iProve: A logical and inductive proof assistant

1 Introduction

iProve is a web-based application with the goal of aiding first-year students in the Imperial Department of Computing and improving their understanding of the concepts taught in the “Logic and Reasoning" module (COMP40018/ COMP40012) [henceforth abbreviated to "module"]. The application provides a platform for constructing proofs through validity-preserving steps, including standard logic techniques (such as natural deduction) and structural induction, taught during the module. Proofs are displayed in a way that clearly illustrates their logical structure, and the tool enables users to navigate a proof both forwards and backwards, similar to the process of constructing a proof by hand. A key feature of iProve is its ability to accommodate a range of granularity levels in the proof steps, allowing users to choose the level of rigour depending on their desired learning objectives.

1.1 Initial Problem

The "Logic and Reasoning" module is the first module which introduces university-level theoretical computer science, focusing on proof writing and the thought process behind programs. Understanding all the new concepts can be challenging for new students, especially as it can be difficult to get feedback on their work, as the correctness of a proof must be evaluated rather than compared to an expected answer. It is crucial for students to have a strong understanding of these concepts, as it sets the foundation for their degree.

The goal of iProve is to provide a tool that can be blended with the existing teaching methods to give students more opportunities to practice and apply their knowledge. This should provide instant feedback on their work, allowing them to identify and correct any mistakes in their reasoning as well as exploring alternative solutions. Ensuring the tool can accurately interpret the formal language of logic taught to students and determine the correctness of formulated proofs is critical. As the target user group are first year computer science students, usability is of utmost importance. The user-friendly interface should allow students to easily input, review and visualise their work.

1.2 Existing Solutions

The department already has a tool called Pandora that is used to support students in their studies. Pandora only supports natural deduction proofs, with no option to perform short proof search (so it requires students to be very explicit about every proof step). Thus, Pandora only provides support for material taught in the first term. In fact, the original paper which introduced Pandora suggested that a proof assistant to help students with the "Reasoning About Programs" section of the module would be a future direction [1].

Moreover, the input is unintuitive and cumbersome, making it confusing for students to use. This is compounded by the tool also requiring outdated software dependencies. Pandora is an old tool and a user-friendly update is needed.

1.3 Our contributions

In light of these factors, it was necessary to create a new tool, **iProve**, that addresses the limitations of the existing solutions. iProve offers a range of features to support students, with the key technical features being:

- Checking the correctness of a First-Order logic proofs: iProve can help students verify that their proof follows the rules of logic taught in the "Logic" module. To achieve this, iProve checks logical implications in a proof sequentially, by transpiling statements made in the particular logic of the module into SMT-LIB code, which is then checked using the Z3 theorem prover.

- Checking correctness of induction principles: iProve can generate induction principles over abstract data types, lists, and numbers, as taught in the “Reasoning" module, and includes an interface whereby students can enter their idea for the principle, and check its correctness before using it in a proof.

- Visualising the proof: iProve presents the proof as a graph, allowing students to see exactly how the implications in the proof work and identify any unnecessary elements they included.

- Flexible granularity: iProve allows students to be as specific as they want to be in their proof, without requiring them to be overly pedantic (they can include as many intermediate steps as they feel is necessary for their, or a marker’s, understanding). This allows them to see whether their proof is correct, even if it is different to the provided sample solution to an exercise.

In addition to the features above, iProve also offers a form of communication between students and lecturer. Students can import and export proofs from iProve, saving them in a downloadable JSON format. This feature will eventually allow the lecturer to set assignments in iProve format that students can then solve and submit as finished proofs (e.g. through Scientia or other tools used by the department for assessment).

