Tautology Checkers in Isabelle and Haskell *

Jørgen Villadsen

Algorithms, Logic and Graphs Section
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Richard Petersens Plads, Building 324, DK-2800 Kongens Lyngby, Denmark

Abstract. With the purpose of teaching functional programming and automated reasoning to computer science students, we formally verify a sound, complete and terminating tautology checker in Isabelle with code generation to Haskell. We describe a series of approaches and finish with a formalization based on formulas in negation normal form where the Isabelle/HOL functions consist of just 4 lines and the Isabelle/HOL proofs also consist of just 4 lines. We investigate the generated Haskell code and present a 24-line manually assembled program.

1 Introduction

Logic textbooks usually have pen-and-paper proofs only. But we find that the formalization of logic can be a very rewarding endeavor.

Our main interest in the formalizations of logic is for teaching an advanced course on automated reasoning for computer science students at the Technical University of Denmark (DTU). The prerequisites for the automated reasoning course include courses in logic and functional programming. In the course, we start with the formal verification of a sound, complete and terminating tautology checker as described in the present paper. We end with the formal verification of a proof system kernel for first-order logic [7].

Both in our courses and and in our student projects and theses we mainly use the proof assistant Isabelle/HOL [10].

`https://isabelle.in.tum.de/`

Our longer term goal is a formalized logic textbook. Rather than formally verifying existing code it is common to develop everything in Isabelle and then use the code generator at the end of the formalization. This is also the strategy that we follow for our tautology checker.

The following Isabelle formalizations have been tested in Isabelle2020/HOL and the following Haskell programs have been tested in Stackage LTS 16.10 (ghc-8.8.3).

* Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2710/paper21.pdf

The Isabelle formalizations are available here:

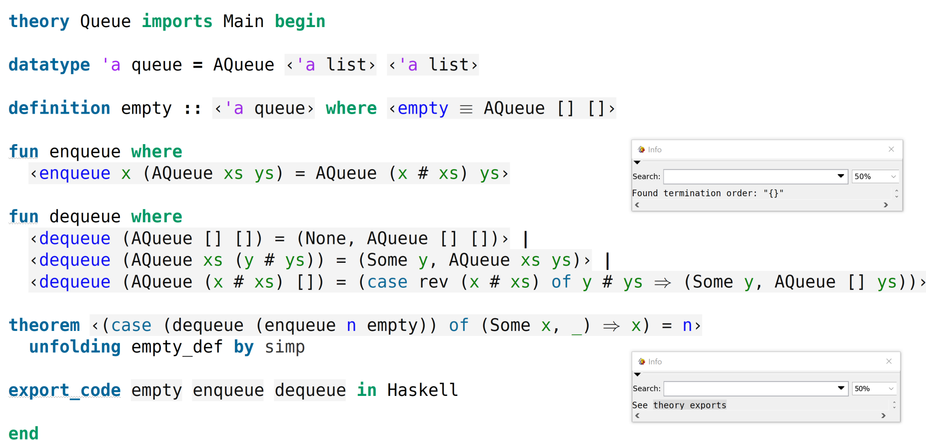
`https://github.com/logic-tools/micro`

The main formalization is in the file Prover.thy and the whole tautology checker — with Isabelle proofs of soundness, completeness and termination as well as a small example and the code generation to Haskell — even fits on a single slide (25 lines including blank lines).

We present a small Isabelle code generation example in Section 2. We discuss related work in Section 3. In Section 4 we investigate a tautology checker based on the standard two-sided sequent calculus with falsity and implication. In Section 5 we investigate tautology checkers based on a one-sided sequent calculus with negation and conjunction and also with negation and disjunction. In Section 6 we describe in details a formalization of a tautology checker based on a one-sided sequent calculus with formulas in negation normal form (NNF). Finally, we conclude with future work in Section 7.

2 Code Generation in Isabelle

Consider the following example from the Isabelle manual on code generation:



```
theory Queue imports Main begin

datatype 'a queue = AQueue ‹'a list› ‹'a list›

definition empty :: ‹'a queue› where ‹empty ≡ AQueue [] []›

fun enqueue where
  ‹enqueue x (AQueue xs ys) = AQueue (x # xs) ys›

fun dequeue where
  ‹dequeue (AQueue [] []) = (None, AQueue [] [])› |
  ‹dequeue (AQueue xs (y # ys)) = (Some y, AQueue xs ys)› |
  ‹dequeue (AQueue (x # xs) []) = (case rev (x # xs) of y # ys ⇒ (Some y, AQueue [] ys))›

theorem ‹(case (dequeue (enqueue n empty)) of (Some x, _) ⇒ x) = n›
  unfolding empty_def by simp

export_code empty enqueue dequeue in Haskell

end
```


The example provides functions for amortized queues by keeping two lists and performing a list reversal when necessary. We often leave out the types but for the constant ‘empty’ the type is needed since there is already a constant empty for the set $\emptyset$ imported from the theory Main.

