main advantages over Nitpick is that it's designed to work with multiple different frontends (ITPs), as well as different backends, such as CVC4 [11]. It has support for Isabelle, Coq, and Lean.

QuickCheck [23, 28, 29, 31, 66]: QuickCheck is a type of property based random testing tool. It's not a particular piece of software but has many implementations in many languages. It works by specifying a property that you want to test about a system, and QuickCheck attempts to falsify that the property holds by giving attempting checking to see if there are counterexamples. Often QuickCheck is simply used for sotware testing, as it is in Haskell [24]. However, in the context of theorem provers, it can be used to find counterexamples to statement you might wish to prove. It has support for Agda, Isabelle, PVS, Coq, and ACL2.

This leaves Atelier B, Getfol, HOL Light, HOL4, JAPE, LEO-II, Metamath, Mizar, RedPRL, Twelf, and Z/EVES not having counter example generators.

4.3 Math Notation in libraries

Support of mathematical notation was suggested [87].

The ITPs that use math notation include:

Agda: Agda has support for Unicode characters. And uses Unicode characters in its math library. This often requires special editor integrations to input the Unicode characters. Figure 6 Shows an example of Agda code

```
StateTIApplicative : ∀ (S : I → Set f) {M} →
                     RawMonad M → RawIApplicative (IStateT S M)
StateTIApplicative S Mon = record
  { pure = λ a s → return (a , s)
  ; _⊛_  = λ f t s → do
     (f' , s')  ← f s
     (t' , s'') ← t s'
     return (f' t' , s'')
  } where open RawMonad Mon
```

Figure 6: Agda source code example, from standard library, Category.Monad.State

Isabelle: Isabelle uses LaTeX math commands to represent math notation. This is strong supported within the IDE. The saved plaintext is chown in Figure 7.

```
lemma cInf_abs_ge:
  fixes S :: "'a::{linordered_idom,conditionally_complete_linorder} set"
  assumes "S \<noteq> {}"
    and bdd: "\<And>x. x\<in>S \<Longrightarrow> \<bar>x\<bar> \<le> a"
  shows "\<bar>Inf S\<bar> \<le> a"
proof -
```

Figure 7: Isabelle source code raw, from core libraries, HOL/Archimedian_Field.thy

Whereas the rendered version shown in Figure 8.

```
lemma cInf_abs_ge:
  fixes S :: "'a::{linordered_idom,conditionally_complete_linorder} set"
  assumes "S ≠ {}"
    and bdd: "⋀x. x∈S ⟹ ¦x¦ ≤ a"
  shows "¦Inf S¦ ≤ a"
proof -
```

Figure 8: Isabelle source code rendered, from core libraries, HOL/Archimedian_Field.thy

Lean: Lean has allows representing math notation using Unicode. As shown in Figure 9.

Z/EVES: Z/EVES uses LaTeX commands to prove propositions. The interface, much like Isabelle, allows the user to press buttons on the interface to input notation. As shown in Figure 10.

The ITPs in this study without math notation are ACL2, Atelier B, Coq, Getfol, HOL Light, HOL4, JAPE, LEO-II, Metamath, Mizar, PVS, RedPRL, and Twelf. These theorem provers may also be improved with math notation support.

```
structure comonad extends C ⇒ C :=
(ε' [] : to_functor ⟶ 𝟭 _)
(δ' [] : to_functor ⟶ to_functor ⋙ to_functor)
(coassoc' : ∀ X, nat_trans.app δ' _ ≫ to_functor.map (δ'.app X) = δ'.app _ ≫ δ'.app _ . obviously)
(left_counit' : ∀ X : C, δ'.app X ≫ ε'.app (to_functor.obj X) = 𝟙 _ . obviously)
(right_counit' : ∀ X : C, δ'.app X ≫ to_functor.map (ε'.app X) = 𝟙 _ . obviously)
```

