On Verified Automated Reasoning in Propositional Logic

Lund, Simon Tobias; Villadsen, Jørgen

Published in:
Intelligent Information and Database Systems.

Link to article, DOI:
10.1007/978-3-031-21743-2_31

Publication date:
2022

Document Version
Peer reviewed version

On Verified Automated Reasoning in Propositional Logic

Simon Tobias Lund and Jørgen Villadsen[0000−0003−3624−1159]

Technical University of Denmark, Kongens Lyngby, Denmark

Abstract. As the complexity of software systems is ever increasing, so is the need for practical tools for formal verification. Among these are automatic theorem provers, capable of solving various reasoning problems automatically, and proof assistants, capable of deriving more complex results when guided by a mathematician/programmer. In this paper we consider using the latter to build the former. In the proof assistant Isabelle/HOL we combine functional programming and logical program verification to build a theorem prover for propositional logic. Finally, we consider how such a prover can be used to solve a reasoning task without much mental labor.

Keywords: Logic · Automated Reasoning · Isabelle Proof Assistant

1 Introduction

Today's information systems must be not only intelligent but also trustworthy. Our contribution is two-fold. First, we program a basic theorem prover and discuss the solution to a riddle. Second, we verify the prover with the purpose of presenting modern automated reasoning tools to a wider audience.

The formal verification of a modern prover for full first-order logic is a major undertaking [16]. However, many natural language arguments can be handled in classical propositional logic, and using the Isabelle/HOL proof assistant [11] we have previously formally verified a number of such provers for formulations based on various sets of logical operators [20]. Recently we have obtained short formal soundness, completeness and termination proofs using the NAND or NOR operators [7]. However, the necessary translations of natural language statements into the modest logical language of NAND or NOR are problematic for efficiency as well as explainability reasons.

In the present work we consider a more traditional formulation of propositional logic based on falsity and implication while still obtaining short formal proofs similar to the formal proofs obtained using the NAND or NOR operators. The prover can be automatically exported to Haskell and several other functional programming languages. The prover implements a deterministic algorithm for the sequent calculus proof system [10]. In order to obtain self-contained functional programs we code a few auxiliary list programs ourselves. We rely on a deep embedding of propositional logic such that we can reason in Isabelle/HOL about the syntax and semantics of propositional logic.

The aim of the prover is not to be as efficient as possible. We could, for example, decrease the computational complexity of checking whether a sequent is a tautology in the last step of the algorithm by representing the lists of atomic propositions as binary search trees. However, the aim of our development is clarity and quick development, to showcase how Isabelle can be used for prototyping and achieving very dependable results without much extra work.

We have developed the prover and its verification in such a way that a skilled programmer without knowledge of Isabelle and with only a basic knowledge of proof theory can understand the development described in this paper. Related to this, we have recently found the tool useful when teaching computational logic and formal methods to BSc and MSc students in computer science [23].

The entire prover as well as the solution to a riddle have been formally verified in Isabelle/HOL [11] and the formalizations are available online:

`https://hol.compute.dtu.dk/Scratch.thy`

Table 1 gives an overview of the file. We explain the name *Scratch* in the beginning of Sect. 4 and the riddle is considered in Sect. 6. Logically there is no difference in Isabelle between a theorem, corollary, lemma or proposition. Normally one would add a name to all results but we have left a few unnamed.

Name		**Description**
'a form	data type	Propositional formulas with atomic propositions of type *'a*
semantics	function	Evaluates a formula under an interpretation
sc	function	Evaluates a sequent under an interpretation
member	function	Checks membership of element in list
member-iff	lemma	Equality between *member* and set-membership
common	function	Checks if lists share common element
common-iff	lemma	Equality between *common* and non-empty set-intersection
mp	function	Definition of the micro prover for use on a sequent
main	theorem	The sequent micro prover is sound and complete
prover	function	Abbreviates the micro prover for use on a single formula
(unnamed)	corollary	The formula micro prover is sound and complete
neg	function	Abbreviates negation
con	function	Abbreviates conjunction
bii	function	Abbreviates bi-implication
one	function	Abbreviates ternary exclusive disjunction
people	data type	The type of atomic propositions in the riddle
riddle	formula	The propositional formula describing the riddle
(unnamed)	proposition	Ann is shown to be a knave
(unnamed)	proposition	Cat is shown to be a knight
(unnamed)	proposition	Bob cannot be shown to be a knave
(unnamed)	proposition	Bob cannot be shown to be a knight

Table 1. Overview of the contents of the *Scratch.thy* file.

