The Incredible Proof Machine

Invited Talk Paper

Joachim Breitner
Karlsruhe Institute of Technology
Karlsruhe, Germany
breitner@kit.edu

ABSTRACT

Bringing the joy and excitement of interactive theorem provers to high school students is a challenge, because having to learn the rules of the syntax, and manually checking handwritten proofs is just not fun. Thus I created the Incredible Proof Machine, which allows students to conduct proofs by dragging blocks and wiring up them up to "proof graphs", all using just the mouse. The result is a surprisingly addictive game-like experience that lures students (and non-students) to prove statements of propositional and predicate logic; or any other natural-deduction based calculus their instructor wants to teach. In this talk, I will explain the motivation behind the Proof Machine and its design decisions, demonstrate the user interface and its various features, show how to define your own tasks and even logics and outline the correspondence between these proof graphs and conventional natural deduction derivation trees.

Preamble

This paper is a transcript of the talk (as planned) given at the LFMTP workshop in Porto. It is necessarily incomplete, as during the talk, I made use of the blackboard, did live-coding and live demonstrations of the program, but it can still convey the message to some extent. It obviously also differs from the talk (as held).

Italics in the following text indicate "stage directions".

1. INTRODUCTION

Thank you for the introduction, and for the invitation to speak here. I am very flattered that what started as a one-shot project turned into something academically noteworthy.

Shortly, I will show you the Incredible Proof Machine, which is a visual theorem prover, first from the user's perspective, then from the point of view of an educator who wants to employ the Proof Machine in his courses, and finally I'll discuss some aspects of the implementation and theoretical results.

LFMTP June 23-23, 2016, Porto, Portugal

ACM ISBN 978-1-4503-4777-8/16/06.

DOI: http://dx.doi.org/10.1145/2966268.2966273

But first, I'd like to tell why I created the Proof Machine in the first place, and why it looks the way it looks.

2. HISTORY

Last October, I had the opportunity to hold a weekend-long workshop for highschool students. These students received a scholarship for promising students with migration background[1], and I could assume an above-average interest in science; mathematics and computer science in particular.

I was free to choose the topic of my workshop, and I wanted to let them know about the joy of performing proofs, in particular formal, rigorous proofs.

Now, doing that on paper is not much fun until you are quite sure of the rules of the game. Until then, the students would have suffer from the demotivation of having to show their experiments to someone and being told that they are wrong.

It is much less daunting if they could use an theorem prover, where they can explore things and get immediate feedback. I am sure you would not be here if you would not agree that interactive theorem proving is fun, addictive and the world's geekiest computer game[2]. I have myself learned about this somewhat late in my studies in Karlsruhe, and have been doing plenty of theorem proving since then; mostly using Isabelle, but also Agda and Coq.

Wouldn't it be great if like-minded students could start much earlier?

But obviously, there is little point in just telling them: "Here, install Isabelle and have fun!". They would get completely lost in the syntax of Isar, in the syntax of HOL, in the proof methods. Remember, these students have no prior knowledge of logic, or serious proving, and even parsing a term is a challenge. All in all, this would fare no better than making them do logic on paper, and is certainly not an option for a one-weekend voluntary workshop.

3. DESIGN

So I set out to create an interactive theorem prover that would allow them, without any prior knowledge of the syntax and the rules of logic, to do their first, simple, but real and rigorous proofs – if only by accident!

I stepped back from the proof representations that we usually use, such as natural language proofs (a literary genre

[1] https://www.start-stiftung.de/stipendium.html

[2] By the way, I am still searching for the original source of that quote.

of it own) or proof trees (quite rigid) and thought about the – or, let me say, my – mental picture of a proof.

To me, when I prove something, I have some facts that I believe or assume to be true, such as assumptions or axioms. They are floating around somewhere here. *Draws two rectangles on the left.* And then there are propositions that I want to prove. Quite similar, but they are maybe around here... *Draws two rectangles on the right.*

And now I need to make a connection here, and when I say connection, I mean that quite literally. For example, if what I want to prove is among the assumptions, I can connect them and I am done. *Draws a line from a rectangle on the left to a rectangle on the right.*

But more likely, the proof needs to perform a step or two. Say, this block says "if it rains, then the street is wet", which we use as an axiom *(Draws rain, arrow, water)* and this block says "it rains", which shall be an assumption in our proof *(Draws rain).* If I now want to proof that the street is wet *(Draws water on the right)* I need to perform some proof rule.