Figure 9: Lean source code rendered, from mathlib, category.monad.basic

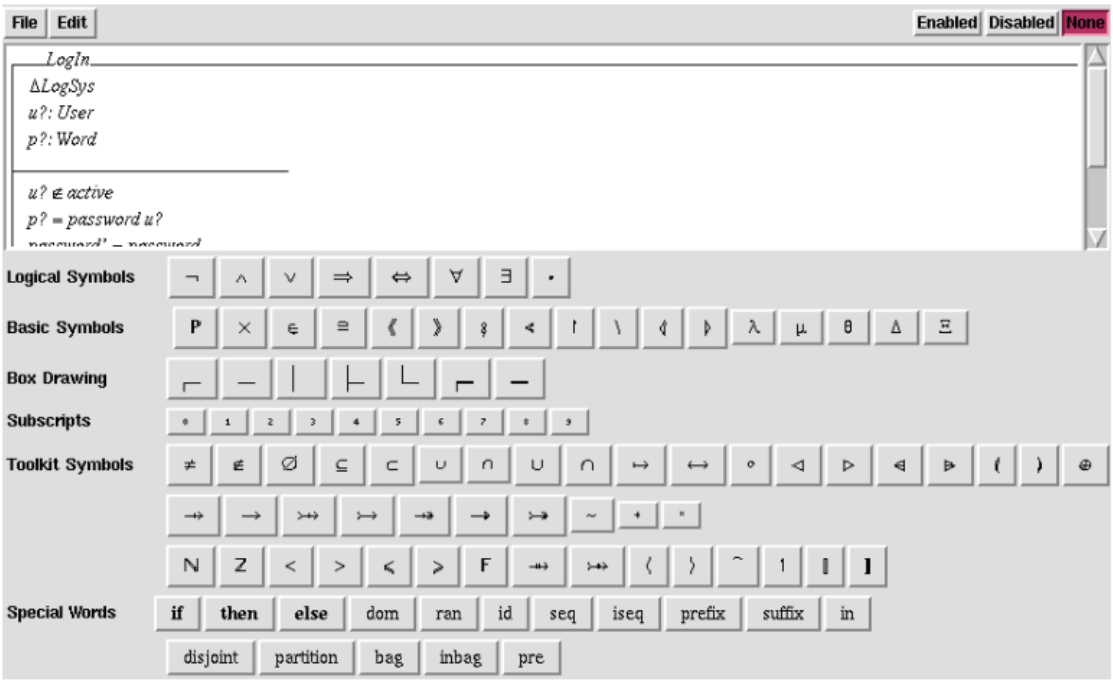

Figure 10: Z/EVES notation support

The ITPs in this study without math notation are ACL2, Atelier B, Coq, Getfol, HOL Light, HOL4, JAPE, LEO-II, Metamath, Mizar, PVS, RedPRL, and Twelf. These theorem provers may also be improved with math notation support.

5 Discussion

We evaluate this review by comparing it to other literature reviews.

It should first stand to say that this is the first living review on ITPs. As of such, this review already has many benefits over the current literature.

The clear improvement is this is the only review on ITPs that automatically updates to reflect the current state of the art. This means it can be referred back to at any time, and even used to track progress on efforts, such as formalizing mathematics.

Due to the availability of web technologies and interactivity, the living review is also much more accessible than a paper one. It allows readers to compare features of ITPs without scouring through large amounts of text or needing to pay large amounts of money to find them. The use of interactive tables and multiple pages in our review mean that the user can extract the knowledge that they are interested in without difficulty.

These benefits of the fact that it is living are great, but even without considering them, this review does build on past reviews.

This is the first review to ever classify the completed scope of ITPs. The classification helps people become aware of a module in their field.

A good place to start is to compare with the literature review we base some of our data on [63]. This review covers Theorem Provers in Formal Methods and what features they have. This literature review covers both Automatic Theorem Provers and Interactive Theorem Provers. For the sake of ITPs, the review has the scope of discovering what features each ITP has. This living review also includes all of the features and compares them. The living review also offers the benefits of being able to check the differences between mathematical libraries. Our contribution therefore, when considering ITPs and not Formal Methods in general, therefore contains a superset of the knowledge in this one. This is however expected, as we used this review as a basis for our one.