2 Links to production

- PaaS deployment (Heroku): `https://iprove.herokuapp.com/`

- Gitlab repository: `https://gitlab.doc.ic.ac.uk/g226002130/iprove`

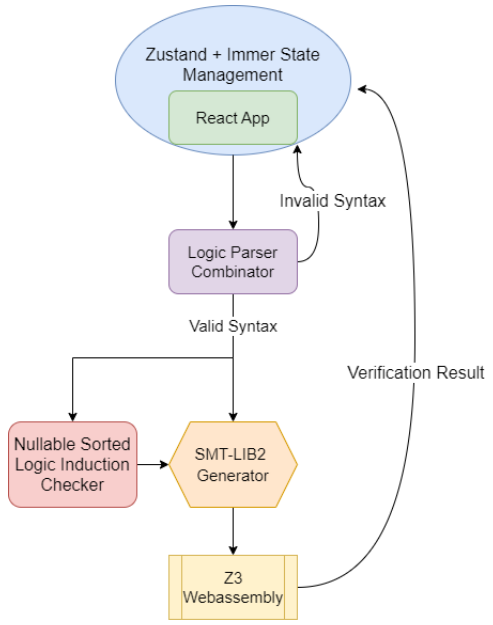

Figure 1: High-Level architecture diagram of iProve

3 Overview of iProve

3.1 Architecture Overview

Our goal was to create an easily accessible product, so we opted for a web-based application in React. The burgeoning complexity of managing state mandated using a React State Management tool; we opted for Zustand and Immer. Our application includes a parser for processing user input. The Abstract Syntax Tree (AST) generated is then piped into an induction checker which uses multiple engines to check induction validity. Finally, if all checks pass, a generator creates SMT-LIB2 output which is piped into the Z3 WebAssembly. The output of Z3 is then rendered back to the user.

3.2 Syntax and Semantics

The language of iProve is designed to follow closely the language introduced in the lectures, but it is also inspired from Haskell, to allow for more complex constructions. iProve supports all logical operators and quantifiers taught in the module, using the keyboard equivalences. Note that after typing a statement with valid syntax, this is displayed using the rendered symbols, to improve readability.

Logical Operator	iProve Syntax	Rendered Symbol
Forall	`FA`	$\forall$
There Exists	`EX`	$\exists$
Not	`~`	$\neg$
Implies	`->`	$\rightarrow$
If and only if	`<->`	$\leftrightarrow$
And	`&`	$\wedge$
Or	`||`	$\vee$

Figure 2: Supported Logical Operators

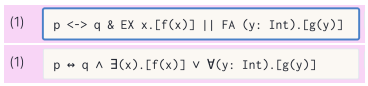

Figure 3: Statement before and after enhancements

To ensure the logic system we use is sound, we have elaborated a document detailing all the constructions supported by iProve (appended at the end of the report).

3.3 Declarations and types sidebar

Similar to many programming languages, before running any checks on a proof, the user needs to declare the elements used. In iProve, this is done by adding statements to the left sidebar. The sidebar supports defining variables, custom data types, functions and predicates. The user must declare functions they intend to use, along with their type signatures. The parser then automatically checks their validity. An example interaction is provided below:

Figure 4: Example of interaction with sidebar

3.4 Core components of user interface

One of the key features of iProve is the novel way of representing the proof. To allow the user to focus on the essential implications in the proof, an iProve proof is represented as a directed graph, with multiple *node types* (containing multiple *statements*), which are joined by *edges*, representing implications between nodes.

3.4.1 Node types

iProve offers four types of nodes, representing different parts of the proof:

1. **Given node & Goal node:** These are start and end points for the proof, added to offer clarity to the proof that the user inserts into iProve. The idea behind these was to make the user think about what is given in an exercise and what is the goal they want to reach.

2. **Proof node:** These are the main building blocks, the internal nodes of the graph, that the user will need to complete a proof. Every proof node should act like a self-contained proof, that takes in a set of

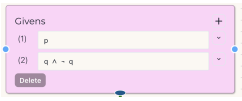

Figure 5: Given node example

givens and reaches a set of goals. As these are internal nodes, to build a complete proof, the givens of the proof node need to be obtained from previous results (either givens or from the "goals" section of other proof nodes). A step inside a proof node need to be justified using one or more of the steps before it, which the user needs to specify.

