Visual theorem proving with the Incredible Proof Machine

Joachim Breitner

Karlsruhe Institute of Technology,
breitner@kit.edu

Abstract. The Incredible Proof Machine is an easy and fun to use program to conduct formal proofs. It employs a novel, intuitive proof representation based on port graphs, which is akin to, but even more natural than, natural deduction. In particular, we describe a way to determine the scope of local assumptions and variables implicitly. Our practical classroom experience backs these claims.

1 Introduction

How can we introduce high-school students to the wonderful world of formal logic and theorem proving?

Manual proofs on paper are tedious and not very rewarding: The students have to learn the syntax first, and whatever they produced, they would have to show it to a teacher or tutor before they knew if it was right or wrong.

Interactive theorem provers can amend some of these problems: These computer programs give immediate feedback about whether a proof is faulty or correct, allow free exploration and can be somewhat addictive – they have been called "the world's geekiest computer game" for a reason. Nevertheless, the students still have to learn the syntax first, and beginners without any background in either logic or programming, initially face a motivationally barren phase.

Therefore we built an interactive theorem prover that allows the students to start conducting proofs immediately and without learning syntax first. With *The Incredible Proof Machine* (http://incredible.pm/) the student just drags blocks – which represent assumptions, proof rules and conclusions – onto a canvas and wires them up, using only the mouse or a touch interface. A unification-based algorithm infers the propositions to label the connections with. Once everything is connected properly, such a graph constitutes a rigorous, formal proof.

If one thinks of assumptions as sources of propositions, conclusions as consumers of propositions, and proof rules as little machines that transform propositions to other propositions, then the connections become conveyor belts that transport truth. This not only justifies the name of the software, but is – in our opinion – a very natural representation of how the human mind approaches proving.

Another way of thinking about the Incredible Proof Machine is that it is the result of taking a graphical programming language (e.g. LabView's G [8]) and mangling it through the Curry–Howard correspondence.

The contributions of this paper are:

- We introduce a visual and natural representation of proofs as graph, which is generic in the set of proof rules. In contrast to previous approaches, it supports locally scoped variables and hence predicate logics.
- We infer the scope of local assumptions and variables implicitly from the graph structure, using post-dominators, instead of expecting an explicit declaration. This is a novel way to implement the usual freshness side-conditions.
- We give a formal description of such graphs, define when such a graph constitutes a valid formal proof, and sketch its relation to conventional natural deduction.
- The Incredible Proof Machine provides an intuitive and beginner-friendly way to learn about logic and theorem proving. We describe its interface design and its implementation.
- We report on our practical experience with the tool, including the results of a standard usability questionnaire.

2 Proof graphs

We begin with a user-level introduction to graphical proofs, as they are used in the Incredible Proof Machine. We put the focus on motivating and explaining the elements of such a proof and giving an intuition about them and defer a rigorous treatment to the subsequent section.

2.1 Conclusion and assumption

What is the intuitive essence of a proof? We assume certain propositions to be true. From these assumption, we conclude that further propositions are true, using the rules of the logic at hand. Eventually we construct a proposition that matches what we want to prove, i.e. the conclusion. In the simplest case, the conclusion is among the assumptions, and the proof is trivial.

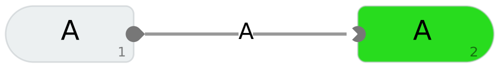


Fig. 1. A very trivial proof

If we depict such a proof, the picture in Fig. 1 might come up: A *block* representing the assumption, a second block representing the conclusion, and a line between them to draw the *connection*. Both blocks are labelled with the proposition they provide resp. expect, namely A, and the line is also labelled with the proposition. This is a valid proof, and the conclusion turns green.

It is worth pointing out that in these proof graphs, the train of thought runs from left to right. Hence, assumptions have *ports* (the little grey circles where connections can be attached to) on their right, and conclusions on their left. Such outgoing and incoming ports also have different shapes. The system does not allow connections between two outgoing or two incoming ports.

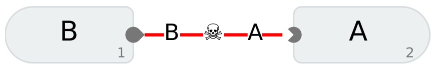


Fig. 2. A very wrong proof

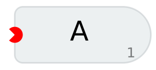


Fig. 3. A very incomplete proof

A wrong proof is shown in Fig. 2, where the proposition of the assumption (B) differs from the proposition of the conclusion (A). Thus, these blocks cannot legally be connected, the false connection is red, and a scary symbol explains the problem. Needless to say, the conclusion is not green.

Similarly, the conclusion in Fig. 3 is not green, as the proof is incomplete. This is indicated by a red port. In general, anything red indicates that the proof is either incomplete or wrong.

2.2 Rule blocks

To conduct more than just trivial proofs, we need more blocks. These correspond to the inference rule of the underlying logic. Figure 4 shows some typical proof blocks and the corresponding natural deduction rule(s) in conventional inference rule format (antecedents above, consequent below the line).

$$\frac{X \qquad Y}{X \wedge Y} \qquad\qquad \frac{X \wedge Y}{X} \quad \frac{X \wedge Y}{Y} \qquad\qquad \frac{X \to Y \qquad X}{Y}$$

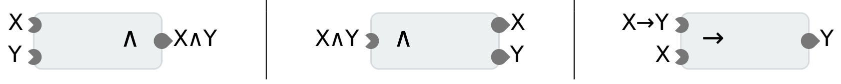


Fig. 4. Some natural deduction rules and their proof block counterparts

Again, incoming ports (on the left) indicate prerequisites of a rule, while outgoing ports (on the right) correspond to the conclusions of a rule. In contrast to usual inference rules, rule blocks can have multiple conclusions, so both conjunction projection rules are represented by just one block. If only one of the conclusions is needed, the other outgoing ports would simply be left unconnected. Unlike unconnected prerequisites, this does in no way invalidate a proof.