The paper is organized as follows. We first discuss related work (Sect. 2) and present the sequent calculus underlying the prover (Sect. 3). After this follows the definition of the prover, focused on functional programming in Isabelle (Sect. 4). Then comes the verification of the prover, focused on proving in Isabelle (Sect. 5). With the prover implemented and verified, we show how it can be used to solve a riddle (Sect. 6). Finally, we conclude with reflections (Sect. 7).

2 Related Work

Our work focuses on a formally verified micro prover based on the operator set containing falsity and implication. We have elsewhere implemented and verified several micro provers based on different operator sets [21]. Most recently, we made two based on just NAND and NOR, respectively. The verification of the prover in this paper is more succinct than the other provers, except the verification of the NAND and NOR provers. There are multiple reasons to prefer the operator set of implication and falsity. First, it is easier to understand the underlying sequent calculus as reasoning with implication and falsity is more intuitive than reasoning with NAND and NOR. Second, the prover in this paper is much more efficient for many problems than the NAND and NOR provers, since the translation of a natural language problem into a formula based on implication and falsity is likely much smaller than the translations into formulas based on NAND or NOR.

Shankar [17] and Michaelis and Nipkow [10] have made and verified provers for propositional logic, but for other operator sets. We focus on reasoning tools and development assisted by theorem provers and proof assistant. When it comes to theorem provers for propositional logic in general, we should mention SAT solving [2], which is much more efficient than our prover.

Blanchette [3] has made an overview of provers in Isabelle. This and our previous work is part of the IsaFoL (Isabelle Formalization of Logic) project for organizing and comparing various logics, proof systems and provers in Isabelle. For first-order logic there is leanTaP [1, 6]: a small, unverified prover in Prolog. Larger and verified provers for first-order logic also exist [15, 16, 19, 22, 24].

3 Sequent Calculus for Propositional Logic

We start with a summary of the sequent calculus for propositional logic. In the following sections we show how to program and verify a prover based on the sequent calculus.

Formulas p, q, ... in classical propositional logic are built from propositional symbols, falsity ($\bot$) and implications ($p \to q$).

We introduce the rest of the usual operators as abbreviations:

$$\neg p \equiv p \to \bot \qquad p \wedge q \equiv \neg(p \to \neg q) \qquad p \vee q \equiv \neg p \to q$$

$$p \leftrightarrow q \equiv (p \to q) \wedge (q \to p)$$

We will use some terminology that might not be widely known with regards to the calculus. An interpretation is an assignment of a truth value to each of the atomic propositions. A formula is valid if it evaluates to true under every interpretation. A sequent is two sets of formulas written as $\Gamma \vdash \Delta$. Informally, a sequent expresses that if we assume all the formulas in Γ then at least one of the formulas in Δ is true. If this property holds for a sequent, then we call it valid. In $\Gamma \vdash \Delta$, we refer to Γ as the antecedents and Δ as the consequents. Assuming the antecedents means that if we consider some interpretation where a formula in the antecedents is false, then the sequent becomes true by default. Using the rules that follow we can construct valid sequents from other valid sequents using rules of inference. In the rules there are a few formulas changed in the transformation happening from premise to conclusion, and the rest of the formulas remain unaffected. We will call these unaffected formulas "passive".

The sequent calculus consists of two axiom schemas and two inference rules:

Let Γ and Δ be finite sets of formulas.