Other literature reviews include a survey on the field of Interactive Theorem Proving [56]. This review covers a brief history of ITPs, and a discussion of different calculi present in ITPs. This survey has a larger scope in terms of history, and detailed discussions of types of calculus and achievements. It however, is not systematic, and more represents an introduction to the field of ITPs, and may not be suitable for those currently in the field to understand the current state of the art. Furthermore, this review is difficult to understand without a strong knowledge of logic.

Finally, John Harrison completed a review of the history of ITPs [42]. As the title suggests, this review covers the history of ITPs, which is again outside the scope of this living review. This review is comprehensive in covering the development of ITPs up until 2014. However, at 7 years, it is currently out of date.

5.1 Limitations with current work

As much as this work does make progress on the current state of the field, it is not without limitations. This section covers possible limitations on this work, or items that were deemed out of scope but would be helpful.

One limitation of this work is that although it does represent a review of the field, the review would likely not be suitable as an introduction. Often, reviews allow users unfamiliar with a field to introduce themselves and better understand the topic so that they can make direct work. This review does not do this, and instead tracks the progress of the field, and may be more suitable to people within it.

A second limitation is that the author is not a mathematician by training, and there were a large number of modules that may require help from a professional mathematician to fully classify. Further, this review may contain errors in it due to the author not having training in the field. Living reviews can update over time however, and it is entirely possible that a mathematician could volunteer their time to complete the classification.

A third limitation is that although this is a living review, it does not often reference conference papers or journal articles, even though it possibly could. A more complete living review might include a discussion of the current trends in research, such as the usability research done on a particular theorem prover. We chose to exclude this from the current contribution due to requiring a large amount of manual work to keep up to date. However, we are convinced that such a review would be worth the effort in creating.

5.2 Future Work

During this investigation, several avenues of future work became clear.

Firstly, it was identified that the field of ITPs lacks empirical usability studies. Work in identifying and verifying usability issues with ITPs would do a good job to fill the gap in the literature. Many of the issues that arose in this review could be fantastic candidates.

Secondly, due to the nature of a living review, it is entirely possible to extend the scope over time. For instance, we considered the scope of libraries, counterexample generators and math provers. One possible extension of the review would be collecting large projects done in ITPs, in order to demonstrate their ability and usage in industry settings. This was outside the scope of this living review, but it could be updated in the future. This way a user from industry could consider whether they should invest in ITPs for their software project.

Finally, the author does hope that this may encourage the creation of more living reviews for other topics. A living review has many advantages, one of the most prominent being it is a considerably more accessible way of presenting information in a field, both to professionals and non professionals.

5.3 Summary

ITPs are important due to their ability to assure very high levels of quality in software, which is only needed more as our reliance on technology only increases. However, ITPs have not yet had a large industry uptake, and it has been suggested [49] that usability issues with the ITPs could be the reason.

Motivated to investigate why a user might find the adoption of ITPs difficult, we performed a systematic literature review on usability issues mentioned in literature. We found there was a profound lack of empirical studies on usability about ITPs, and that there was a large number of potential usability problems that could be investigated.

From this systematic literature review, we decided to attempt to shed more light on these usability issues. We did this by creating a living review of ITPs. The review semi-automatically update periodically to reflect the current state of the field. This living review was designed with the intent of being accessible to newcomers, but also to ensure that it doesn't become invalid years down the track.

In this living review, we examined the usability issues of small mathematical scopes, lack of counterexample generators and lack of math notation. We did this by systematically examining math modules created for ITPs, and classified them to the Mathematical Subject Classification. This allows readers to compare ITPs by the areas of mathematics that they cover, helping mathematicians make decisions about ITPs. Further, it tracks progress towards the formalization of different sections of mathematics.

We identified that some ITPs are currently better for different topics. Particularly depending on whether you want are looking to verify software or prove mathematical theorems. However, even if you wanted to prove mathematical theorems, different ITPs were better in different situations depending on the mathematical field.