The graph in Fig. 5 shows a proof that from $A \wedge B$ and $A \to B \to C$ we can conclude C. Note the connection labels, which indicate the proposition that is “transported” by a connection.

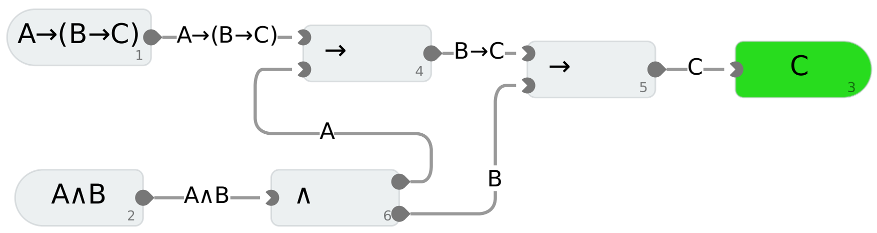


Fig. 5. A more complex proof

2.3 Local hypotheses

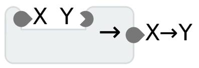


Fig. 6. Implication introduction

Figure 4 shows both the introduction and elimination rule for conjunction, as well as the elimination form for implication (*modus ponens*) – clearly we are missing a block that allows us to introduce the implication. Such a block would produce output labelled $A \to B$ if given an input labelled B, where the proof for this B may make use of A. But that local hypothesis A must not be used elsewhere! This restriction is hinted at by the shape of the implication introduction block in Fig. 6, where the dent in the top edge of the block suggests that this block will encompass a subproof. To support this, the block – colloquially called a "sliding calliper" – can be horizontally expanded as needed.

The graph in Fig. 7 shows the simplest proof using the calliper: By connecting the port of the local hypothesis A with the assumption A, we obtain a valid proof of $A \to A$.

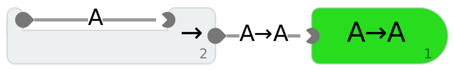


Fig. 7. Implication done right

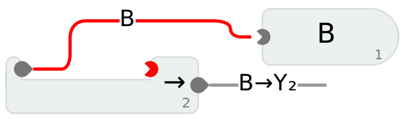


Fig. 8. Implication done wrong

The graph next to it (Fig. 8) shows an invalid use of the implication introduction block: The hypothesis is not used locally to prove the assumption of the block, but is instead connected directly to the conclusion of the proof. The Incredible Proof Machine allows the user to make that connection, but complains about it by colouring it in red.

In this picture you can see that despite the proof being in an invalid state, the system determined that the implication produced by this block would have B as the assumption, and a not yet determined proposition Y_2 as the conclusion. The ability to work with partial and even wrong proofs is an important ingredient to a non-frustrating user experience.

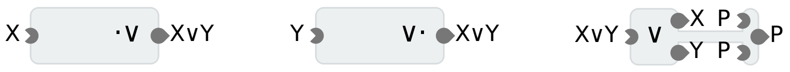


Fig. 9. Disjunction introduction and elimination rules

A block can have more than one local hypothesis, with different scoping rules. An example for that is the elimination block for disjunction, shown in Fig. 9. In this case, the conclusion of the block (P) is the same as the local goal on each side of the block. This seems to be a bit redundant, but is necessary to delimit the scope of the two local hypotheses, respectively, and to keep the two apart – after all, using the local hypothesis from one side in the proof of the other side leads to unsoundness (Fig. 10).

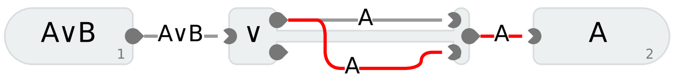


Fig. 10. A local hypotheses of the disjunction block used wrongly.

2.4 Predicate logic

So far, the user can only conduct proofs in propositional logic, which is a good start for beginners, but gets dull eventually. Therefore the Incredible Proof Machine also supports predicate logic. This opens a whole new can of worms, as the system has to keep track of the scope of local, fixed variables.

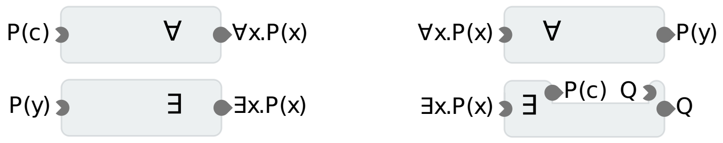


Fig. 11. Blocks for quantifiers

The additional rules are shown in Fig. 11. The introduction rule for the existential quantifier (bottom left) and the elimination rule for the universal quantifier (top right) are straight forward: If one can prove $P(y)$ for some term y, then $\exists x.P(x)$ holds, and conversely if one has $\forall x.P(x)$, then $P(y)$ holds for some term y.

At the first glance, it seems strange that the introduction rule for the universal quantifier (top left) has the same shape as the one for the existential quantifier. But there is a small difference, visible only from the naming convention: To obtain $\forall x.P(x)$ the user has to prove $P(c)$ for an (arbitrary but fixed) *constant* c.