We have extended the example in the manual with a theorem and a proof: **unfolding** empty_def **by** simp (which first unfolds the definition of the empty queue and then finishes the proof by simplification of the resulting expression).

The screenshot shows two ‘Info’ pop-up windows to the right. The top one reports that termination for the enqueue/dequeue functions have been automatically proved and the bottom one reports that the code generation to Haskell has been successful.

The exported lines are essentially as follows:

```
import Prelude (Maybe(..), print, reverse)

data Queue a = AQueue [a] [a]

empty = AQueue [] []

enqueue x (AQueue xs ys) = AQueue (x : xs) ys

dequeue (AQueue [] []) = (Nothing, AQueue [] [])
dequeue (AQueue xs (y : ys)) = (Just y, AQueue xs ys)
dequeue (AQueue (x : xs) []) =
  (case reverse (x : xs) of y : ys -> (Just y, AQueue [] ys))

main = print (case dequeue (enqueue 0 empty) of (Just x, _) -> x)
```

Isabelle can generate code to Haskell, OCaml, Scala and Standard ML. Note that the Isabelle theorem establishes a fact for all n (of any type) but the Haskell printout only concerns the value 0.

3 Related Work

Completeness proofs go back to Hilbert for propositional logic [16] and to Gödel for first-order logic [5]. Henkin simplified Gödel's proof [6].

Shankar [13] formalized in 1985 a tautology checker for propositional logic using the Boyer-Moore theorem prover.

Michaelis and Nipkow recently formalized propositional proof systems in Isabelle/HOL [9]. We have used their formalization as starting point but we avoid the use of a prover returning counterexamples. We have also made the prover non-sequential, i.e. deterministic, and have simplified the termination measure as well as the soundness and completeness proofs.

Nowadays many provers for propositional logic are based on SAT solving and the resolution calculus [1]. Systems like leanTAP for first-order logic are usually not formally verified [3]. Schlichtkrull has proved the completeness of first-order resolution, also in Isabelle/HOL [12].

Blanchette gives an overview of the formalized metatheory of various other logical calculi and automatic provers in Isabelle/HOL [2]. Paulson formalized Gödel's incompleteness theorems in Isabelle/HOL [11]. Kumar et al. formalized higher-order logic [8] (soundness only).

For our introductory course on logical systems and logic programming we have recently developed the Sequent Calculus Verifier (SeCaV) for first-order logic [4] but it consists of thousands of lines in Isabelle/HOL and has no decision procedure.

We recently gave a talk "A Micro Prover for Teaching Automated Reasoning" (presentation only) at the Seventh Workshop on Practical Aspects of Automated Reasoning (PAAR 2020).

4 Sequent Calculus — Falsity and Implication

We first investigate a tautology checker based on the standard two-sided sequent calculus with falsity and implication.

Formulas p, q, ... in classical propositional logic are built from propositional variables (not further specified for now), falsity ($\bot$) and implications ($p \to q$). Let Γ and Δ be finite sets of formulas.

The axioms 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 left and right introduction rules:

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

We obtain the following Haskell code (we often use our own list membership function in order to make it a bit easier to consider various other functional programming languages):

```
import Prelude ((&&), (||), (==), Bool(..), print)

data Form a = Pro a | Falsity | Imp (Form a) (Form a)

member _ [] = False
member m (n : a) = m == n || member m a

common _ [] = False
common a (m : b) = member m a || common a b

mp a b (Pro n : c) [] = mp (n : a) b c []
mp a b c (Pro n : d) = mp a (n : b) c d
mp _ _ (Falsity : _) [] = True
mp a b c (Falsity : d) = mp a b c d
mp a b (Imp p q : c) [] = mp a b c [p] && mp a b (q : c) []
mp a b c (Imp p q : d) = mp a b (p : c) (q : d)
mp a b [] [] = common a b

prover p = mp [] [] [] [p]

main = print (prover (Imp (Pro 0) (Pro 0)))
```

We leave the underlying sequent calculus implicit. The last two arguments are the two sides of a sequent. The first two arguments are lists of propositional variables that we have so far encountered in the left side and in the right side, respectively, as made clear in the two first cases of the function.