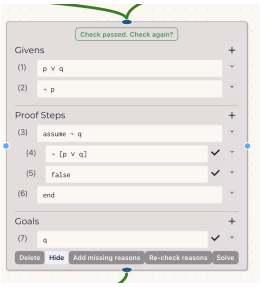

Figure 6: Proof node example

3. **Induction node:** These are added to separate logical implications from induction principles, and can be used to check the correctness of an induction principle, based on material presented in the "Reasoning" part of the module. Here, the user specifies the type on which induction will be done, the relevant base cases and induction cases and the induction goal. To complete a proof that requires induction, the base cases and induction cases need to be justified using results from other nodes, similar to the givens in a proof node.

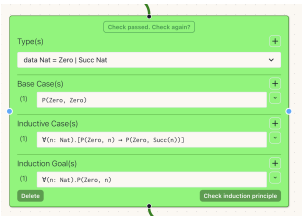

Figure 7: Valid induction node example

3.4.2 Edge types

iProve offers three types of edges, depending on the state of a the implication it represents:

1. **Unchecked edge:** Edges for which the implications have not been verified yet. These are added by the user before they are ready to verify their work.
2. **Valid edge:** Edges that have passed the Z3 checks.
3. **Invalid edge:** Edges that have failed the Z3 checks and need to be fixed to get a complete proof.

Connecting nodes with edges means that the goals of the source nodes should imply the givens of the target nodes. Until the user triggers the Z3 check the edge is unchecked (and represented as black), and then it changes according to the Z3 output (green if valid and red if invalid).

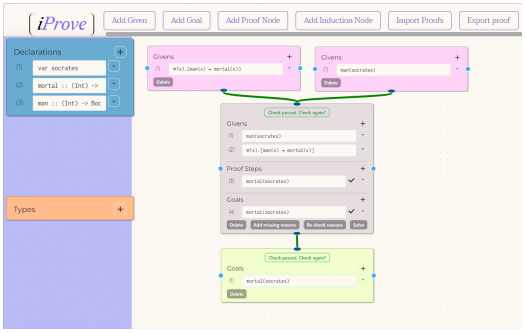

Figure 8: An example proof checked in iProve

3.5 Language and Parsing

The first concern with a tool such as iProve is converting user input into a format understandable by our proof checker. We chose the `ts-parsec` parser combinator library due to past experience with this paradigm, and the flexibility it provided to incrementally add new supported syntax. The syntax of each proof line is broadly that of many-sorted first-order logic, augmented by Haskell-style function definitions (including complete guards and pattern-matching support) and inductive data type definitions. Iterative user- and developer-testing also led to the addition of many aliases and snippets of syntactic sugar to ease proving in the language. The parser is designed to give useful and precise syntax error feedback, and successfully parsed ASTs are returned to the logic back-end.

3.6 Nullable Sorted Logic (NSL)

Whilst students are taught many-sorted FOL on the course, the existence of polymorphic inductive types and recursively-defined functions means that our logic system is not, in fact, completely first-order. This is primarily due to the fact that recursively-defined functions are borrowed from λ-style logics, and can thus be non-terminating or otherwise partial. Proof search is very difficult on these logics in general, and projects which do exist have very little modern infrastructure support. As a result, we remain using the Z3 FOL prover, and have developed our own variant of FOL (dubbed **NSL**) on top of this. It's core difference to standard FOL lies in the fact that satisfaction of a formula can only be derived if every function called during the derivation is well-defined. In practice, students can mark functions as `partial` (they are total by default to prevent forcing students to inductively prove totality of every function), and the NSL engine will inspect every term used by the student, appending recursive definedness conditions to each proof step before being sent to the Z3 solver. This is implemented with a *short-circuit semantics*

for certain special operators, e.g. $\forall x.[P(x) \to Q(x)]$ will only enforce definedness of $Q(x)$ for x where $P(x)$ holds.