Furthermore, and not visible from the shape of the block, is that this constant c is available only locally, in the proof of $P(c)$. To enforce this, the Incredible Proof Machine identifies those proof blocks from where all paths pass through the universal quantifier introduction block on their way to a conclusion, and only the free variables of these blocks are allowed to be instantiated by a term that mentions c. This restriction implements the usual freshness side condition in an inference rule with explicit contexts:

$$\frac{\Gamma \vdash P(c) \qquad c \text{ does not occur in } \Gamma}{\Gamma \vdash \forall x.P(x)}$$

Such a local constant is also used in the elimination rule for the existential quantifier (bottom right), where in order to prove a proposition Q, we may use that $P(c)$ holds for some constant c, but this constant may only occur in this part of the proof, and moreover the proposition $P(c)$ is a local hypothesis (Section 2.3) and may not escape this scope.

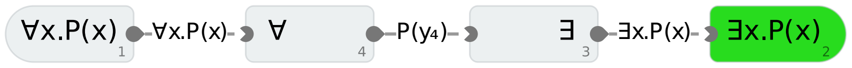


Fig. 12. A proof that $\forall x.P(x)$ entails $\exists x.P(x)$.

In this formulation of predicate logic, the universe is unspecified, but not empty. In particular, it is valid to derive $\exists x.P(x)$ from $\forall x.P(x)$ (Fig. 12).

The asymmetry in Fig. 11 is striking, and the question arises why the elimination block for the existential quantifier would not just produce $P(c)$ as its output, forming a proper dual to the universal quantifier introduction block. This could work, but it would require the Incredible Proof Machine to intelligently determine a scope for c; in particular it had to ensure that scopes nest properly. With some scopes extending backwards (universal quantifier introduction) and some forwards (existential quantifier elimination), automatically inferring sensible and predictable scoping becomes tricky, so we chose to use a block shape that makes the scope explicit. More on scopes in Section 3.2.

2.5 Helper block

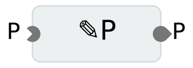


Fig. 13. The helper block

With full-scale theorem provers such as Isabelle or Coq it is quite helpful to break down a proof into smaller steps and explicitly state intermediate results. The same holds for the Incredible Proof Machine, and is made possible using the so-called helper block, shown in Fig. 13. Once placed in the proof area, the user can click on it and enter a proposition, which is then both assumed and produced by this block. Logically, this corresponds to a use of the cut rule.

The block is also useful if the desired proposition is not inferred, which can be the case with partial proofs, especially if quantifiers are involved.

2.6 Custom blocks

After performing a few proofs with the Incredible Proof Machine, the user soon notices that some patterns appear repeatedly. One such pattern would be a proof by contradiction, which consists of the three blocks highlighted in Fig. 14: Tertium non datur, disjunction elimination and ex falso quodlibet. (Note that the negation of X is expressed as $X \to \bot$.)

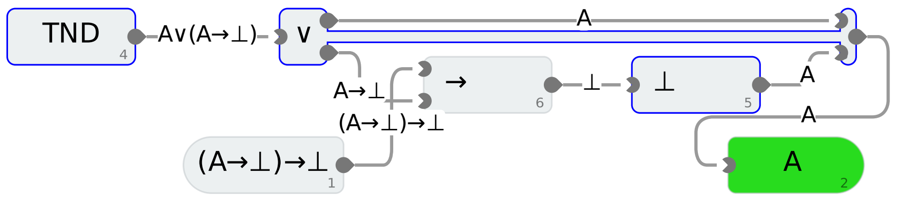


Fig. 14. A primitive proof of double negation elimination

When the user has selected a part of the proof this way (by shift-clicking), he can create a custom block that represents the selected proof fragment. In this case, the custom block would look as in Fig. 15, and with that block, which now directly represents a proof by contradiction, the whole proof is greatly simplified (Fig. 16). This mechanism corresponds to the lemma command in, say, Isabelle.

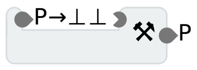


Fig. 15. A custom block

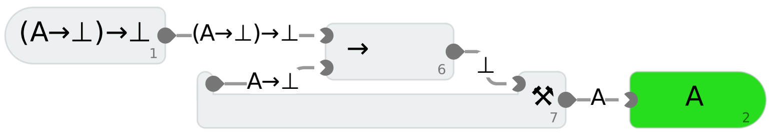


Fig. 16. A shorter proof of double negation elimination

2.7 Custom logics

The rule blocks, and hence the underlying logic, are not baked into the Incredible Proof Machine, but read from a simple text file. Figure 17 shows the declaration of the first disjunction introduction block, the implication introduction block and the universal quantifier introduction block.

```
rules:
- id: disjI1
  desc:
    intro: ·∨
  free: ["X","Y"]
  ports:
    in:
      type: assumption
      proposition: X
    out:
      type: conclusion
      proposition: X∨Y
- id: impI
  desc:
    intro: →
  free: ["X","Y"]
  ports:
    hyp:
      type: local hypothesis
      consumedBy: in
      proposition: X
    in:
      type: assumption
      proposition: "Y"
    out:
      type: conclusion
      proposition: X→Y
- id: allI
  desc:
    intro: ∀
  free: ["P"]
  local: ["c"]
  ports:
    in1:
      type: assumption
      proposition: P(c)
      scoped: ["c"]
    out2:
      type: conclusion
      proposition: ∀x.P(x)
```

Fig. 17. Extract of `predicate.yaml`, where the rule blocks are defined

Each rule needs to have an identifier (`id`), but may specify a more readable description (`desc`), which includes a hint towards what side of the block the description should be aligned. The next two fields specify which variables in the following propositions are `free`, i.e. may be instantiated upon connecting the blocks, and which are `local`, i.e. different for each instance of the block.