The formalization is in the following file:

Implication.thy

Unfortunately the Isabelle formalization is almost a hundred lines. We obtain a much smaller formalization by simplifying the sequent calculus, as we describe in the rest of the paper.

5 Conjunction, Disjunction and Negation

Before we consider formulas in negation normal form (NNF) we first investigate a tautology checker based on a one-sided sequent calculus with negation and conjunction.

```
import Prelude ((&&), (||), (==), Bool(..), print)

data Form a = Pro a | Neg (Form a) | Con (Form a) (Form a)

member _ [] = False
member m (n : a) = m == n || member m a

common _ [] = False
common a (m : b) = member m a || common a b

mp a b (Pro n : c) = mp (n : a) b c
mp a b (Neg (Pro n) : c) = mp a (n : b) c
mp a b (Neg (Neg p) : c) = mp a b (p : c)
mp a b (Neg (Con p q) : c) = mp a b (Neg p : Neg q : c)
mp a b (Con p q : c) = mp a b (p : c) && mp a b (q : c)
mp a b [] = common a b

prover p = mp [] [] [p]

main = print (prover (Neg (Con (Pro 0) (Neg (Pro 0)))))
```

The termination proof requires the following size function:

```
sz (Pro _) = 1
sz (Neg p) = 1 + sz p
sz (Con p q) = 2 + sz p + sz q
```

But except for the complication concerning the termination proof the above tautology checker is straightforward.

We then investigate a tautology checker based on a one-sided sequent calculus with negation and disjunction.

```
import Prelude ((&&), (||), (==), Bool(..), print)

data Form a = Pro a | Neg (Form a) | Dis (Form a) (Form a)

member _ [] = False
member m (n : a) = m == n || member m a

common _ [] = False
common a (m : b) = member m a || common a b

mp a b (Pro n : c) = mp (n : a) b c
mp a b (Neg (Pro n) : c) = mp a (n : b) c
mp a b (Neg (Neg p) : c) = mp a b (p : c)
mp a b (Neg (Dis p q) : c) = mp a b (Neg p : c) && mp a b (Neg q : c)
mp a b (Dis p q : c) = mp a b (p : q : c)
mp a b [] = common a b

prover p = mp [] [] [p]

main = print (prover (Dis (Pro 0) (Neg (Pro 0))))
```

The termination proof does not require any special size function and the above tautology checker is straightforward.

The formalizations are in the following files:

Conjunction.thy

Disjunction.thy

The formalizations are in all cases almost a hundred lines and so we turn to a tautology checker based on a one-sided sequent calculus with formulas in negation normal form (NNF).

6 Negation Normal Form

We describe a concise formalization of a tautology checker based on a one-sided sequent calculus with formulas in negation normal form (NNF).

In the micro prover we use sets of propositional variables instead of lists of propositional variables. This makes the formalization in Isabelle a bit shorter but in general makes the generated code in Haskell longer.

We start by showing the entire formalization and then we describe the formalization in details. The formalization has the usual boilerplate, like the first line with the name of the theory and the last line with the end command:

```
theory Prover imports Main begin

datatype 'a form = Atom bool 'a | Op ‹'a form› bool ‹'a form›

primrec val where
  ‹val i (Atom b n) = (if b then i n else ¬ i n)› |
  ‹val i (Op p b q) = (if b then val i p ∧ val i q else val i p ∨ val i q)›

function cal where
  ‹cal e [] = (∃ n ∈ fst e. n ∈ snd e)› |
  ‹cal e (Atom b n # s) = (if b then cal ({n} ∪ fst e, snd e) s
                                 else cal (fst e, snd e ∪ {n}) s)› |
  ‹cal e (Op p b q # s) = (if b then cal e (p # s) ∧ cal e (q # s)
                                else cal e (p # q # s))›
  by pat-completeness auto
  termination by (relation ‹measure (λ(-, s). ∑ p ← s. size p)›) auto

definition ‹prover p ≡ cal ({}, {}) [p]›

value ‹prover (Op (Atom True n) False (Atom False n))›

lemma complete: ‹cal e s ⟷
    (∀ i. ∃ p ∈ set s ∪ Atom True ‘ fst e ∪ Atom False ‘ snd e. val i p)›
  unfolding bex-Un by (induct rule: cal.induct) (auto split: if-split)

theorem ‹prover p ⟷ (∀ i. val i p)›
  unfolding complete prover-def by auto

end
```

Isabelle/HOL has a special *intelligible semi-automated reasoning* language, Isar for short [14], in which we normally formulate our proofs, but the classical reasoner (*auto*) of Isabelle/HOL [15] is so powerful that by defining the lemmas just right there is hardly any proving left for us to do.

