Teaching Automated Reasoning and Formally Verified Functional Programming in Agda and Isabelle/HOL

Asta Halkjær From Jørgen Villadsen*
DTU Compute - Department of Applied Mathematics and Computer Science - Technical University of Denmark

We formalize two micro provers for propositional logic in Isabelle/HOL and Agda. The provers are used in an automated reasoning course at DTU where they concretize discussions of soundness and completeness. The students are familiar with functional programming beforehand but formalizing the provers, and other programs, introduces the students to formally verified functional programming in a proof assistant. Proofs that have been informal in previous courses, for instance of termination, can now be verified by the machine, and the provers provide practical examples. Similarly, the formal meta-languages provided by the formalizations clarify boundaries that can be muddled with pen and paper, for instance between syntactic and semantic arguments. We find that the automation available in Isabelle/HOL provides succinctness while the verification in Agda closer resembles functional programming.

1 Introduction

Proof assistants come in many flavors but the two major ones have been those based on simple type theory, also known as higher-order logic, like HOL Light, HOL4 and Isabelle/HOL and those based on dependent type theory like Agda, Coq and Lean. In this paper we consider their use in teaching automated reasoning and functional programming. Specifically, we translate a formalization in Isabelle/HOL [24], where the functional programs can be written succinctly but the formal verification uses additional features, into a formalization in Agda (2.6.1) [9, 39] where the verification is also functional programming.

Our main interest in the formalizations of logic is for teaching logic, automated reasoning and functional programming to computer science students at the Technical University of Denmark (DTU). We have developed the Sequent Calculus Verifier (SeCaV) [10] for first-order logic but it consists of thousands of lines in Isabelle/HOL and has no decision procedure. Our object of study here is a decision procedure for a small, but adequate, fragment of classical propositional logic. We formalize two such provers. The first returns counterexamples and the second simply answers true or false. These provers are not trivial: they break down the formula in the style of a sequent calculus and neither of the proof assistants can verify their termination automatically. Still, the provers are simple enough to be the first examples in a course. As provers for propositional logic they enable discussions of properties like soundness and completeness that are relevant in automated reasoning. At the same time, they are concrete examples of functional programming where properties like termination can be viewed from that perspective.

In 2020 we used the provers in our new MSc course "Automated Reasoning" with 27 students:

`https://kurser.dtu.dk/course/02256`

We only presented the Isabelle/HOL formalization [36]. In 2021 52 students have registered for the spring course and we now also have the Agda formalization and we put additional focus on the functional programming aspects. The formalizations, 64 lines (47 sloc) in file `Micro_Prover.thy` and 416 lines (333 sloc) in file `microprover.agda` are available here:

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

*Corresponding author: `jovi@dtu.dk`

Presentation at International Workshop on Trends in Functional Programming in Education (TFPIE 2021)

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

Abbreviations:

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

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:

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

Figure 1: The sequent calculus.

The paper strives to be self-contained so consulting the formalizations is optional. We use a fragment of propositional logic with implication and falsity only. This fragment makes it possible to create a suitable set of sample formulas for the students to consider. For ease of reference, the two-sided sequent calculus for classical propositional logic that we base the provers on is provided in fig. 1. Also for ease of reference, we give the full Isabelle/HOL formalization without examples in fig. 2.

As an illustration, here is a manually constructed sequent calculus proof in the online tool for teaching logic `http://logitext.mit.edu` (the online tool does not provide a formally verified prover):

$$\dfrac{\dfrac{\dfrac{\mathrm{A} \vdash \mathrm{B}, \mathrm{A}}{\vdash \mathrm{A} \to \mathrm{B}, \mathrm{A}}\,(\to\mathrm{r}) \qquad \mathrm{A} \vdash \mathrm{A}}{(\mathrm{A} \to \mathrm{B}) \to \mathrm{A} \vdash \mathrm{A}}\,(\to\mathrm{l})}{\vdash ((\mathrm{A} \to \mathrm{B}) \to \mathrm{A}) \to \mathrm{A}}\,(\to\mathrm{r})$$