Then the list of `ports` is indexed by an arbitrary identifier (`in`, `out`, …). A port has a `type`, which is `assumption`, `conclusion` or `local hypothesis`. In the latter case, this port may only be used towards proving the port specified in the `consumedBy` field. Every port specifies the `proposition` that is produced resp. expected by this block. A local constant (such as the `c` in rule `allI`) is usually `scoped` by a port of type `assumption` (see Sections 2.4 and 3.2).

It is simple to experiment with completely different logics, without changing the code. For example, we have implemented a Hilbert-style system for propositional logic (one rule block for modus ponens and three rules blocks for the axioms) and the typing derivations of the simply typed lambda calculus.

A similar file specifies the pre-defined tasks, which can be grouped into sessions, and each session can use a different logic, or – for an educational progression between the sessions – can show only a subset of a logic's rules.

Naturally, none of these files are user-visible. They would, however, provide the mechanism by which an educator who wants to use the Incredible Proof Machine in his course, simply by editing the rules and tasks therein as desired.

3 Theory

The previous section has (intentionally) only scratched the surface of the Incredible Proof Machine, and avoided some more fundamental questions such as: What precisely makes up a proof graph (and what is just cosmetic frill)? When is it valid? And what does it actually prove?

These questions are answered in this section with some level of formalism. In particularly, we define when the shape of a proof graph is valid (e.g. no cycles, local hypotheses wired up correctly) and when scoped variables are used correctly. While we use the language of *port graphs* as introduced in [2], the notion of a well-shaped graph is a new contribution.

3.1 Port graphs

In contrast to those in [2], our port graphs are *directed.*

Definition 1 (Port graph signature). *A (directed)* port graph signature ∇ *over a set $\mathcal{N}$ of node names and a set $\mathcal{P}$ of port names consists of the two functions ${}_{\rightarrow}\nabla : \mathcal{N} \to 2^{\mathcal{P}}$ and $\nabla_{\rightarrow} : \mathcal{N} \to 2^{\mathcal{P}}$, which associate to a node name the names of its incoming resp. outgoing ports.*

Definition 2 (Port graph). *A (directed)* port graph G *over a signature ∇ is a tuple (V, n, E) consisting of*

- *a set V of vertices,*
- *a function $n : V \to \mathcal{N}$ to associate a node to each vertex and*

– *a multiset $E \subseteq (V \times \mathcal{P}) \times (V \times \mathcal{P})$ of edges such that for every edge (v_1, p_1)—$(v_2, p_2) \in E$ we have $p_1 \in \nabla_{\rightarrow}(n(v_1))$ and $p_2 \in {}_{\rightarrow}\nabla(n(v_2))$.*

The notation s—t used for an edge is just syntax for the tuple (s, t).

We need a number of graph-theoretic definitions.

Definition 3 (Path). *A* path *in G is a sequence of edges (v_1, p'_1)—(v_2, p_2), (v_2, p'_2)—(v_3, p_3), $\ldots$, (v_{n-1}, p'_{n-1})—$(v_n, p_n) \in E$. The path* begins *in (v_1, p'_1) (or just v_1) and* ends *in (v_n, p_n) (or just v_n).*

Definition 4 (Terminal node, pruned graph). *A node $n \in \mathcal{N}$ is called a* terminal node*, if $\nabla_{\rightarrow}(n) = \{\}$, and a vertex $v \in V$ is called a terminal vertex if $n(v)$ is a terminal node. A graph is called* pruned *if every vertex v is either a terminal vertex, or there is a path from v to a terminal vertex.*

3.2 Scopes

The key idea to support both local assumptions and scoped variables is that the scope of a local proof can be implicitly inferred from the shape of the graph.

It is desirable to have as large as possible scopes, so that as many proof graphs as possible are well-scoped. On the other hand, the scopes must be small enough to still be valid. This motivates

Definition 5 (Scope). *In a port graph $G = (V, n, E)$, the* scope *of an incoming port (v, p) with $p \in {}_{\rightarrow}\nabla(n(v))$ is the set $S(v, p) \subseteq V$ of vertices post-dominated by the port. More precisely: $v' \in S(v, p)$ iff v' is not a terminal vertex and every path that begins in v' and ends in a terminal vertex passes through (v, p).*

As an indication that this is a sensible definition, we show that the scopes nest the way one would expect them to. We restrict this to pruned graphs; pruning a graph removes only unused and hence irrelevant parts of the proof.

Lemma 1 (Scopes nest). *Let $G = (V, n, E)$ be a pruned graph. For any two (v_1, p_1) and (v_2, p_2) with $v_i \in V$ and $p_i \in {}_{\rightarrow}\nabla(n(v_i))$ $(i = 1, 2)$, we have $S(v_1, p_1) \subseteq S(v_2, p_2)$ or $S(v_2, p_2) \subseteq S(v_1, p_1)$ or $S(v_1, p_1) \cap S(v_2, p_2) = \{\}$.*

Proof. We show that $S(v_1, p_1) \cap S(v_2, p_2) \neq \{\}$ implies $S(v_1, p_1) \subseteq S(v_2, p_2)$ or $S(v_2, p_2) \subseteq S(v_1, p_1)$.

Let $v \in S(v_1, p_1) \cap S(v_2, p_2)$. The vertex v is not terminal, so there is a path from v to a terminal node, and it necessarily passes through (v_1, p_1) and (v_2, p_2). W.l.o.g. assume (v_1, p_1) occurs before (v_2, p_2) on that path. Then all paths from (v_1, p_1) to a terminal node go through (v_2, p_2), as otherwise we could construct a path from v to a terminal node that does not go through (v_2, p_2).