We also directly examined the ITPs for counterexample generators and support for math notation, and included this within our living review.

This review will be updated periodically and therefore will remain a reference for the field.

Bibliography

[1] 2008. Shortest paths. *Algorithms and data structures: The basic toolbox*. Springer Berlin Heidelberg. 191–215.

[2] Adams, M. 2010. Introducing HOL zero. *Mathematical software – ICMS 2010* (Berlin, Heidelberg, 2010), 142–143.

[3] Aitken, J.S., Gray, P., Melham, T. and Thomas, M. 1998. Interactive Theorem Proving: An Empirical Study of User Activity. *Journal of Symbolic Computation*. 25, 2 (1998), 263–284. DOI:https://doi.org/https://doi.org/10.1006/jsco.1997.0175.

[4] Aitken, S. and Melham, T. 2000. An analysis of errors in interactive proof attempts. *Interacting with Computers*. 12, 6 (2000), 565–586. DOI:https://doi.org/10.1016/S0953-5438(99)00023-5.

[5] Angiuli, C., Cavallo, E., Hou (Favonia), K.-B., Harper, R. and Sterling, J. 2018. The RedPRL proof assistant (invited paper). *Proceedings of the 13th International Workshop on logical frameworks and meta-languages: Theory and practice, Oxford, UK, 7th July 2018* (2018), 1–10.

[6] Asperti, A. and Coen, C.S. 2010. Some Considerations on the Usability of Interactive Provers. *Proceedings of the 10th ASIC and 9th MKM International Conference, and 17th Calculemus Conference on Intelligent Computer Mathematics* (Berlin, Heidelberg, 2010), 147–156.

[7] Asperti, A., Sacerdoti Coen, C., Tassi, E. and Zacchiroli, S. 2007. User Interaction with the Matita Proof Assistant. *Journal of Automated Reasoning*. 39, 2 (Aug. 2007), 109–139. DOI:https://doi.org/10.1007/s10817-007-9070-5.

[8] Aspinall, D. and Kaliszyk, C. 2016. Towards Formal Proof Metrics. *Fundamental Approaches to Software Engineering* (Berlin, Heidelberg, 2016), 325–341.

[9] Associate Editors of Mathematical Reviews and zbMATH 2020. MSC2020-mathematics subject classification system.

[10] Barras, B., Tankink, C. and Tassi, E. 2015. Asynchronous Processing of Coq Documents: From the Kernel up to the User Interface. *Interactive Theorem Proving* (Cham, 2015), 51–66.

[11] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A. and Tinelli, C. 2011. CVC4. *Computer aided verification* (Berlin, Heidelberg, 2011), 171–177.

[12] Becker, H., Bos, N., Gavran, I., Darulova, E. and Majumdar, R. 2021. Lassie: HOL4 Tactics by Example. *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New York, NY, USA, 2021), 212–223.

[13] Beckert, B. and Grebing, S. 2012. Evaluating the Usability of Interactive Verification Systems. *COMPARE* (2012).

[14] Beckert, B. and Grebing, S. 2015. Interactive Theorem Proving - Modelling the User in the Proof Process. *Bridging@CADE* (2015).

[15] Beckert, B., Grebing, S. and Böhl, F. 2015. A Usability Evaluation of Interactive Theorem Provers Using Focus Groups. *Software Engineering and Formal Methods* (Cham, 2015), 3–19.

[16] Beckert, B., Grebing, S. and Ulbrich, M. 2017. An Interaction Concept for Program Verification Systems with Explicit Proof Object. *Hardware and Software: Verification and Testing* (Cham, 2017), 163–178.

[17] Benzmüller, C., Sultana, N., Paulson, L. and Theiss, F. 2015. The higher-order prover leo-II. *Journal of Automated Reasoning*. 55, (Dec. 2015). DOI:https://doi.org/10.1007/s10817-015-9348-y.

[18] Berman, B.A. 2014. *Development and user testing of new user interfaces for mathematics and programming tools*. University of Iowa.