The axiom schemas of the sequent calculus are of the form:

$$\Gamma \cup \{p\} \vdash \Delta \cup \{p\} \qquad \Gamma \cup \{\bot\} \vdash \Delta$$

The rules of the sequent calculus are the left and right introduction rules:

$$\frac{\Gamma \vdash \Delta \cup \{p\} \qquad \Gamma \cup \{q\} \vdash \Delta}{\Gamma \cup \{p \to q\} \vdash \Delta} \qquad \frac{\Gamma \cup \{p\} \vdash \Delta \cup \{q\}}{\Gamma \vdash \Delta \cup \{p \to q\}}$$

In the following explanation of the inferences, we write about the affected formulas as if they were the only ones present in the sequents. The explanations and arguments we provide for this special case also generalize to situations where there are passive formulas. For example, if a sequent is made true in some interpretation because the formula p is an antecedent and its truth value is false, then the sequent will also be made true by $p \to q$ being a consequent. If, on the other hand, a sequent is made true in an interpretation because of the truth value of a passive formula, then that passive formula will still make the derived sequent true. Similar arguments can be applied for the impossibility of introducing invalid sequents from valid ones.

The right introduction rule follows directly from the semantics (the truth of all formulas on the left-hand side imply the truth of at least one of the formulas on the right-hand side). If we can show q assuming p, then p implies q.

The left introduction rule is slightly harder to interpret. Consider that if $\Gamma \vdash \Delta$ can be shown, then for any interpretation either a formula in Γ is false or a formula in Δ is true. Showing the validity of a sequent with the empty set for Δ therefore amounts to showing that Γ is contradictory. Thus, if we can show that p is valid and that q is contradictory, we can also show that $p \to q$ is contradictory.

4 Programming the Prover

To begin the work in Isabelle we need to declare the name of our theory, import the other theories we will use, and create an Isabelle environment. The name of the theory needs to match the name of the file we are working in. Because of that we have given our theory the name *Scratch* so it can be copied and pasted into the default file displayed when starting Isabelle. *Main* is the core theory of Isabelle/HOL and contains everything necessary for implementing and verifying the prover.

theory *Scratch* **imports** *Main* **begin**

We start by defining the type of formulas. This takes a type variable, *'a*, which represents the type of atomic propositions (strings, natural numbers, and so on). A formula is one of three things: an atomic proposition of type *'a*, falsity, or an implication from one formula to another. We can observe that formulas are thus represented by binary trees, where the leafs are either propositions or falsity and parent nodes are implications from the left sub-tree to the right.

datatype *'a form*
  $= Pro\ 'a\ (\langle\cdot\rangle) \mid Falsity\ (\langle\bot\rangle) \mid Imp\ \langle\ 'a\ form\ \rangle\ \langle\ 'a\ form\ \rangle\ (\mathbf{infixr}\ \langle\to\rangle\ 0)$

The interpretation of a formula is defined as a function from the type of atomic propositions to booleans. We can define the semantics of a formula by a function taking a formula and an interpretation, and returning the truth value of the formula under that interpretation. The function is defined by pattern matching on the formula: an atomic proposition gives its value in the interpretation, falsity always gives *False*, and implication is defined in terms of Isabelle's built-in implication.

primrec *semantics* **where**
$\langle\ semantics\ i\ (\cdot n) = i\ n\ \rangle \mid$
$\langle\ semantics\ \text{-}\ \bot = False\ \rangle \mid$
$\langle\ semantics\ i\ (p \to q) = (semantics\ i\ p \longrightarrow semantics\ i\ q)\ \rangle$

We can convert lists to sets and use logical quantifiers on their elements. This allows us to express the semantics of sequents through the following one-line definition. It expresses that a sequent is true under an interpretation if all the antecedents imply at least one of the consequents.

abbreviation
  $\langle\ sc\ X\ Y\ i \equiv (\forall p \in set\ X.\ semantics\ i\ p) \longrightarrow (\exists q \in set\ Y.\ semantics\ i\ q)\ \rangle$

So that our theory is self-contained to the largest extent possible, and so that a translation of the prover to another programming language is as easy as possible, we define some functions for set operations on lists. These could be defined directly by translating the lists to sets, but then the program would not be directly translatable. Alternatively, we could have used the operations provided by Isabelle, but then the prover would not be as self-contained.