Now consider a $v' \in S(v_1, p_1)$. All paths to a terminal node go through (v_1, p_1), and hence also through (v_2, p_2), and we obtain $S(v_1, p_1) \subseteq S(v_2, p_2)$.

3.3 Graph shapes

The above definition of scopes allows us to say how a local hypothesis needs to be wired up. We also need a relaxed definition of acyclicity.

Definition 6 (Local hypothesis). *A* local hypothesis specification *for a graph signature ∇ is a partial function $h_n : \mathcal{P} \rightharpoonup \mathcal{P}$ for every $n \in \mathcal{N}$ such that $h_n(p) = p'$ implies $p \in \nabla_{\!\rightarrow}(n)$ and $p' \in {}_{\rightarrow}\!\nabla(n)$. In that case, p is a local hypothesis of n and p' defines its scope.*

Definition 7 (Well-scoped graph). *A port graph $G = (V, n, E)$ with a local hypothesis specification h is* well-scoped *if for every edge (v_1, p_1)—(v_2, p_2) where $h_{n(v_1)}(p_1) = p'$ we have $(v_2, p_2) = (v_1, p')$ or $v_2 \in S(v_1, p')$.*

Definition 8 (Acyclic graph). *A port graph $G = (V, n, E)$ with a local hypothesis specification h is* acyclic *if there is no path connecting a node to itself, disregarding paths that pass by some local hypothesis, i.e. where there is a (v, p) on the path with $p \in \operatorname{dom} h_{n(v)}$.*

Definition 9 (Saturated graph). *A port graph $G = (V, n, E)$ is* saturated *if every (v, p) with $p \in {}_{\rightarrow}\!\nabla(n(v))$ is incident to an edge.*

To summarise when a graph is in a good shape to form a proof, we give

Definition 10 (Well-shaped graph). *A port graph G is* well-shaped *if it is well-scoped, acyclic and saturated.*

3.4 Propositions

So far we have described the shape of the graphs; it is about time to give them meaning in terms of logical formulas. We start with propositional logic (no binders and no scoped variables) first.

Definition 11 (Formulas). *Let $\mathcal{X}$ be a set of variables, and $\mathcal{F}_X$ a set of formulas with variables in X.*

Definition 12 (Labelled signature). *A port graph signature ∇ is labelled by formulas $l : \mathcal{N} \times \mathcal{P} \to \mathcal{F}_{\mathcal{X}}$.*

For two vertices v_1, v_2 with the same node name, the free variables of the formulas need to be distinct. So in the context of a specific graph, we annotate the variables with the vertex they originate from:

$$l' : V \times \mathcal{P} \to \mathcal{F}_{\mathcal{X} \times V}$$
$$l'(v, p) = l(n(v), p)[x_v/x \mid x \in \mathcal{X}]$$

We use the subscript syntax x_v to denote the tuple (x, v).

The Incredible Proof Machine employs a unification algorithm to make sure the formulas expected on either side of an edge match, if possible. Here, we abstract over this and simply require a unifying substitution, which we model as a function.

Definition 13 (Instantiation). *An* instantiation *for a port graph G with a labelled signature is, for every vertex $v \in N$, a function $\theta_v : \mathcal{F}_{\mathcal{X}\times N} \to \mathcal{F}_{\mathcal{X}\times N}$.*

Definition 14 (Solution). *An instantiation θ for a port graph G with a labelled signature is a* solution *if for every edge (v_1, p_1)—$(v_2, p_2) \in E$ we have $\theta_{v_1}(l'(v_1, p_1)) = \theta_{v_2}(l'(v_2, p_2))$.*

Definition 15 (Proof graph). *A* proof graph *is a well-shaped port graph with a solution.*

3.5 Scoped variables

To support binders and scoped variables, we need to define which variables are scoped (a property of the signature), and then ensure that the scopes are adhered to (a property of the graph). For the latter we need – a bit vaguely, to stay abstract in the concrete structure of terms – the notion of the range of an instantiation, $\operatorname{ran}\theta_i \subseteq \mathcal{X} \times N$, which is the set of free variables of the formulas that the substitution substitutes for.

Definition 16 (Scoped variables). *A port graph signature ∇ can be annotated with* variable scopes *by a partial function $s_n : \mathcal{X} \rightharpoonup \mathcal{P}$, for every $n \in \mathcal{N}$, where $s_n(x) = p$ implies $p \in {}_{\rightarrow}\!\nabla(n)$.*

Definition 17 (Well-scoped instantiation). *An instantiation θ for a port graph G with a labelled signature and scoped variables is* well-scoped *if for every scoped variable x, i.e. $s_{n(v)}(x) = p$ for some vertex $v \in V$, $x \in \operatorname{ran}\theta_{v'}$ implies that $v' \in S(v, p)$.*

In the presence of scoped variables, we extend Definition 15 to

Definition 18 (Proof graph). *A* proof graph *is a well-shaped port graph with a well-scoped solution.*

3.6 Example

After this flood of definitions, let us give a complete and comprehensive example.