Now the rule that we need to use here is "modus ponens", also known as the elimination rule for implication. What is my mental picture of that? Well, it is a box with two connections, one for the implication and one for the premise, and another one, in the other direction, for the conclusion. I can now wire these up with my assumptions and my goal, and the proof is done.

If we look at this picture, it reminds us a bit of a flow chart, with a flow going strictly from left to right. And what flows through these lines? Truth! Or at least assumed truth, since there are assumptions on the left. Under this analogy, what is a box like this? We can think of it as a little factory or machine that produces new truths (in this case, that the street is wet), and this machine needs ingredients to build this truth (in this case, the implication and that it rains).

And this brings us to my variations on Wadler's famous slogan:

Propositions as conveyor belts,
Proof rules as machines,
Proofs as factories.

And hence the name of my program; **the Incredible Proof Machine**.

4. THE MACHINE

In such an approach, what do I have to do to perform a given proof task? I really only have to place proof blocks and wire them up. I do not have to enter any syntax. With this core idea in mind, I set out to develop the Proof Machine, and it is about time I show it to you.

As you would expect in the 2010s, it is a web application, and you just have to open incredible.pm in your browser to start using it. It greets you with a question that might remind you of the mid-90ies and offers you a choice of proof tasks to work on. These are grouped by sessions, where every session introduces a new concept.

Let us start directly with the first task in session 2; this corresponds to our example on the blackboard, only that we are now in a more abstract world, where instead of the proposition "it rains" we have the more abstract "B".

On the left, I see the blocks of my logic; there would be more we had used a later session. Among the block is one that has a striking resemblance with what I have drawn on the black board, so I can drag that onto the canvas. Just like the blocks for the assumptions and the conclusion of the task, this block has little grey connectors; the pointy ones are outgoing ports and the Pacman-like ones are incoming ports. I can wire them up again by dragging from one port to another, and if I do that three times, the proof is complete. My reward is that the conclusion turns green and I am tempted to tackle another task.

Let us pick a task where we need a very interesting block, such as this one *(picks the task that proves $A \to B$, $B \to C$ implies $A \to C$).* Here, we have to prove an implication. What block do we have for that? The one with the implication on the right, which we obviously connect to the goal. So what is the deal with the strange indent on the top of the block? Let us check what these connectors do. The one on the right requires to produce a proof of C. Tricky, but luckily the port on the left is an *outgoing* port which provides us with an A, and together with the two assumptions, we can complete the proof. I can expand the size of this block to arrange things nicely.

5. VALID PROOFS

So the system believes that this is valid proof. On what grounds? Because the proof has none of four possible issues.

1. The first one is quite simple: If any block has one of its inputs without a connection, then the proof is incomplete.

2. Then, if any connection is between ports where the proposition cannot be unified, the connection becomes red, we see a lovely skull, and the proof is not valid.

3. Obviously, circular reasoning is not allowed either. In fact, every valid proof can be rearranged so that all connections go strictly from left to right.

4. And the forth possible issue is a wrongly used local hypothesis. The A provided by the block down there must only be used to prove the C; if there is a path that bypasses this, then the proof is invalid. This effectively ensures that a valid proof is nicely nested, but does not enforce it syntactically.

It is a deliberate decision that the system allows for such invalid connections to be made, so that the student can explore what is possible and learn from it.

6. REMAINING USER INTERFACE

So while we are at it, let me give you a tour through the interface of the machine, and its features.

In the top right corner, there are buttons for undo and redo, to zoom in and out, to save the current proof as an SVG file and a button to reset the proof.

On the left, we have the current task, given in inference rule notation. Nothing exciting here.