We could also just specify the provers in a regular programming language instead of a proof assistant, but we appreciate the support given by the latter. The canonical meta-language for presenting a logic, whether in a paper or a teaching situation, is formal natural language. Not just the syntax and semantics are presented in this way, but also proof systems and their side conditions or meta-theoretical proofs themselves. This can lead to ambiguity and sometimes confusion, in particular in students. The terms we use in formal language have precise meanings but they are often learned by experience and are not always clear to newcomers. The same observation can be made about proofs concerning functional programs. This motivates our interest in precise meta-languages like simple or dependent type theory. Here we can define our logics and programs in an unambiguous and even machine-verifiable way that leaves nothing open for interpretation. While students will sometimes confuse notions when using pen and paper and, say, make semantic arguments in the middle of a syntactic proof, a formal meta-language keeps the boundaries straight. As such, we contrast the meta-languages of Isabelle/HOL and Agda in the formalizations of simple provers to better understand the strengths and weaknesses of each. We consider syntax and semantics, the specification of executable provers and their termination, and meta-theoretical results like soundness and completeness. The comparison provides some pointers on what language to choose depending on one's priorities.

```
theory Micro-Prover imports Main begin

datatype 'a form = Pro 'a | Falsity (‹ ⊥ ›) | Imp ‹ 'a form › ‹ 'a form › (infix ‹ → › 0)

primrec semantics where
‹ semantics i (Pro n) = i n › |
‹ semantics - ⊥ = False › |
‹ semantics i (p → q) = (semantics i p ⟶ semantics i q) ›

abbreviation ‹ sc X Y i ≡ (∀ p ∈ set X. semantics i p) ⟶ (∃ q ∈ set Y. semantics i q) ›

function μ where
‹ μ A B (Pro n # C) [] = μ (n # A) B C [] › |
‹ μ A B C (Pro n # D) = μ A (n # B) C D › |
‹ μ - - (⊥ # -) [] = {} › |
‹ μ A B C (⊥ # D) = μ A B C D › |
‹ μ A B ((p → q) # C) [] = μ A B C [p] ∪ μ A B (q # C) [] › |
‹ μ A B C ((p → q) # D) = μ A B (p # C) (q # D) › |
‹ μ A B [] [] = (if set A ∩ set B = {} then {A} else {}) ›
 by pat-completeness simp-all

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

lemma sat: ‹ sc (map Pro A @ C) (map Pro B @ D) (λn. n ∈ set L) ⟹ L ∉ μ A B C D ›
 by (induct rule: μ.induct) auto

theorem main: ‹ (∀ i. sc (map Pro A @ C) (map Pro B @ D) i) ⟷ μ A B C D = {} ›
 by (induct rule: μ.induct) (auto simp: sat)

primrec member where
‹ member - [] = False › |
‹ member m (n # A) = (if m = n then True else member m A) ›

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

primrec common where
‹ common - [] = False › |
‹ common A (m # B) = (if member m A then True else common A B) ›

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

function mp where
‹ 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) [] = (if mp A B C [p] then mp A B (q # C) [] else False) › |
‹ mp A B C (Imp p q # D) = mp A B (p # C) (q # D) › |
‹ mp A B [] [] = common A B ›
 by pat-completeness simp-all

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

lemma mp-iff [iff]: ‹ mp A B C D ⟷ μ A B C D = {} ›
 by (induct rule: μ.induct) simp-all

definition ‹ prover p ≡ mp [] [] [] [p] ›

corollary ‹ prover p ⟷ (∀ i. semantics i p) ›
 unfolding prover-def by (simp flip: main)

end
```

Figure 2: The full formalization in Isabelle/HOL.

We should be clear that the two formalizations are not one-to-one. In Isabelle/HOL we leave the sequent calculus implicit while in Agda we bring it out explicitly, making the formalization bigger but also more informative. Another advantage for Isabelle/HOL is that we formalize simply-typed programs in simple type theory. If we computed more evidence in our provers, the formalization in Agda would likely be smoother. We leave this for future work. Instead, we work with the same simple definitions and see how Agda lives up to the challenge.

We discuss related work in section 2. We formalize the syntax and semantics of propositional logic in section 3 and define the sequent calculus in section 4 where we also prove its soundness (we only do this in Agda since in Isabelle/HOL the underlying sequent calculus is left implicit). We define the first prover which returns counterexamples in section 5, and prove its termination, soundness and completeness. In section 6 we move to the refined second prover and show its soundness and completeness by equivalence with the first prover. We discuss teaching formally verified functional programming in section 7 along with future work, including possible variants of the provers. Finally, we conclude in section 8.

2 Related Work

The history of completeness proofs goes back to Hilbert for propositional logic [42]. Later, Gödel famously showed the completeness of first-order logic [11] and Henkin simplified Gödel's proof [13]. Kumar et al. recently mechanized the semantics of higher-order logic and a soundness proof for the inference system of the HOL Light kernel [19]. On the constructive side, Persson showed completeness for intuitionistic first-order logic in Martin-Löf type theory using the proof assistant ALF [27]. We have recently formalized a classical first-order sequent calculus and its relation to natural deduction [10].