To prove that $\exists x.(\mathrm{P}(x) \wedge \mathrm{Q}(x))$ entails $\exists x.\mathrm{P}(x)$ (where the bold **P** indicates a constant, not a variable in the sense of $\mathcal{X}$), we would use node names $\mathcal{N} = \{\mathtt{a}, \mathtt{c}, \mathtt{exE}, \mathtt{conjE}, \mathtt{exI}\}$ and port names $\mathcal{P} = \{\mathtt{in}, \mathtt{out}, \mathtt{in2}, \mathtt{out2}\}$. The (labelled and scoped) signature is given by

$$\begin{aligned} {}_{\rightarrow}\!\nabla(\mathtt{a}) &= \{\} & \nabla_{\!\rightarrow}(\mathtt{a}) &= \{\mathtt{out}\} \\ {}_{\rightarrow}\!\nabla(\mathtt{c}) &= \{\mathtt{in}\} & \nabla_{\!\rightarrow}(\mathtt{a}) &= \{\} \\ {}_{\rightarrow}\!\nabla(\mathtt{exE}) &= \{\mathtt{in}, \mathtt{in2}\} & \nabla_{\!\rightarrow}(\mathtt{exE}) &= \{\mathtt{out}, \mathtt{out2}\} \\ {}_{\rightarrow}\!\nabla(\mathtt{conJ}) &= \{\mathtt{in}\} & \nabla_{\!\rightarrow}(\mathtt{conjE}) &= \{\mathtt{out}, \mathtt{out2}\} \\ {}_{\rightarrow}\!\nabla(\mathtt{exI}) &= \{\mathtt{in}\} & \nabla_{\!\rightarrow}(\mathtt{exI}) &= \{\mathtt{out}\} \end{aligned}$$

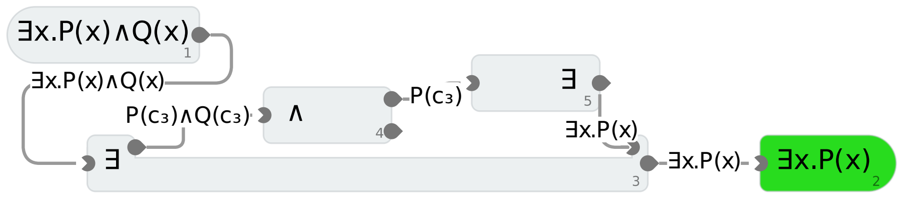


Fig. 18. A comprehensive example

$$
\begin{aligned}
l(\mathtt{a},\mathtt{out}) &= \exists x.(\mathbf{P}(x) \wedge \mathbf{Q}(x)) & l(\mathtt{c},\mathtt{in}) &= \exists x.\mathbf{P}(x) \\
l(\mathtt{exE},\mathtt{in}) &= \exists x.P(x) & l(\mathtt{exE},\mathtt{out}) &= P(c) \\
l(\mathtt{exE},\mathtt{in2}) &= Q & l(\mathtt{exE},\mathtt{out2}) &= Q \\
l(\mathtt{conjE},\mathtt{in}) &= X \wedge Y & l(\mathtt{conjE},\mathtt{out}) &= X \\
& & l(\mathtt{conjE},\mathtt{out2}) &= Y \\
l(\mathtt{exI},\mathtt{in}) &= P(y) & l(\mathtt{exI},\mathtt{out}) &= \exists x.P(x) \\
\\
h_{\mathtt{exE}}(\mathtt{out}) &= \mathtt{in2} & s_{\mathtt{exE}}(c) &= \mathtt{in2}.
\end{aligned}
$$

A well-shaped proof graph for this signature – also shown in Fig. 18 – is given by $N = \{1..5\}$, $n(1) = \mathtt{a}$, $n(2) = \mathtt{c}$, $n(3) = \mathtt{exE}$, $n(4) = \mathtt{conjE}$, $n(5) = \mathtt{exI}$ and

$$
\begin{aligned}
E = \{&(1,\mathtt{out})—(3,\mathtt{in}),\ (3,\mathtt{out})—(4,\mathtt{in}), \\
&\quad (4,\mathtt{out})—(5,\mathtt{in}),\ (5,\mathtt{out})—(3,\mathtt{in2}),\ (3,\mathtt{out2})—(2,\mathtt{in})\}.
\end{aligned}
$$

A solution for this graph is given by these higher-order substitutions:

$$
\begin{aligned}
\theta_3 &= [(\lambda x.\mathbf{P}(x) \wedge \mathbf{Q}(x))/P_3,\ \exists x.\mathbf{P}(x)/Q_3] \\
\theta_4 &= [\mathbf{P}(c_3)/X_4,\ \mathbf{Q}(c_3)/Y_4] \\
\theta_5 &= [(\lambda x.\mathbf{P}(x))/P_5,\ c_3/y_5]
\end{aligned}
$$

Note that this is well-scoped: We have $c_3 \in \operatorname{ran}\theta_i$ only for $i \in S(3,\mathtt{in2}) = \{4,5\}$.

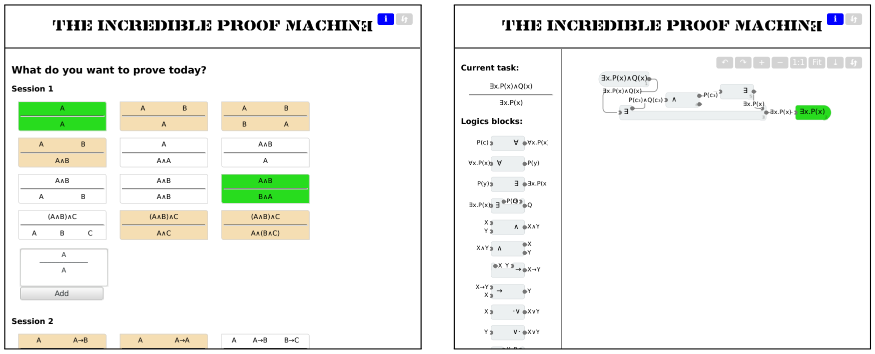
3.7 Proof Conclusions

In order to relate a proof graph with a proof in a given logic, we assume a partition of the nodes $\mathcal{N}$ into assumptions $\mathcal{N}_A$, conclusions $\mathcal{N}_C$ and rules $\mathcal{N}_R$, where $\mathcal{N}_A$ contains only assumptions (no input and precisely one output) and $\mathcal{N}_C$ only conclusions (no output and precisely one input). For a vertex $v \in V$ with $n(v) \in \mathcal{N}_A \cup \mathcal{N}_C$ let $l'(v) = l'(v,p)$ where p is the single outgoing resp. incoming port of $n(v)$.