3.7 SMT Generator

The Z3 solver accepts queries in the form of SMT satisfiability questions, using the SMT-LIB language. In many cases, our ASTs admit simple translation to SMT - e.g. the conversion from $P \to Q$ to `(implies P Q)` is trivial once parsed into our internal representation. However, we support many Haskell-style augmentations to the basic syntax which require much more involved translation. One challenge was implementing the natural deduction step of forall-introduction. To do this, we need to create a new "block scope" which must be translated into $\forall$ statements wrapping each term in the scope. `data` statements are trivial using SMT-LIB's `(declare-datatypes)` command, and we can use the `match` command to enable pattern matching (since destructuring tuples and lists can be expressed in terms of matching constructors of algebraic datatypes). A more challenging problem was that all recursive functions in SMT-LIB must be defined in a single define-funs-rec block in order to permit mutual recursion, and there are no guards built into the language. This required us to build a `LogicInterface` engine which collates all function definitions into a single SMT block, and folds all guards and patterns for each function into a single nested chain of `(let)` bindings and `(if)` expressions.
Finally, we initialize the prover context with our builtin types (Maybe and List), send each given in the form of an `assert` command, and give a negated assertion of our goal before returning a verdict using `(check-sat)`.

3.7.1 Builtin `List` operations

Inductively defined lists have no builtin operations in Z3. This presents an additional challenge, since we need to be able to concatenate, slice, and index lists in order for our syntax to completely cover the content taught in the reasoning about programs module. However, Z3 does not have any support for defining polymoprhic/generic functions. Thus, we engineered a primitive type inference system which traverses the AST and takes all the user's type declarations (as well as Z3 type information) into account to derive the types of every term to which a list operation is applied. We store our list operations as SMT-LIB code in JS format strings, and apply ad-hoc polymorphism by substituting each type into the required templates, and inserting one function definition per type used.

3.8 Induction Principles

Induction principles can be checked over two kinds of type: inductive data structures, and bounded integers. Since lists are inductively defined, we need no extra (non-syntactic) methods for generating or checking them. A rewrite engine traverses ASTs corresponding to the propositions to be proven inductively, and whenever the induction variable is encountered, it will generate a base case for non-recursive constructors, and an inductive case for recursive constructors. The generation engine can handle mutual induction by design. For bounded integers, the simple single base and inductive case is generated separately, but still used by the same rewrite engine facilitating induction over numbers as part of mutual induction. The check is performed by simply asking Z3 if the principles are equivalent, however we must append an extra `Underdeterminer` constructor to each type inducted over before passing it to the solver. This ensures that Z3 is unable to prove either principle by itself and short-circuit the process.

3.9 Browser Support

iProve, unlike many other proof assistants, is entirely browser-based, meaning that it can be accessed anywhere with an internet connection without the need to download a supporting application. Furthermore, all of iProve's functionality is client-side - once loaded, connection can be severed and iProve will continue to function, with Z3 builds running in WebAssembly. As a result, the backend only functions to serve the webpage.