Below, there are two bars: The upper one shows how many proof blocks your proof uses, and the lower how many blocks the smallest known solution has. This is an extra incentive for those who are bored by just solving the task to think harder about the structure of the proof. In in this example, it is actually hard to come up with a sub-optimal solution, but if I just add a pointless block onto the canvas, you see that the bar is no longer rewardingly green.

This brings me to an interesting feature of these proof graphs: They allow for completely non-linear editing. I can work from the beginning of the task, or from the back. I can edit any part of the proof, and I can leave unconnected parts around. These parts can even be broken, as in this example, but as long as they are not used to prove the goals, it does not matter.

Contrast this with conventional text-based theorem provers with linear focus, where editing something at the top is at least annoying, or other educational tree based provers out there, where you usually have to throw away a subtree if you want to change some inner node. By not getting in the way with such restrictions, unnecessary frustration is avoided.

But back to the user interface. We have already used the proof blocks. Below is a helper block. This corresponds to a cut rule, or Isabelle's **have** or, under Curry-Howard, a type signature. Once placed on the canvas, you can click on it and enter a proposition, but it is rarely used. It comes in handy if unification is stuck due to higher order terms, or in order to structure larger proofs.

And then we have custom blocks. Here, the user can abstract over partial proofs and create his own blocks, so this corresponds to defining a lemma in other systems. Lets jump to the task where we have to prove the principle of proof by contradiction. Doing that requires three blocks: Tertium non datur, disjunction elimination and ex falso.

Now in order to not having to place these three blocks every time we want to do a proof by contradiction, we can select these blocks using shift-click. The system now creates a new block based on the selected ones, and with one button we can add it to our own blocks. The name of the block is some system-assigned Unicode character – this is sufficient for recognizability and does not require the user to reach for the keyboard.

We can now use this block like any other one, for example to simplify the proof graph at hand.

7. PREDICATE LOGIC

At this point, I believe it is pretty clear to this audience how the Incredible Proof Machine is used to do simple proofs in propositional logic. But it gets more interesting if one tries to do predicate logic.

So let us look at how that is done *(Opens the task that shows that* $(\exists x.P(x)) \to A$ *implies* $\forall x.(P(x) \implies A)$*.)* The block for the universal quantifier introduction requires me to show the predicate for some constant c. If I drag this onto the canvas, this c gets a unique subscripted index; different numbers mean different constants. The trick is now that this constant may only be used in the part of the proof that goes to the input port of the universal quantifier introduction block. If there is also a path around it, then the system will not allow the free variables on the left to unify with the predicate mentioning c_3. This effectively enforces the usual freshness condition of the natural deduction rule for the universal quantifier.

Similarly the elimination rule for the existential quantifier: In order to show something (here Q) using an existentially quantified proposition, I have to show it assuming the proposition holds for some fresh constant and again this fresh constant is only accessible in this local proof.

To show that this restriction is necessary, let us create a proof task that would then be provable, namely that in every reflexive relation there is an element that related to every other element. We can, in the overview, create our own tasks to experiment with. *Creates the task that assumes* $\forall x.P(x,x)$ *and proves* $\exists x.\forall y.P(x,y)$. This proof is rejected, but why? Because it would have to unify y_5 with c_4, and hence also y_3. If it did that, the proof would be accepted. But since y_3 occurs outside the scope of c_4, it does not do that.

8. DEFINING YOUR OWN TASKS

So let us turn now to the perspective of someone who wants to use the Incredible Proof Machine for their own class. You would go to the GitHub page to fetch the source code and read the installation instructions. I will not go into detail here now.

Let us say we want to create an additional task. The tasks are defined in the file `sessions.yaml`. Generally, the Incredible Proof Machine is configured using YAML files, which is a simple indentation sensitive format for structured data. This files defines the sessions with some meta-data and a list of tasks. The meta-data includes the name of the session, which logic should be used, and which rules should be visible in this particular session.

A task is simply defined by a list of assumptions and a list of conclusions, and we can optionally store the size of the smallest known proof here. This number is not checked systematically.