If we assume that the nodes in $\mathcal{N}_R$ faithfully implement the inference rules of a natural deduction-style implementation of a given logic, we can state

Theorem 1 (Soundness and Completeness). *The existence of a proof graph, with a vertex for every conclusion ($\mathcal{N}_C \subseteq n(V)$), implies that from the set of formulas $\{l'(v) \mid n(v) \in \mathcal{N}_A\}$, all formulas in $\{l'(n) \mid n(v) \in \mathcal{N}_C\}$ are derivable by natural deduction, and vice versa.*

A mechanized proof of this theorem, built using the interactive theorem prover Isabelle, can be found in the Archive of Formal Proofs [6].



THE INCREDIBLE PROOF MACHINE
What do you want to prove today?
Session 1
Add
Session 2
THE INCREDIBLE PROOF MACHINE
Current task:
∃x.P(x)∧Q(x)
∃x.P(x)
Logics blocks:

Fig. 19. The Incredible Proof Machine, task selection and proving

4 Implementation

The Incredible Proof Machine is based on web technologies (HTML, JavaScript, SVG) and runs completely in the web browser. Once it is loaded, no further internet connection is required – this was useful when the workshop WiFi turned out to be unreliable. This also simplifies hosting customised versions. It adjusts to the browser's configured language, currently supporting English and German.

The logical core is implemented in Haskell, which we compile to JavaScript using GHCJS. It uses the `unbound` library [16] to handle local names, and a translation of Nipkow's higher-order pattern unification algorithm [13]. There is little in the way of a LCF-style trusted core, and the system can easily be tricked from the browser's JavaScript console.

The Incredible Proof Machine greets its users with a list of tasks to prove (Fig. 19, left). Attempted tasks are highlighted yellowishly; completed tasks in green. The main working view (Fig. 19, right) consist of a left pane, listing the current task and the various blocks which can be dragged onto the main pane. The interface supports undo/redo, zooming and can save the proof as an SVG graphic.

The system continuously checks and annotates the proof, even in the presence of errors, supporting an incremental work flow. This currently happens synchronously and it gets a little sluggish with larger proofs.

Custom blocks are also listed in the left pane, where they can be created and deleted. The overview page allows users to quickly define new tasks.

Custom blocks, new tasks and the state of all proofs are stored in the browser (by way of Web Storage), so a returning user can continue where he left.

An educator would customise the Incredible Proof Machine by adjusting the files that contain the logic definition (Section 2.7) and the tasks.

All code is liberally licensed Free Software, and contributions at http://github.com/nomeata/incredible are welcome.

5 Evaluation

The Incredible Proof Machine has been used in practice, and our experience shows that it does indeed achieve the desired goal of providing an entertaining low barrier entry to logic and formal proofs.

5.1 Classroom experience

Development of the Incredible Proof Machine was initiated when the author was given the possibility to hold a four day workshop with high school students to a topic of his choosing. The audience consisted of 13 students ages 13 to 20, all receiving a scholarship by the START-Stiftung for motivated students with migration background. Prior knowledge in logic, proofs or programming was not expected, and in most cases, not present.

Within the 14 hours spent exclusively working with the Incredible Proof Machine, the class covered the propositional content; two very apt students worked quicker and managed most of the predicate proofs as well.

After a very quick introduction to the user interface, the procedure was to let the students explore the next session, which was unlocked using a password we gave them to keep everyone on the same page, on their own. They should experiment and come up with their own mental picture of the next logical connective, before we eventually discussed it together, explained the new content, and handed out a short text for later reference.

We evaluated the user experience using a standardised usability questionnaire (UEQ, [9]). The students answered 26 multiple-choice questions, which are integrated into a score in six categories. The evaluation tool compares the score against those of 163 product evaluations. It confirms an overall positive user experience, with the Incredible Proof Machine placed in the top quartile in the categories Attractiveness, Dependability and Novelty and beating the averages in Perspicuity, Efficiency and Novelty. Additional free-form feedback also confirmed that most students enjoyed the course – some greatly – and that some found the difficulty level rather high.

We have since used the Incredible Proof Machine two further times as a 90 minute taster course addressing high-school students interested in the mathematics and computer science courses at our university.

5.2 Online reception

The Incredible Proof Machine is free to use online, and "random people from the internet" have played with it. While we cannot provide representative data on how it was perceived, roughly two hundreds posts on online fora (Twitter, Reddit, Hacker News)[1] are an encouraging indication. Users proudly posted screenshots of their solutions, called the Incredible Proof Machine "addictive" and one even reported that his 11-year old daughter wants to play with it again.

A German science podcast ran a 3-hour feature presenting the Incredible Proof Machine. [5]

6 Future Directions

We plan to continue developing the Incredible Proof Machine into a versatile and accessible tool to teach formal logic and rigorous proving. For that we want to make it easier to use, more educational and more powerful.

To improve usability, we envision a better visualisation of the inferred scopes, to ease proofs in predicate logic.