[19] Bertot, Y. and Castéran, P. 2004. *Interactive theorem proving and program development: Coq'art: The calculus of inductive constructions*. Springer Berlin Heidelberg.

[20] Blanchette, J.C. and Nipkow, T. 2010. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. *Interactive theorem proving* (Berlin, Heidelberg, 2010), 131–146.

[21] Bornat, R. 2005. *Proof and disproof in formal logic: An introduction for programmers*. Oxford University Press.

[22] Bourke, T., Daum, M., Klein, G. and Kolanski, R. 2012. Challenges and Experiences in Managing Large-Scale Proofs. *Intelligent Computer Mathematics* (Berlin, Heidelberg, 2012), 32–48.

[23] Bulwahn, L. 2012. The new quickcheck for isabelle. *Certified programs and proofs* (Berlin, Heidelberg, 2012), 92–108.

[24] Claessen, K. and Hughes, J. 2000. QuickCheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.* 35, 9 (Sep. 2000), 268–279. DOI:https://doi.org/10.1145/357766.351266.

[25] Clearsy The industrial tool to efficiently deploy the b method - atelier b.

[26] Cruanes, S. and Blanchette, J. 2016. Extending nunchaku to dependent type theory. *Electronic Proceedings in Theoretical Computer Science*. 210, (Jun. 2016), 3–12. DOI:https://doi.org/10.4204/EPTCS.210.3.

[27] Czaplicki, E. Elm book.

[28] Dénès, M. and Pierce, B.C. 2014. QuickChick: Property-based testing for coq. (2014).

[29] Dybjer, P., Haiyan, Q. and Takeyama, M. 2003. Combining testing and proving in dependent type theory. *Theorem proving in higher order logics* (Berlin, Heidelberg, 2003), 188–203.

[30] Eastaughffe, K. 1998. Support for Interactive Theorem Proving: Some Design Principles and Their Application. (1998).

[31] Eastlund, C. 2009. DoubleCheck your theorems. *Proceedings of the eighth international workshop on the ACL2 theorem prover and its applications* (New York, NY, USA, 2009), 42–46.

[32] G. Gonthier 2008. Formal proof of the Four-Color theorem. *Notices of the American Mathematical Society*.

[33] Giunchiglia, F. and Cimatti, A. 1994. Introspective metatheoretic reasoning. *Logic program synthesis and transformation — meta-programming in logic* (Berlin, Heidelberg, 1994), 425–439.

[34] Grabowski, A., Kornilowicz, A. and Naumowicz, A. 2010. Mizar in a nutshell. *Journal of Formalized Reasoning*. 3, 2 (2010), 153–245. DOI:https://doi.org/10.6092/issn.1972-5787/1980.

[35] Grebing, S., Klamroth, J. and Ulbrich, M. 2020. Seamless Interactive Program Verification. *Verified Software. Theories, Tools, and Experiments* (Cham, 2020), 68–86.

[36] Grebing, S. and Ulbrich, M. 2020. Usability Recommendations for User Guidance in Deductive Program Verification. *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY*. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, and M. Ulbrich, eds. Springer International Publishing. 261–284.

[37] Green, T.R.G. and Petre, M. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing*. 7, 2 (1996), 131–174.

[38] Grov, G. and Lin, Y. 2018. The Tinker tool for graphical tactic development. *International Journal on Software Tools for Technology Transfer*. 20, 2 (Apr. 2018), 139–155. DOI:https://doi.org/10.1007/s10009-017-0452-7.

[39] Hähnle, R. and Huisman, M. 2019. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. *Computing and Software Science: State of the Art and Perspectives*. B. Steffen and G. Woeginger, eds. Springer International Publishing. 345–373.

[40] HALES, T., ADAMS, M., BAUER, G., DANG, T.D., HARRISON, J., HOANG, L.T., KALISZYK, C., MAGRON, V., MCLAUGHLIN, S., NGUYEN, T.T. and al., et 2017. A FORMAL PROOF OF THE KEPLER CONJECTURE. *Forum of Mathematics, Pi*. 5, (2017), e2. DOI:https://doi.org/10.1017/fmp.2017.1.