In this paper we verify the completeness of concrete decision procedures, not just logical calculi. Shankar did the same in the Boyer-Moore theorem prover for a different fragment of propositional logic. They relate the prover to an axiomatic proof system instead of the semantics [35]. Margetson and Ridge formalized completeness of first-order logic in Isabelle/HOL [20]. Their proof is in the Beth-Hintikka style [17] using an implicit search procedure on proof trees. Later they arrive at an actual prover that can be exported to OCaml [30, 38]. Blanchette, Popescu and Traytel also employ the Beth-Hintikka style in their work on soundness and completeness proofs by coinductive methods in Isabelle/HOL [4]. They produce Haskell code for a verified semidecision procedure for first-order logic, parameterized by the proof rules for a sequent calculus or tableau system. Blanchette gives an overview of the formalized metatheory of various other logical calculi and automatic provers in Isabelle/HOL [3]. Jensen et al. verify a declarative prover for first-order logic in Isabelle/HOL [16].

The work by Michaelis and Nipkow on propositional proof systems in Isabelle/HOL is close to ours [21]. We have used their formalization as starting point and they give a similar prover but we have simplified the termination measure, return value and completeness proofs. We have also made the prover non-sequential, i.e. deterministic, which simplifies our induction proofs. They only give a prover that returns counterexamples and use it primarily as aid for their completeness proof for sequent calculus. We give a prover designed to be executable with definitions based on lists and booleans rather than sets.

Many provers for propositional logic are based on SAT solving and the resolution calculus [1]. Michaelis and Nipkow formalized a resolution calculus [21]. Schlichtkrull proved the completeness of first-order resolution [31] and the completeness has also been proved for ordered resolution [32, 33].

Paulson has formalized Gödel's incompleteness theorems in Isabelle/HOL [25, 26]. Popescu and Traytel upgraded Paulson's development, giving an abstract development of the incompleteness theorems that does not rely on any notion of model or semantic interpretation [28].

Proof assistants have been used for teaching logic and formal methods [2, 5, 14, 18] but usually without formalizations of proof systems and provers, though there are exceptions [7, 8, 10, 34, 38]. There are also examples, like Proust [29], where a proof assistant itself is developed as part of a curriculum, but again without formally verifying correctness.

Proof assistants have also been used for teaching semantics of programming languages [23] and verified analysis of algorithms [22].

3 Syntax and Semantics

In both Isabelle/HOL and Agda we deeply embed the propositional logic in the meta-logic by giving the syntax as a datatype and the semantics as a function on that datatype.

3.1 Syntax

The syntax of our propositional logic consists of propositional symbols, a logical constant for falsity and finally implication. In Isabelle/HOL it becomes the following datatype from fig. 2 where we use a type variable *'a* for the universe of propositional symbols:

datatype *'a form* = *Pro 'a* | *Falsity* (‹ ⊥ ›) | *Imp* ‹ *'a form* › ‹ *'a form* › (**infix** ‹ → › *0*)

All types in Isabelle are equipped with equality, so the use of a type variable is no complication. In Agda we fix the propositional symbols to be natural numbers since these have decidable equality and thus we are free from having to explain the intricacies of equality in Agda:

```
Id : Set
Id = ℕ
```

The precedence of the infix symbol for implication has to be given separately:

```
infixr 6 _⇒_
```

But then we are ready to declare the datatype corresponding to the syntax:

```
data Form : Set where
  falsity : Form
  pro : (x : Id) → Form
  _⇒_ : (p : Form) (q : Form) → Form
```

The constructor names match the Isabelle/HOL but in lowercase, whereas the type is now capitalized.

3.2 Semantics

The Isabelle/HOL semantics function returns a boolean in the meta-logic for whether the formula is true under a given interpretation:

primrec *semantics* **where**
‹ *semantics i* (*Pro n*) = *i n* › |
‹ *semantics* - ⊥ = *False* › |
‹ *semantics i* (*p* → *q*) = (*semantics i p* ⟶ *semantics i q*) ›

We delegate to the meta-logic implication to interpret the object logic counterpart. Similarly, we use the universal and existential quantifiers from the meta-logic to interpret a sequent:

abbreviation ‹ *sc X Y i* ≡ (∀ *p* ∈ *set X*. *semantics i p*) ⟶ (∃ *q* ∈ *set Y*. *semantics i q*) ›

We can understand X as assumptions and Y as possible conclusions: the sequent holds if the assumptions guarantee that at least one of the conclusions hold.