So let us add the proof by contradiction in a variant that uses a negation symbol. After running `make` and opening `index.html` in my local folder, I see the newly added task. But of course, I cannot prove it yet, because there are no proof rules for negation.

In order to add these, I have to edit the files in the `logics` subdirectory. These describe the proof rules, again in an YAML file. Let us add the introduction and elimination rules for negation.

A proof rule has a name, such as `notI`. This name can also be used as the description, as in the case of *terium non datur*, but most rules have a somewhat fancy description, which a helpful symbol aligned to the left or to the right.

It has ports. The introduction rule for negation has three ports: The conclusion is $\neg P$. The assumption is a contradiction, i.e. false ($\bot$). And finally, the local hypothesis that we can use to prove this assumption is that P holds. Here we need to specify for which input this local hypothesis may be used.

Finally, we have to say that the P mentioned in these propositions is free and may unify with any proposition.

We also need to add this rule to the list of visible rules in our sessions.

Again, we run `make` to turn the YAML files into JSON files that the browser can read, and reload the page.

And there, we have our nice shiny negation introduction block. Note how the Incredible Proof Machine inferred this particular shape for us, just from the high-level description of its ports.

With this, we can complete the proof.

As you can see, it is rather simple to extend or likewise completely change both the presented tasks and the used logic.

9. DEPLOYMENT

So now you have the tasks and sessions adjusted to the need of your course, what do you need on your webserver to

make it fly? Nothing! The whole Incredible Proof Machine lies as static HTML and JavaScript files on the server, and runs completely in the browser, so you have no special requirements there. Moreover, and that saved my workshop where the WiFi could not cope with every student having their own laptop: once the machine is loaded in the browser, it can be used without an Internet connection.

So that is all there is to adjusting the Proof Machine. I should say that the set of operator and quantifier symbols is currently hard-coded, but if this is a limitation to you, let me know.

10. IMPLEMENTATION

Which brings us right to the next topic: The implementation. I already told you that the Proof Machine runs completely in the browser, so obviously there is JavaScript involved. But I am not masochistic enough to implement a theorem prover in that language.

Instead, the logical core is implemented in Haskell, and then compiled down to JavaScript using the rather new GHCJS compiler. What do I mean by logical core? Everything related to parsing propositions, to unification, to checking the well-shapedness of the proof graph. This is best explained by looking at the type signature of the exported function, and I hope this audience is indeed interested in such practical details.

The main entry point is the function `incredibleLogic`, which takes a context and a proof, and returns either an error or an analysis.

Throughout the signature, I regularly need to name things by strings. To keep these strings statically apart, I use phantom types to let the type checker help me here, this is what the `Key` type is about.

For the type of propositions, I define a simple lambda calculus. It has n-ary application. It distinguishes between variables and constants, where a symbol like $\wedge$ would be a constant, and it uses the `unbound` library by Stephanie Weirich and Brent Yorgey to handle all the details of alpha-equivalence and substitution.

The context is simply a bunch of rules. This is more or less the information present in the YAML file that we have edited.

A proof then consists of blocks and connections. Blocks are either assumptions, conclusions, helper blocks, or rule blocks. In the first three cases they carry their own proposition, while in the latter, they reference a rule of the context. Every block has a unique number, this is used for example to rename the local variables. A connection then simply connects a port of one block with another port of some block, both given by their names. The connections have a sort key so that the unification algorithm processes them from oldest to youngest; this way unification errors are more likely to show up at the just edited connection.

And finally, what does the analysis return? Most importantly, the `qed` field tells the UI whether the proof is a proper proof, and the field `portLabels` gives the proposition at each port. The other fields report the four error conditions that I talked about: At every edge, the unification might have failed or given up, there might be unconnected incoming ports, there might by cycles or there might be wrongly used local hypotheses.

What is interesting inside the implementation? Probably not much, mostly straight-forward code, for example some not every efficient graph algorithms to check the various shape constraints.

For the unification, I was very happy to have found this paper on higher order unification by Tobias Nipkow [3], where I just had to translate the ML code to Haskell code. My main modification is that even if some of the input equations fail to unify, I do not completely abort, but continue with the others. This way, mistakes in one part of the proof do not necessarily prevent the machine from properly processing other parts.