We now describe the formalization in details. Formulas in negation normal form (NNF) are obtained using the following equivalences:

$$\phi \to \psi \equiv \neg\phi \vee \psi \qquad \phi \leftrightarrow \psi \equiv (\phi \to \psi) \wedge (\psi \to \phi)$$

$$\neg(\phi \wedge \psi) \equiv (\neg\phi \vee \neg\psi) \qquad \neg(\phi \vee \psi) \equiv (\neg\phi \wedge \neg\psi)$$

$$\neg\neg\phi \equiv \phi$$

So we are left with atomic propositional formulas, possibly negated, and build propositional formulas using conjunction and disjunction.

```
datatype 'a form = Atom bool 'a | Op ‹'a form› bool ‹'a form›
```

We use a boolean to indicate a negated atomic propositional formula and we use a boolean to choose between conjunction and disjunction. This is reflected in the semantics, the function *val*, taking a formula and an interpretation (i).

```
primrec val where
  ‹val i (Atom b n) = (if b then i n else ¬ i n)› |
  ‹val i (Op p b q) = (if b then val i p ∧ val i q else val i p ∨ val i q)›
```

The type of the interpretation (i) is automatically inferred.

```
function cal where
  ‹cal e [] = (∃ n ∈ fst e. n ∈ snd e)› |
  ‹cal e (Atom b n # s) = (if b then cal ({n} ∪ fst e, snd e) s
                                  else cal (fst e, snd e ∪ {n}) s)› |
  ‹cal e (Op p b q # s) = (if b then cal e (p # s) ∧ cal e (q # s)
                                 else cal e (p # q # s))›
  by pat-completeness auto
  termination by (relation ‹measure (λ(-, s). ∑ p ← s. size p)›) auto
```

The micro prover, the function *cal*, is a combination of the previously defined provers based on conjunction, disjunction and negation.

A function definition produces a proof obligation which expresses completeness and compatibility of patterns that is usually solved by a combination of the methods *pat_completeness* and *auto* (simplification of all goals).

Termination of the function *cal* must be proved. The termination proof also uses the method *auto* with a sum of the formula sizes as the decreasing measure.

We now obtain the tautology checker using a simple definition (it is not possible to use an abbreviation instead when we want to use code generation).

```
definition ‹prover p ≡ cal ({}, {}) [p]›
```

After the termination proof we can execute the tautology checker and move on to the proof of soundness and completeness.

value ‹*prover* (*Op* (*Atom True n*) *False* (*Atom False n*))›

The above command uses normalization by evaluation (NBE) and compiles the expression to Standard ML and executes it as such using the integration of Standard ML in Isabelle/HOL.

lemma *complete*: ‹*cal e s* $\longleftrightarrow$
  $(\forall\, i.\ \exists\, p \in$ *set s* $\cup$ *Atom True* ‘ *fst e* $\cup$ *Atom False* ‘ *snd e*. *val i p*)›
 unfolding *bex-Un* **by** (*induct rule*: *cal.induct*) (*auto split*: *if-split*)

This is the key lemma for the soundness and completeness proof.
Some comments:

- We need to state the lemma for arbitrary arguments e and s in order for the induction proof to go through.
- We formulate the validity of the sequent in the usual way by requiring that for all interpretations i there exists a formula p where the semantics is true.
- We use the function *set* to turn the list into a set of formulas because the proof automation for set theory is very strong.
- We finally solve all proof obligations using the method *auto*.

theorem ‹*prover p* $\longleftrightarrow$ $(\forall\, i.$ *val i p*)›
 unfolding *complete prover-def* **by** *auto*

This is the soundness and completeness proof based on the lemma *complete* and the definition of the prover *prover*.

export-code *prover* **in** *Haskell*

We export the code to Haskell. In order to allow for experiments we export the tautology checker *prover*. The function *cal* is added automatically.