The first such function we define is *member*, which returns true if a given element exists in a list:

primrec *member* **where**
‹ *member* - [] = *False* › |
‹ *member m* (*n* # *A*) = (*m* = *n* ∨ *member m A*) ›

We can then prove that it is equivalent to the set-membership operator ∈. First, we define the lemma we want to show, then we give the proof. The proof is done by induction on the list. This is done by applying *induct A* to the proof state. This then creates two proof obligations; we need to show that the lemma holds for the empty list and for *x* # *A* (an arbitrary element *x* added to the front of *A*) if we assume it holds for *A*. By performing induction on *A* we obtain proof obligations matching the cases of the function. Since the function calls reduce nicely and the lemma is simple, we can solve both proof obligations with Isabelle's simple prover for logical rewriting: *simp*. To apply *simp* to both goals in the proof state we need to use *simp-all*.

lemma *member-iff* [*iff*]: ‹ *member m A* ⟷ *m* ∈ *set A* ›
by (*induct A*) *simp-all*

Using *member*, we can define a function for checking whether two lists contain a common element:

primrec *common* **where**
‹ *common* - [] = *False* › |
‹ *common A* (*m* # *B*) = (*member m A* ∨ *common A B*) ›

The desired property can be expressed using set operators as the intersection of the two lists not being the empty set:

lemma *common-iff* [*iff*]: ‹ *common A B* ⟷ *set A* ∩ *set B* ≠ {} ›
by (*induct B*) *simp-all*

In the later proofs Isabelle will automatically use the above translations to sets since we annotated the lemmas with [*iff*]. There is significant advantage to having the later proofs rely on these translations, as opposed to the actual definitions of *member* or *common*. We might at some point in the development change the implementation of these functions, for example by having the lists be binary search trees. It would then only be necessary to change the proofs of these two lemmas; the rest of the theory would still succeed. Thus, one can do modular development in Isabelle, with lemmas of the main properties of the different functions providing the static interfaces.

The following function (*mp*) is the core of the prover. It works by splitting implication-formulas up into their left- and right-sides at the appropriate sides of the sequents. Atomic propositions are moved into separate lists (*A* and *B*), so we can terminate when the lists containing non-atomic formulas (*C* and *D*) are empty. The inferences of the calculus are implemented quite directly, taking

a conclusion and attempting to show the premise. The left implication rule, which uses two premises, calls the prover on both premise-sequents and conjuncts the result. Thus, the prover constructs the proof tree by recursive calls on the premises which could show the given sequent. If all of these calls lead to axioms, the proof succeeds. We attempt to use the propositional axiom (utilizing the *common* function from above) when all formulas are decomposed, while the falsity axiom can be used immediately when we obtain falsity on the left-hand side of a sequent. Falsity on the right-hand side of a sequent can never be used to show validity, so it is just removed in the recursive call.

function *mp* **where**
‹ *mp A B* (·*n* # *C*) [] = *mp* (*n* # *A*) *B C* [] › |
‹ *mp A B C* (·*n* # *D*) = *mp A* (*n* # *B*) *C D* › |
‹ *mp* - - (⊥ # -) [] = *True* › |
‹ *mp A B C* (⊥ # *D*) = *mp A B C D* › |
‹ *mp A B* ((*p* → *q*) # *C*) [] = (*mp A B C* [*p*] ∧ *mp A B* (*q* # *C*) []) › |
‹ *mp A B C* ((*p* → *q*) # *D*) = *mp A B* (*p* # *C*) (*q* # *D*) › |
‹ *mp A B* [] [] = *common A B* ›
by *pat-completeness simp-all*

To be able to perform structural induction on the function, and because it is a nice result in itself, we show termination of the prover. We do this by defining a notion of size of a sequent which decreases in each recursive call. A notion of size that works is the combined size of formulas in C and D. This obviously works for the recursive calls involving atomic propositions and falsity, as they are removed from either C or D. In the calls involving implication we always remove an operator, thereby reducing the size of a formula. With this size definition, the goal produced for each recursive call can be solved by *simp*. Thus, we finish the proof by applying *simp-all*.

termination
by (*relation* ‹ *measure* (λ(-, -, *C*, *D*). ∑*p* ← *C* @ *D*. *size p*) ›) *simp-all*

Showing termination also makes automatic export to languages like Haskell possible through Isabelle's code generation (see end of Sect. 6).