The rest of the machine, that is the user interface, the keeping tabs on the tasks and so on, is implemented as plain JavaScript. I would have loved to implement that in Haskell as well, possibly using Functional Reactive Programming, but this way I could use a ready-made graph editing library called `JointJS`, which I really did not want to reimplement.

11. THEORY

Finally, let me touch upon the theoretical aspects of the Incredible Proof Machine. I kept going on talking about proof graphs, and valid proofs, but what does that actually mean? How can I know that the Proof Machine only accepts valid proofs, and how that for everything that we want to be provable there is a proof graph?

Now, the actual rules are not really part of the Proof Machine, which is, in this sense, a metalogical framework – after all, that is why I can speak here. In that light, I reformulate the question: compared to a more conventional proof representation, does the proof machine allow me to prove more or less. The conventional proof representation here is going to be natural deduction proof trees.

So first I have to find a correspondence between my proof blocks, and derivation rules in natural deduction. Let me demonstrate that using the existential elimination block, because it uses all the features. Let me quickly draw that block again. The block shape currently does not indicate where local variables are scoped, so let me come up with an annotation for that as well.

The corresponding natural deduction rule would be this one. *Draws the inference rule.* Note that I am writing this in the style without explicit contexts, and I am using rectangles to indicate the scope of variables. Unfortunately, there is a variety of ways of presenting natural deduction inference rules.

The theoretical result that I can offer now is that for every valid proof graph there is a corresponding natural deduction tree, and the other way around. I have formalised this proof in Isabelle, you can find it in the Archive of Formal Proofs [2].

Proving this is one of those things where intuitively, it is quite clear what to do and why it is true. I just start from the conclusion backwards and build the tree... or I just start from the tree, and turn it into a graph. In practice, all kind of technical issues need to be taken care of, especially as graphs are not a very nice structure. Anyways, to at least bring in some insight from the proof: The requirement that proof graphs are cyclic corresponds to the fact that derivation trees are finite. The scopes, which I use to check whether a local hypothesis or local constant is used correctly, correspond to subtrees.

One main challenge in the proof was that the conditions around local variables is a global one in the proof graphs: A local variable must not occur anywhere in the graph outside

its scope. In the natural deduction formulation, the freshness is a local check, involving only the context and instantiation of the rule, and such a local constant might be used somewhere else again. Going from global to local freshness is not an issue, but the other direction required some careful renaming.

I should note that in my proof I assume the existence of substitutions that unify the propositions along each connection, so the unification algorithm is not included in the formal proof.

12. WHAT NEXT?

This concludes my tour with and about the current state of the Incredible Proof Machine.

Of course I have many plans for the machine. The most relevant is probably that I want the machine to be able to present the user's proof in different forms, such as proof trees or natural language proofs. Using highlighting on mouse hover, the connection between the various forms could be made visible and help the student to understand the various proof formats easily.

By the way, the Proof Machine is open source and contributions are very welcome.

So as a final remark, I'd like to say that although the Proof Machine is never going to be a full fledged theorem prover, I do believe that it has something to contribute to the world of serious theorem provers. The design space of non-linear, non-textual interactions with interactive theorem provers is not fully explored yet. Why do I have to give names to intermediate facts in an Isabelle proof, instead of just pointing at them? Why do I have to order my lemmas, which gives little insight into what uses what? The document model of modern Isabelle UIs is already a big step forward, and I am very curious what surprises the next decade will bring here. I hope that my work will inspire these surprises.

Thank you for your attention.

13. REFERENCES

[1] J. Breitner. Visual theorem proving with the incredible proof machine. In *International Conference on Interactive Theorem Proving (ITP)*, volume 9807 of *LNCS*. Springer, 2016. to appear.

[2] J. Breitner and D. Lohner. The meta theory of the incredible proof machine. *Archive of Formal Proofs*, May 2016.

[3] T. Nipkow. Functional unification of higher-order patterns. In *LICS*, 1993.