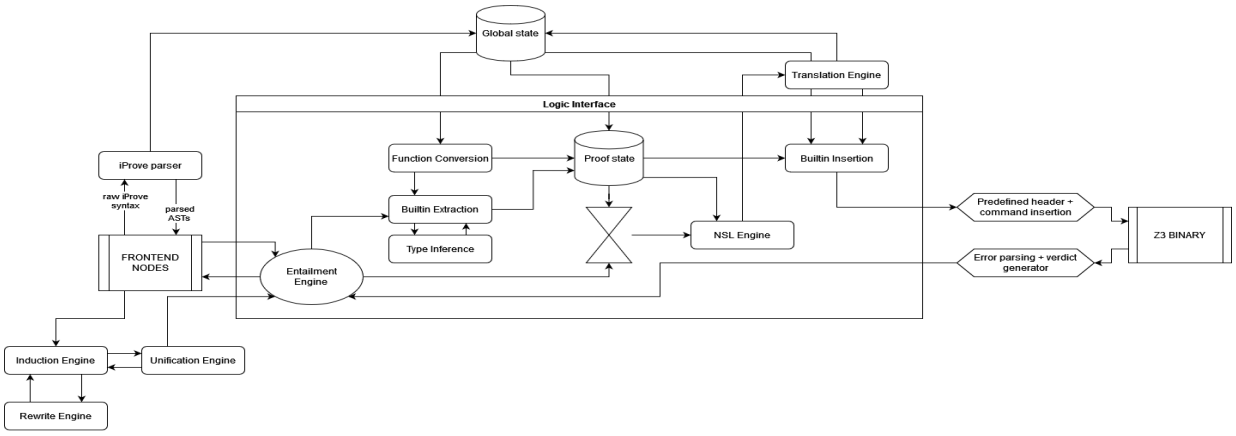

Figure 9: Low-level overview of back-end data flow

4 Evaluation

4.1 Evaluation Criteria

Our mission with iProve was to create a platform that would allow first-year students to check their proofs to enhance their learning of the Logic and Reasoning course. This mission statement guided us in selecting certain Key Performance Indicators (KPIs) to focus on during our evaluation of the final solution. The KPIs we selected were: correctness, functionality coverage, ease of use, and availability. Firstly it was of utmost importance that our product be correct, so that students can rely on it. We also wanted to improve on existing systems by providing more support for the material taught in the module. Ease-of-use was also a key goal as adoption among students and lecturers is predicated on this. Indeed we had heard that many students avoided Pandora due to the initial setup process so it was necessary to improve on this aspect.

4.2 Correctness and Completeness

We had multiple approaches to ensure the correctness of our system. Firstly, we used unit tests to ensure the correctness of individual components. We used the *jest* framework and set up testing as a stage in the pipeline that runs on every master branch commit and merge request.

To perform system-wide testing, we implemented a wide range of exercises from the tutorial sheet in iProve . We have exported and saved in JSON format more than 30 proofs which cover all the topics taught in the module. The exported files can be found here: `https://gitlab.doc.ic.ac.uk/g226002130/iprove/-/tree/master/Examples` These proofs were done with different levels of granularity, to ensure iProve's robustness.

Finally, we created a document, which is added to the end of this report, that outlines how the Nullable Sorted Logic engine works which was checked by our supervisor for its correctness.

4.3 User Feedback

Through extensive testing with our users, we gathered valuable feedback that helped us identify important usability improvements. We addressed these improvements before the final evaluation. Some of the most significant changes include separating types and declarations from proof bodies, adding in-app syntax reminders, and improving error messages. At this stage, we also tested different versions of teaching material for the iProve onboarding. From this, we generated an iProve handbook for students, which is included in the GitLab repository.

4.4 Final User Evaluation

During the user evaluation, we introduced iProve to two PMT groups (18 first year students). They were given a walkthrough of the product at the end of their PMT sessions and allowed some time to experiment with the product - they were provided with some exercises which they attempted to solve in iProve. Each student then completed a questionnaire in which each question was linked to a category. This yielded an average score for each category out 10.

Category	Pandora	iProve
Onboarding	4.1	8.2
Usability	5.6	7.4
Usefulness	7.3	8.4
Likeliness to Recommend	5.2	7.1
Overall Satisfaction	5.4	7.2

These results clearly show that students are pleased with iProve and prefer it over Pandora. Some students remained behind on a voluntary basis to be interviewed about how they would use iProve and to hear general thoughts. From these discussions, we discovered that students would want to adopt iProve as a checking tool, but that it would not replace pen and paper proofs as they would still want to replicate exam conditions when first creating the proof.

This could influence potential future development of iProve, for example by adding a feature for scanning in hand-written proofs, and exploring ways to reduce repetitive type declarations. After discussions with our supervisor, we have decided against implementing polymorphic type inference for automatic predicate typing, as forcing the user to reason about a function's types would aid the teaching process.

5 Conclusion

In conclusion, iProve is a complete and correct proof creation tool that students can use to improve their understanding of the "Logic and Reasoning" material. Additionally, its novel approach to proof visualization enables students to understand the implications of a proof in a more intuitive way, leading to a deeper understanding of the material. With its easy-to-use interface, comprehensive support for the proofs covered in the module, and accurate proof-checking capabilities, iProve would be a valuable addition to the tools available for teaching this subject.

References

[1] K. Broda, J. Ma, G. Sinnadurai, and A. Summers. Pandora: A reasoning toolbox using natural deduction style. *Logic Journal of IGPL*, 15(4):293–304, 2007.