The result of the code generation can be found in the Appendix. We present the following 24-line manually assembled program based on the code generation.

```
import Prelude ((&&), (||), (==), Bool(..), print)

fst (x, _) = x

snd (_, y) = y

any _ [] = False
any p (x : xs) = p x || any p xs

fold f (x : xs) s = fold f xs (f x s)
fold _ [] s = s

member [] _ = False
member (x : xs) y = x == y || member xs y

newtype Set a = Set [a]

bex_set (Set xs) p = any p xs

bot_set = Set []

insert_set x (Set xs) = Set (if member xs x then xs else x : xs)

member_set x (Set xs) = member xs x

sup_set (Set xs) a = fold insert_set xs a

data Form a = Atom Bool a | Op (Form a) Bool (Form a)

cal e [] = bex_set (fst e) (\ n -> member_set n (snd e))
cal e (Atom b n : s) =
  (if b then cal (sup_set (insert_set n bot_set) (fst e), snd e) s
    else cal (fst e, sup_set (snd e) (insert_set n bot_set)) s)
cal e (Op p b q : s) =
  (if b then cal e (p : s) && cal e (q : s) else cal e (p : q : s))

prover p = cal (bot_set, bot_set) [p]

main = print (prover (Op (Atom True 0) False (Atom False 0)))
```

Of course it is better to use the result of the code generation without modification but it is nevertheless relevant for teaching purposes that a simple program can easily be manually assembled.

7 Conclusion and Future Work

We have presented a formalization of a tautology checker for propositional logic with termination, soundness and completeness proofs in Isabelle/HOL. We can use the code export features of Isabelle/HOL to generate standalone Haskell, OCaml, Scala or Standard ML code. The formalized provers are for small, but adequate, fragments of classical propositional logic.

The automation in Isabelle/HOL is very powerful. In particular the function package is powerful and easy to use. The termination proof is independent of the function specification but supplying a termination proof makes an induction principle and code generation available.

The micro provers are simple but not trivial: they break down the formula in the style of a sequent calculus and not even termination is verified automatically. The micro provers are concise enough to be the first examples in a course on automated reasoning. Our approach shows how to use Isabelle/HOL and it also shows a prover program in Haskell with termination, soundness and completeness proofs.

We are working on formalizations of micro provers in other proof assistants like Agda and Coq. We also plan to consider provers for first-order logic and higher-order logic.

Acknowledgements

Thanks to Asta Halkjær From, Alexander Birch Jensen and Anders Schlichtkrull for discussions.

Appendix: Isabelle Code Generation

Listing of the three Haskell files exported from the Isabelle theory file Prover.thy

List.hs

```
{-# LANGUAGE EmptyDataDecls, RankNTypes, ScopedTypeVariables #-}

module List(fold, member, insert, removeAll) where {

import Prelude ((==), (/=), (<), (<=), (>=), (>), (+), (-), (*), (/), (**),
  (>>=), (>>), (=<<), (&&), (||), (^), (^^), (.), ($), ($!), (++), (!!), Eq,
  error, id, return, not, fst, snd, map, filter, concat, concatMap, reverse,
  zip, null, takeWhile, dropWhile, all, any, Integer, negate, abs, divMod,
  String, Bool(True, False), Maybe(Nothing, Just));
import qualified Prelude;

fold :: forall a b. (a -> b -> b) -> [a] -> b -> b;
fold f (x : xs) s = fold f xs (f x s);
fold f [] s = s;

member :: forall a. (Eq a) => [a] -> a -> Bool;
member [] y = False;
member (x : xs) y = x == y || member xs y;

insert :: forall a. (Eq a) => a -> [a] -> [a];
insert x xs = (if member xs x then xs else x : xs);

removeAll :: forall a. (Eq a) => a -> [a] -> [a];
removeAll x [] = [];
removeAll x (y : xs) = (if x == y then removeAll x xs else y : removeAll x xs);

}
```

Set.hs

```
{-# LANGUAGE EmptyDataDecls, RankNTypes, ScopedTypeVariables #-}

module Set(Set, bex, insert, member, bot_set, sup_set) where {

import Prelude ((==), (/=), (<), (<=), (>=), (>), (+), (-), (*), (/), (**),
  (>>=), (>>), (=<<), (&&), (||), (^), (^^), (.), ($), ($!), (++), (!!), Eq,
  error, id, return, not, fst, snd, map, filter, concat, concatMap, reverse,
  zip, null, takeWhile, dropWhile, all, any, Integer, negate, abs, divMod,
  String, Bool(True, False), Maybe(Nothing, Just));
import qualified Prelude;
import qualified List;

data Set a = Set [a] | Coset [a];

bex :: forall a. Set a -> (a -> Bool) -> Bool;
bex (Set xs) p = any p xs;

insert :: forall a. (Eq a) => a -> Set a -> Set a;
insert x (Coset xs) = Coset (List.removeAll x xs);
insert x (Set xs) = Set (List.insert x xs);

member :: forall a. (Eq a) => a -> Set a -> Bool;
member x (Coset xs) = not (List.member xs x);
member x (Set xs) = List.member xs x;

bot_set :: forall a. Set a;
bot_set = Set [];

sup_set :: forall a. (Eq a) => Set a -> Set a -> Set a;
sup_set (Coset xs) a = Coset (filter (\ x -> not (member x a)) xs);
sup_set (Set xs) a = List.fold insert xs a;

}
```