5 Verifying the Prover

After constructing the prover, the big question is whether we did so correctly. In regards to this, there are really two properties that are important. Arguably the most important is soundness. We want to be sure that if our prover can construct a proof of a formula then the formula must also be valid. A property that is often harder to prove (and unattainable for some logics) is completeness, which expresses that any valid formula can be proven. The following theorem, which is the main result of our theory, shows both soundness and completeness. We do this by proving that a sequent is valid with regards to the semantics – i.e. true under every interpretation – if and only if our prover returns true.

The proof is done by structural induction on the cases of the function. This is initiated by *(induct rule: mp.induct)* which tells Isabelle to transform the main proof goal (the theorem itself) into inductive proof goals matching the cases of *mp*. Each of these goals are harder to prove than what we have considered up until this point, but still within the grasp of Isabelle's proof methods. We apply the proof methods *simp*, *blast*, *meson*, and *fast*, which is enough to solve all goals. Finding the right proof methods for a problem might seem like a challenge, but it is easy enough with Isabelle's proof-finding tool *Sledgehammer* [4]. When applying *Sledgehammer* to the proof state, it tries various theorem provers and returns the method capable of solving the current goal if it finds one. Thus, the proof methods can be found one by one, and then compacted to the one-line style we have applied.

theorem *main*: ‹ $(\forall i.\ sc\ (map\ \cdot\ A\ @\ C)\ (map\ \cdot\ B\ @\ D)\ i) \longleftrightarrow mp\ A\ B\ C\ D$ ›
  by (*induct rule*: *mp.induct*) (*simp-all*, *blast*, *meson*, *fast*)

When we have a sound and complete prover for sequents, we can derive one for formulas. This is done using the fact that a formula can be proven by proving the sequent containing it on the right-hand side and nothing else.

definition ‹ $prover\ p \equiv mp\ []\ []\ []\ [p]$ ›

We can then show soundness and completeness of the prover for formulas by using the soundness and completeness of the prover for sequents plus *simp*.

corollary ‹ $prover\ p \longleftrightarrow (\forall i.\ semantics\ i\ p)$ ›
  unfolding *prover-def* **by** (*simp flip*: *main*)

Showing completeness is in general a daunting task. One can show soundness by induction on the structure of proofs, which means that the rules being independently sound is sufficient. Completeness is achieved when all the rules together are enough to prove any valid formula. This implication from the validity of a formula to the existence of a proof cannot be done by naive induction, as not all valid formulas are constructed from other valid formulas (as is the case for proofs).

In Isabelle we are able to show both soundness and completeness in one line! This is partly because of Isabelle's intelligent proof methods, which are able to prove many theorems automatically, and partly because we couple the calculus with a proof technique by defining it as a function which can be executed. Technically, this works because completeness is now achievable by induction. In the function, the relation between premise and conclusion in each rule is equality. Thus, completeness can be shown by induction on the implications from conclusions to premises. Equivalently, we need only prove that any formula for which our prover returns true is valid, and that any formula for which our prover returns false is not valid. In this case completeness is no more complicated than soundness.