[41] Harrison, J. 2009. HOL light: An overview. *Theorem proving in higher order logics* (Berlin, Heidelberg, 2009), 60–66.

[42] Harrison, J., Urban, J. and Wiedijk, F. 2014. History of interactive theorem proving. *Handbook of the History of Logic*. 135–214.

[43] Hentschel, M. 2016. *Integrating Symbolic Execution, Debugging and Verification*. Technische Universität Darmstadt.

[44] Hentschel, M., Hähnle, R. and Bubel, R. 2016. An Empirical Evaluation of Two User Interfaces of an Interactive Program Verifier. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2016), 403–413.

[45] Hentschel, M., Hähnle, R. and Bubel, R. 2016. The Interactive Verification Debugger: Effective Understanding of Interactive Proof Attempts. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2016), 846–851.

[46] Hunter, C., Robinson, P. and Strooper, P. 2005. Agent-Based Distributed Software Verification. *Proceedings of the Twenty-Eighth Australasian Conference on Computer Science - Volume 38* (AUS, 2005), 159–164.

[47] Jean Karim Zinzindohoué, J.P., Karthikeyan Bhargavan 2017. *HACL*: A Verified Modern Cryptographic Library.*

[48] Kadoda, G. 2000. *A Cognitive Dimensions view of the differences between designers and users of theorem proving assistants.*

[49] Kadoda, G.F. 1997. *Formal software development tools: An investigation into usability.*

[50] Kadoda, G.F., Stone, R. and Diaper, D. 1999. Desirable features of educational theorem provers - a cognitive dimensions viewpoint. *PPIG* (1999).

[51] Kaufmann, M., Manolios, P. and Moore, J.S. 2000. *Computer-aided reasoning: An approach*. Springer US.

[52] Kawabata, H., Tanaka, Y., Kimura, M. and Hironaka, T. 2018. Traf: A Graphical Proof Tree Viewer Cooperating with Coq Through Proof General. *Programming Languages and Systems* (Cham, 2018), 157–165.

[53] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S. 2009. SeL4: Formal Verification of an OS Kernel. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), 207–220.

[54] Leroy, X. 2009. Formal verification of a realistic compiler. *Communications of the ACM*. 52, 7 (2009), 107–115.

[55] Lin, Y., Grov, G. and Arthan, R. 2016. Understanding and maintaining tactics graphically OR how we are learning that a diagram can be worth more than 10K LoC. *Journal of Formalized Reasoning*. 9, 2 (Dec. 2016), 69–130. DOI:https://doi.org/10.6092/issn.1972-5787/6298.

[56] Marić, F. 2015. A survey of interactive theorem proving. *Zbornik radova*. (Jul. 2015).

[57] Matsakis, N.D. and Klock, F.S. 2014. The Rust Language. *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (New York, NY, USA, 2014), 103–104.

[58] McConnell, S. 2004. *Code Complete, Second Edition*. Microsoft Press.

[59] Mitsch, S. and Platzer, A. 2017. The KeYmaera X Proof IDE - Concepts on Usability in Hybrid Systems Theorem Proving. *Electronic Proceedings in Theoretical Computer Science*. 240, (Jan. 2017), 67–81. DOI:https://doi.org/10.4204/eptcs.240.5.

[60] Moore, J.S. and Zhang, Q. 2005. Proof pearl: Dijkstra's shortest path algorithm verified with ACL2. *Theorem proving in higher order logics* (Berlin, Heidelberg, 2005), 373–384.

[61] Moura, L. de, Kong, S., Avigad, J., Doorn, F. van and Raumer, J. von 2015. The Lean Theorem Prover (System Description). *Automated Deduction - CADE-25* (Cham, 2015), 378–388.

[62] Nagashima, Y. and He, Y. 2018. PaMpeR: Proof Method Recommendation System for Isabelle/HOL. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2018), 362–372.