Prover.hs

```
{-# LANGUAGE EmptyDataDecls, RankNTypes, ScopedTypeVariables #-}

module Prover(Form, prover) where {

import Prelude ((==), (/=), (<), (<=), (>=), (>), (+), (-), (*), (/), (**),
  (>>=), (>>), (=<<), (&&), (||), (^), (^^), (.), ($), ($!), (++), (!!), Eq,
  error, id, return, not, fst, snd, map, filter, concat, concatMap, reverse,
  zip, null, takeWhile, dropWhile, all, any, Integer, negate, abs, divMod,
  String, Bool(True, False), Maybe(Nothing, Just));
import qualified Prelude;
import qualified Set;

data Form a = Atom Bool a | Op (Form a) Bool (Form a);

cal :: forall a. (Eq a) => (Set.Set a, Set.Set a) -> [Form a] -> Bool;
cal e [] = Set.bex (fst e) (\ n -> Set.member n (snd e));
cal e (Atom b n : s) =
  (if b then cal (Set.sup_set (Set.insert n Set.bot_set) (fst e), snd e) s
    else cal (fst e, Set.sup_set (snd e) (Set.insert n Set.bot_set)) s);
cal e (Op p b q : s) =
  (if b then cal e (p : s) && cal e (q : s) else cal e (p : q : s));

prover :: forall a. (Eq a) => Form a -> Bool;
prover p = cal (Set.bot_set, Set.bot_set) [p];

}
```

References

1. Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of Satisfiability*. IOS press, 2009.
2. Jasmin Christian Blanchette. Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, pages 1–13. ACM, 2019.
3. Melvin Fitting. leanTAP revisited. *J. Log. Comput.*, 8(1):33–47, 1998.
4. Asta Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull, and Jørgen Villadsen. Teaching a Formalized Logical Calculus. In *Proceedings of the 8th International Workshop on Theorem proving components for Educational software (ThEdu'19)*, 2020.
5. Kurt Gödel. *Über die Vollständigkeit des Logikkalküls*. PhD thesis, University of Vienna, 1929.
6. Leon Henkin. *The Completeness of Formal Systems*. PhD thesis, Princeton University, 1947.
7. Alexander Birch Jensen, John Bruntse Larsen, Anders Schlichtkrull, and Jørgen Villadsen. Programming and verifying a declarative first-order prover in Isabelle/HOL. *AI Communications*, 31(3):281–299, 2018.
8. Ramana Kumar, Rob Arthan, Magnus O Myreen, and Scott Owens. Self-formalisation of higher-order logic. *Journal of Automated Reasoning*, 56(3):221–259, 2016.
9. Julius Michaelis and Tobias Nipkow. Formalized proof systems for propositional logic. In A. Abel, F. Nordvall Forsberg, and A. Kaposi, editors, *23rd Int. Conf. Types for Proofs and Programs (TYPES 2017)*, volume 104 of *LIPIcs*, pages 6:1–6:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
10. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
11. Lawrence C. Paulson. A machine-assisted proof of Gödel's incompleteness theorems for the theory of hereditarily finite sets. *The Review of Symbolic Logic*, 7(3):484–498, 2014.
12. Anders Schlichtkrull. Formalization of the resolution calculus for first-order logic. *Journal of Automated Reasoning*, 61(1-4):455–484, 2018.
13. Natarajan Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1(4):407–434, 1985.
14. Makarius Wenzel. Isabelle/Isar—a generic framework for human-readable proof documents. *From Insight to Proof—Festschrift in Honour of Andrzej Trybulec*, 10(23):277–298, 2007.
15. Makarius Wenzel. The Isabelle/Isar Reference Manual. Part of the Isabelle distribution, 2020.
16. Richard Zach. Completeness before Post: Bernays, Hilbert, and the development of propositional logic. *Bulletin of Symbolic Logic*, 5(3):331–366, 1999.