6 Using the Prover

We will use the prover to solve a riddle based on one from *What is the Name of this Book* [18] by Raymond Smullyan, which itself is a (more interesting) variation of an older riddle. It is as follows:

> *You find yourself on a desert island inhabited by knights and knaves. Knights only ever tell the truth and knaves only ever tell lies. In a clearing, you come upon three islanders: Ann, Bob, and Cat. You ask Bob how many knights there are among them. Bob answers something in a foreign language. You then ask Ann what Bob said. Ann answers: "He said there is one knight among us." Cat then reacts: "Don't listen to Ann, for she is a knave!" Who, if any, among them are knights?*

We translate the riddle to propositional logic. First, we create the following atomic propositions: A for "Ann is a knight", B for "Bob is a knight", and C for "Cat is a knight". Next we observe that a statement is true if and only if the person who said it is a knight. It becomes clear that it would be useful to have more propositional constructs than $\rightarrow$ and $\bot$. Thus, we define the operators from the start of Section 3 plus the ternary exclusive disjunction $\mathbf{one}(p, q, r)$, which is true when exactly one of the three arguments is true. A discussion on how to construct these operators comes later. We can sum up what we learn from Bob's and Ann's statements as "if Ann is a knight, then Bob is a knight if and only if there is exactly one knight", or in propositional logic: $A \rightarrow (B \leftrightarrow \mathbf{one}(A, B, C))$. We can see from the structure that we will not learn anything if Ann is a knave, which makes sense since we will then not know what Bob said. Cat's statement can be captured by "Cat is a knight if and only if Ann is a knave", or in propositional logic: $C \leftrightarrow \neg A$.

To utilize our prover we first need the operators used above. We implement them as abbreviations in Isabelle. The definition of $\neg$ (*neg*) is straightforward:

abbreviation ‹ *neg a* ≡ (*a* → ⊥) ›

$p \rightarrow \neg q$ is true exactly when either p or q is false. Thus, $p \wedge q$ can be defined as $\neg(p \rightarrow \neg q)$:

abbreviation ‹ *con a b* ≡ *neg* (*a* → *neg b*) ›

$p \leftrightarrow q$ is also straightforward:

abbreviation ‹ *bii a b* ≡ *con* (*a* → *b*) (*b* → *a*) ›

one (*one*) is defined as three bi-implications, where each one expresses that a formula is true if and only if the other two are not:

```
abbreviation ‹ one a b c ≡
  con
    (bii a (con (neg b) (neg c)))
    (con
      (bii b (con (neg a) (neg c)))
      (bii c (con (neg a) (neg b)))) ›
```

It is now possible to define the riddle itself.

First, we make a *people* data type, which is a union type of *Ann* (*A*), *Bob* (*B*), or *Cat* (*C*). This will be the type of atomic propositions in the formulas.

datatype *people* = *Ann* | *Bob* | *Cat*

We then put the facts learned from the islanders in the abbreviation *riddle*. It contains the formulas described previously, connected by a conjunction.

abbreviation ‹ *riddle* ≡ (
con
(·*Ann* → *bii* (·*Bob*) (*one* (·*Ann*) (·*Bob*) (·*Cat*)))
(*bii* (·*Ann*) (*neg* (·*Cat*)))) ›

Now we use the prover to solve the riddle. In the following lines we write statements about the riddle, and prove them using the proof method *eval*. This will export the prover to Standard ML and execute it with the given formula as input. As this relies on the correctness of the code generator and the Standard ML compiler, the result is less trust-worthy than those obtained purely by Isabelle. We could use *code-simp* instead of *eval* to make sure all evaluation stays within Isabelle, but this comes at a considerable cost to performance.

First, we can show that Ann is a knave, as *riddle* implying her not being a knight is shown valid by the prover:

proposition ‹ *prover* (*riddle* → *neg* (·*Ann*)) › **by** *eval*

Next, we show that Cat is a knight by proving that *riddle* implies *Cat*:

proposition ‹ *prover* (*riddle* → ·*Cat*) › **by** *eval*

Finally, we show that Bob can be either knight or knave. This follows quite trivially from Ann being a knave, as it means Bob might have said anything (except for "there is one knight"). For Bob, both knighthood and knavehood is unprovable. Since we have shown that the prover is complete, one of these would have succeeded if we could be sure of either.

proposition ‹ ¬*prover* (*riddle* → *neg* (·*Bob*)) › **by** *eval*

proposition ‹ ¬*prover* (*riddle* → ·*Bob*) › **by** *eval*

We finalize the theory in Isabelle:

end

The lines examining the riddle could also be done in the environment of another programming language by exporting the prover using Isabelle's code generation. The supported languages are Haskell, OCaml, Scala and Standard ML. In this way Isabelle's code generation can be used to incorporate the verified prover into a larger project in one of these programming languages. In general, one can have a project where only important core elements are verified. Thus, verification does not need to happen all at once; one can start with the most important or difficult parts.