To make it more educational, we plan to add an interactive tutorial mode targeting self-learners. A simultaneous translation of the proof graph in to a (maybe bumpy) natural language proof, with a way to explore how the respective components correspond, will also greatly improve the learning experience.

A powerful, yet missing, feature is the ability to abstract not only over proofs (lemmas), but also over terms (definitions). Inspired by ML's sealing of abstract types [12], we'd make, for a given proof, a given definition (such as $\neg A := A \to \bot$) either transparent or abstract. This would encourage and teach a more disciplined approach to abstraction than if a definition could be unfolded locally whenever convenient.

Proof graphs might be worthwhile to use also in full interactive theorem provers: Consider a local **proof** with intermediate results in a typical Isar proof. It would be quite natural to free the user from having to place them into a linear order, to give names and to refer to these names when he could just draw lines! The statefulness of Isabelle code (e.g. attribute changes, simplifier setup, etc.) pose some interesting challenges in implementing this idea.

7 Related Work

Given how intuitive it appears to us to write proofs as graphs, we were surprised to find little prior work on that. Closest to our approach is [1], which identified (undirected) port graphs as defined in [2] as the right language to formulate these

[1] https://twitter.com/nomeata/status/647056837062324224, https://reddit.com/mbtk2, https://reddit.com/3m7li1, https://news.ycombinator.com/item?id=10276160, https://twitter.com/d_christiansen/status/647117704764256260, https://twitter.com/mjdominus/status/675673521255788544, https://twitter.com/IlanGodik/status/716258636566290432

ideas in, and covers intuitionistic propositional logic. We develop their approach further by deducing scopes from the graph structure and by supporting predicate logic as well, and we believe that *directed* port graphs, as used in this work, are more suitable to represent proofs.

The inner workings of the Incredible Proof Machine, as well as our implementation of predicate logic, were obviously influenced by Isabelle's [14].

We looked into existing graphical and/or educational approaches to formal logic. We particularly like *Domino on Acid* [4], which represents proofs as domino pieces. It provides a very game-like experience, but is limited to propositional proofs with $\rightarrow$ and $\perp$ only. The graphical interactive tools *Polymorphic Blocks* [10] and *Clickable Proofs* [15] support all the usual propositional connectives, but none of these, though, support predicate logic.

There is a greater variety in tools that allow mouse-based editing of more textual proof representations. Examples are *Logitext* [17], which sports a slick interface and provides a high assurance due to the Coq [7] back end, and *KeY* [3], a practical system for program verification. *Easyprove* [11] sticks out as it allows the user to click their way to proper, though clumsy, English proofs. With all these tools the user usually loses a part of his proof when he needs to change a step done earlier, while the Incredible Proof Machine allows him to edit anything at any time, and broken or partial proof fragments can stay around.

8 Conclusions

We lowered the entry barrier to formal logic and theorem proving by offering an intuitive graphical interface to conduct proofs. We have used our program in practice and found that this approach works: Young students with no prior knowledge can work with the tool, and actually enjoy the puzzle-like experience.

We therefore conclude that the non-linear, graphical proof representation, as presented in this work, has advantages over more conventional text-based approach in learning logic.

Acknowledgements

We thank Denis Lohner, Richard Molitor, Martin Mohr and Nicole Rauch for their contributions to the Incredible Proof Machine, and Andreas Lochbihler and Sebastian Ritterbusch for helpful comments on a draft of this paper. Furthermore, I thank the anonymous referees for the encouraging review and the list of feature requests.

References

1. Alves, S., Fernández, M., Mackie, I.: A new graphical calculus of proofs. In: TERMGRAPH. EPTCS, vol. 48 (2011)
2. Andrei, O., Kirchner, H.: A rewriting calculus for multigraphs with ports. ENTCS 219 (2008)

3. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of Object-oriented Software: The KeY Approach, LNCS, vol. 4334. Springer (2007)
4. Benkmann, M.: Visualization of natural deduction as a game of dominoes, http://www.winterdrache.de/freeware/domino/data/article.html
5. Breitner, J.: Incredible proof machine. Conversation with Sebastian Ritterbusch, Modellansatz Podcast, episode 78, Karlsruhe Institute of Technology (2016), http://modellansatz.de/incredible-proof-machine
6. Breitner, J.: The meta theory of the incredible proof machine. Archive of Formal Proofs (May 2016), http://isa-afp.org/entries/Incredible_Proof_Machine.shtml, Formal proof development
7. Coq development team, T.: The Coq proof assistant reference manual. LogiCal Project (2004), http://coq.inria.fr, version 8.0
8. Johnson, G.W.: LabVIEW graphical programming. McGraw-Hill (1997)
9. Laugwitz, B., Held, T., Schrepp, M.: Construction and evaluation of a user experience questionnaire. In: HCI and Usability for Education and Work, LNCS, vol. 5298. Springer (2008)
10. Lerner, S., Foster, S.R., Griswold, W.G.: Polymorphic blocks: Formalism-inspired UI for structured connectors. In: CHI. ACM (2015)
11. Materzok, M.: Easyprove: a tool for teaching precise reasoning. In: TTL. Université de Rennes 1 (2015)
12. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. TOPLAS 10(3) (Jul 1988)
13. Nipkow, T.: Functional unification of higher-order patterns. In: LICS (1993)
14. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
15. Selier, T.: A Propositionlogic-, naturaldeduction-proof app(lication). Bachelor's thesis, Utrecht University (2013)
16. Weirich, S., Yorgey, B.A., Sheard, T.: Binders unbound. In: ICFP. ACM (2011)
17. Yang, E.Z.: Logitext, http://logitext.mit.edu/