[63] Nawaz, M.S., Malik, M., Li, Y., Sun, M. and Lali, M.I.U. 2019. A Survey on Theorem Provers in Formal Methods. (2019).

[64] Norell, U. 2009. Dependently typed programming in agda. *Proceedings of the 4th international workshop on types in language design and implementation* (New York, NY, USA, 2009), 1–2.

[65] Norman Megill 2007. *Metamath: A Computer Language for Pure Mathematics*. Lulu Press USA.

[66] Owre, S. 2006. Random testing in PVS. (2006).

[67] Pablo Le Hénaff nunchaku-lean · GitLab.

[68] Paulson, L.C. ed. 1994. *Isabelle: A generic theorem prover*. Springer Berlin Heidelberg.

[69] Pfenning, F. and Schuermann, C. 2002. Twelf's User Guide.

[70] R. Arthan 2005. ProofPower–SLRP user guide. Technical report. *Lemma 1 Limited*.

[71] Ringer, T., Sanchez-Stern, A., Grossman, D. and Lerner, S. 2020. REPLica: REPL Instrumentation for Coq Analysis. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New York, NY, USA, 2020), 99–113.

[72] Roe, K. and Smith, S. 2016. CoqPIE: An IDE Aimed at Improving Proof Development Productivity. *Interactive Theorem Proving* (Cham, 2016), 491–499.

[73] Rushby, J. 2006. Tutorial: Automated formal methods with PVS, SAL, and yices. *Fourth IEEE international conference on software engineering and formal methods (SEFM'06)* (2006), 262–262.

[74] Saaltink, M. 1997. The z/EVES system. *ZUM '97: The z formal specification notation* (Berlin, Heidelberg, 1997), 72–85.

[75] Satyanarayan, A., Moritz, D., Wongsuphasawat, K. and Heer, J. 2017. Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*. 23, 1 (Jan. 2017), 341–350. DOI:https://doi.org/10.1109/TVCG.2016.2599030.

[76] Shams, Z., Sato, Y., Jamnik, M. and Stapleton, G. 2018. Accessible Reasoning with Diagrams: From Cognition to Automation. *Diagrammatic Representation and Inference* (Cham, 2018), 247–263.

[77] Shao, Z. 2010. Certified software. *Commun. ACM*. 53, 12 (Dec. 2010), 56–66. DOI:https://doi.org/10.1145/1859204.1859226.

[78] Slind, K. and Norrish, M. 2008. A brief overview of HOL4. *Proceedings of the 21st international conference on theorem proving in higher order logics* (Berlin, Heidelberg, 2008), 28–32.

[79] Spichkova, M. and Simic, M. 2017. Human-centred analysis of the dependencies within sets of proofs. *Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 21st International Conference, KES-20176-8 September 2017, Marseille, France*. 112, (Jan. 2017), 2290–2298. DOI:https://doi.org/10.1016/j.procs.2017.08.256.

[80] Tassi, E. 2008. *Interactive theorem provers: Issues faced as a user and tackled as a developer*. alma.

[81] Torlak, E. and Jackson, D. 2007. Kodkod: A relational model finder. *Tools and algorithms for the construction and analysis of systems* (Berlin, Heidelberg, 2007), 632–647.

[82] Wenzel, M. 2014. Asynchronous User Interaction and Tool Integration in Isabelle/PIDE. *Interactive Theorem Proving* (Cham, 2014), 515–530.

[83] Wenzel, M. 2011. Isabelle as Document-Oriented Proof Assistant. *Intelligent Computer Mathematics* (Berlin, Heidelberg, 2011), 244–259.

[84] Wenzel, M. 2006. Structured Induction Proofs in Isabelle/Isar. *Mathematical Knowledge Management* (Berlin, Heidelberg, 2006), 17–30.

[85] Wiedijk, F. 2007. The QED manifesto revisited. (2007).

[86] Wohlin, C. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. *Proceedings of the 18th international conference on evaluation and assessment in software engineering* (New York, NY, USA, 2014).

[87] Zacchiroli, S. 2007. *User interaction widgets for interactive theorem proving*. alma.