7 Concluding Remarks

We have developed a tool for automatic reasoning in propositional logic and verified it in Isabelle/HOL. We have also shown how to use it to solve a problem of reasoning, in our case a riddle. Many problems can be modelled in propositional logic, and many more in the higher-order logic of Isabelle/HOL. Each part of our development was accompanied by machine-checked proofs, which provide very high confidence while remaining easy to write up and quick to verify. We believe that most algorithmic challenges are solvable within the framework of proof assistants, with clear advantages gained from the confidence of machine-checked proofs, and often with not much added labor.

The abbreviation feature in Isabelle/HOL is convenient, but our prover solving the riddle works on the expressions of the formula data type, cf. Table 2. Nevertheless, the execution time in Isabelle/HOL for the file is around a second.

	Count for formula: $riddle \rightarrow \neg\ Ann$
Proposition Ann	16
Proposition Bob	14
Proposition Cat	14
Falsity ($\bot$)	77
Implication ($\rightarrow$)	120

Table 2. Counting atomic propositions and operators in a formula.

One of the benefits of working with a proof assistant like Isabelle/HOL is that appropriate changes can be made and the theory is automatically verified again. For example, the soundness and completeness theorems can be verified in Isabelle/HOL if falsity is omitted from the syntax of propositional logic, in the semantics and in the sequent calculus. Of course we can no longer define negation, conjunction, disjunction and the riddle. In fact we obtain the so-called implicational propositional logic [5, 14, 25].

We should ask the question: How can we trust Isabelle? After all, it is a program and could have bugs. This has been a challenge for more than 50 years since the first proof assistant SAM (Semi-Automated Mathematics) [9]. For decades, proof assistants like HOL4 and Isabelle have relied on the so-called LCF approach: a relatively small proof kernel, using abstract data types in the programming language Standard ML to ensure soundness, only assuming that the proof kernel is soundly implemented [8, 12, 13]. In addition, proof assistants are thoroughly tested and they are more and more considered practical tools for programmers, mathematicians and scientists in general.

Acknowledgements

We would like to thank Agnes Moesgård Eschen, Asta Halkjær From, Frederik Krogsdal Jacobsen and Alexander Birch Jensen for comments on drafts.

We would also like to thank the anonymous reviewers for their suggestions. Following our riddle example, we can indeed dream about the system translating a text from a fragment of natural language to formulas in propositional logic, use our logical approach and next translate back to a conclusion expressed in natural language.

References

1. Beckert, B., Posegga, J.: leanTaP: Lean Tableau-based Deduction. Journal of Automated Reasoning **15**(3), 339–358 (1995)
2. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability. IOS press (2009)
3. Blanchette, J.C.: Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In: Mahboubi, A., Myreen, M.O. (eds.) Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019. pp. 1–13. ACM (2019)
4. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. Journal of Automated Reasoning **51**(1), 109–128 (2013). https://doi.org/10.1007/s10817-013-9278-5
5. Church, A.: Introduction to Mathematical Logic. Princeton Mathematical Series, Princeton University Press (1956)
6. Fitting, M.: leanTAP Revisited. Journal of Logic and Computation **8**(1), 33–47 (1998)
7. From, A.H., Lund, S.T., Villadsen, J.: A case study in computer-assisted metareasoning. In: González, S.R., Machado, J.M., González-Briones, A., Wikarek, J., Loukanova, R., Katranas, G., Casado-Vara, R. (eds.) Distributed Computing and Artificial Intelligence, Volume 2: Special Sessions 18th International Conference. pp. 53–63. Springer (2021). https://doi.org/10.1007/978-3-030-86887-1_5
8. Gordon, M.: From LCF to HOL: A short history. In: Proof, Language, and Interaction: Essays in Honour of Robin Milner, pp. 169–185. MIT Press, Cambridge, MA, USA (2000)
9. Guard, J.R., Oglesby, F.C., Bennett, J.H., Settle, L.G.: Semi-automated mathematics. J. ACM **16**(1), 49–62 (jan 1969). https://doi.org/10.1145/321495.321500
10. Michaelis, J., Nipkow, T.: Formalized Proof Systems for Propositional Logic. In: Abel, A., Forsberg, F.N., Kaposi, A. (eds.) 23rd International Conference on Types for Proofs and Programs (TYPES 2017). LIPIcs, vol. 104, pp. 6:1–6:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
11. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002)
12. Paulson, L.C.: Computational logic: Its origins and applications. Proc. R. Soc. A. 474 20170872 **2210** (2018). https://doi.org/10.1098/rspa.2017.0872
13. Paulson, L.C., Nipkow, T., Wenzel, M.: From LCF to Isabelle/HOL. Formal Aspects of Computing **31**(6), 675–698 (dec 2019). https://doi.org/10.1007/s00165-019-00492-1

14. Pfenning, F.: Single axioms in the implicational propositional calculus. In: Lusk, E.L., Overbeek, R.A. (eds.) 9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings. Lecture Notes in Computer Science, vol. 310, pp. 710–713. Springer (1988)
15. Ridge, T., Margetson, J.: A Mechanically Verified, Sound and Complete Theorem Prover for First Order Logic. In: Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings. pp. 294–309 (2005)
16. Schlichtkrull, A., Blanchette, J.C., Traytel, D.: A verified prover based on ordered resolution. In: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 152–165. CPP 2019, Association for Computing Machinery (2019). https://doi.org/10.1145/3293880.3294100
17. Shankar, N.: Towards Mechanical Metamathematics. Journal of Automated Reasoning **1**(4), 407–434 (1985)
18. Smullyan, R.: What Is the Name of This Book? Prentice-Hall, Inc. (1978)
19. Tourret, S., Blanchette, J.: A modular Isabelle framework for verifying saturation provers. In: Hritcu, C., Popescu, A. (eds.) CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021. pp. 224–237. ACM (2021). https://doi.org/10.1145/3437992.3439912
20. Villadsen, J.: Tautology checkers in Isabelle and Haskell. In: Calimeri, F., Perri, S., Zumpano, E. (eds.) Proceedings of the 35th Edition of the Italian Conference on Computational Logic (CILC 2020). vol. 2710, pp. 327–341. CEUR-WS.org (2020), `http://ceur-ws.org/Vol-2710/paper-21.pdf`
21. Villadsen, J.: Tautology Checkers in Isabelle and Haskell. In: Calimeri, F., Perri, S., Zumpano, E. (eds.) Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020. CEUR Workshop Proceedings, vol. 2710, pp. 327–341. CEUR-WS.org (2020)
22. Villadsen, J., Schlichtkrull, A., From, A.H.: A Verified Simple Prover for First-Order Logic. In: Konev, B., Urban, J., Rümmer, P. (eds.) Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning co-located with Federated Logic Conference 2018 (FLoC 2018), Oxford, UK, July 19th, 2018. CEUR Workshop Proceedings, vol. 2162, pp. 88–104. CEUR-WS.org (2018)
23. Villadsen, J., Jacobsen, F.K.: Using Isabelle in two courses on logic and automated reasoning. In: Ferreira, J.F., Mendes, A., Menghi, C. (eds.) Formal Methods Teaching. pp. 117–132. Springer (2021). https://doi.org/10.1007/978-3-030-91550-6_9
24. Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.: A comprehensive framework for saturation theorem proving. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12166, pp. 316–334. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_18
25. Łukasiewicz, J.: The shortest axiom of the implicational calculus of propositions. Proceedings of the Royal Irish Academy. Section A: Mathematical and Physical Sciences **52**, 25–33 (1948)