In Agda we introduce a shorthand for the type of an interpretation, which produces a boolean like in Isabelle:

```
Interp : Set _
Interp = Id → Bool
```

The semantics function no longer returns a simple boolean but a type that is inhabited if and only if the formula is true under the interpretation. This allows us to use existing Agda definitions like Dec, Any and All in the later development. Moreover, it makes explicit the connection between logical formulas and functional programs. Note for instance that implication is interpreted as the function space:

```
semantics : (i : Interp) (p : Form) → Set
semantics i falsity = ⊥
semantics i (pro x) = T (i x)
semantics i (p ⇒ q) = semantics i p → semantics i q
```

The function T lifts a boolean to a type that is only inhabited if the boolean is true. As an example, the identity function witnesses the validity of $\phi \to \phi$:

```
semantics-ex2 : ∀ {i p} → semantics i (p ⇒ p)
semantics-ex2 = id
```

The truth of a formula under a given interpretation is a decidable property: we can show that the formula is either true or that its truth leads to a contradiction:

```
eval : (i : Id → Bool) (p : Form) → Dec (semantics i p)
eval i falsity = no id
eval i (pro x) with i x
... | false = no id
... | true = yes tt
eval i (p ⇒ q) = eval i p →-dec eval i q
```

Consider the case for pro: the with abstraction refines our goal based on the interpretation of the propositional symbol. If it is false, the formula is false so we pick the constructor no of the Dec type to reflect this. We then need to give evidence that given a value of type ⊥, the semantics of the formula in this case, we can produce a value of type ⊥: the identity function suffices. In the true case we need to give an inhabitant of ⊤ of which there is just tt.

The sequent semantics is a function from evidence that All formulas on the left-hand side are true, to evidence that Any formula on the right-hand side is:

```
semantics' : (i : Interp) (l r : List Form) → Set
semantics' i l r = All (semantics i) l → Any (semantics i) r
```

4 Proof System

In Isabelle/HOL we leave the underlying sequent calculus implicit (unlike Michaelis and Nipkow [21]). Its definition in simple type theory makes use of features beyond functional programming. In Agda, however, we can bring it out as just another datatype. We use the infix constructor _≫_ which takes a list of formulas on both sides, with the intended interpretation given above. A proof becomes a value of the datatype as constructed by the following cases (cf. fig. 1):

```
data _≫_ : (l r : List Form) → Set where
  fls-l : ∀ {l r} → falsity :: l ≫ r
  imp-l : ∀ {p q l r} → l ≫ p :: r → q :: l ≫ r → p ⇒ q :: l ≫ r
  imp-r : ∀ {p q l r} → p :: l ≫ q :: r → l ≫ p ⇒ q :: r
  per-l : ∀ {l l' r} → l ≫ r → l ↭ l' → l' ≫ r
  per-r : ∀ {l r r'} → l ≫ r → r ↭ r' → l ≫ r'
  basic : ∀ {p l r} → p :: l ≫ p :: r
```

Our use of lists makes the proof rules simple to state but means that the ordering matters. The per-l and per-r constructors allow us to permute the left-hand and right-hand side of the sequent, respectively, with evidence of type ↭ that the two lists are permutations of each other.

4.1 Soundness

Proofs are now values like any other and lemmas become functional programs that can inspect or produce them. It is not difficult to prove that the proof system is sound. We refer to the formalization for the two implication cases:

```
proper : ∀ {l r} i → l ≫ r → semantics' i l r
proper i fls-l (() All.:: _)
proper i (imp-l sc sc') = proper-imp-l i (proper i sc) (proper i sc')
proper i (imp-r sc) = proper-imp-r i (proper i sc)
proper i (per-l sc eq) lhs = proper i sc (All-resp-↭ (↭-sym eq) lhs)
proper i (per-r sc eq) lhs = Any-resp-↭ eq (proper i sc lhs)
proper i basic (px All.:: _) = here px
```

The permutation cases follow from the library lemmas All-resp-↭ and Any-resp-↭ by passing on the evidence that the two lists are permutations of each other.

4.2 Weakening

We show a weakening lemma on the right-hand side that is useful later:

```
weaken : ∀ {l r} p → l ≫ r → l ≫ p :: r
weaken p fls-l = fls-l
weaken p (imp-l sc sc') = imp-l (per-r (weaken p sc) (swap p _ refl)) (weaken p sc')
weaken p (imp-r sc) = per-r (imp-r (per-r (weaken p sc) (swap p _ refl))) (swap _ p refl)
weaken p (per-l sc eq) = per-l (weaken p sc) eq
weaken p (per-r sc eq) = per-r (weaken p sc) (prep p eq)
weaken p basic = per-r basic (swap _ p refl)
```

We only need to permute the right-hand side occasionally, to make the rules line up and the evidence that they do so is simple.

5 First Prover

Moving to Isabelle, consider now the first prover:

```
function μ where
⟨μ A B (Pro n # C) [] = μ (n # A) B C []⟩ |
⟨μ A B C (Pro n # D) = μ A (n # B) C D⟩ |
⟨μ - - (⊥ # -) [] = {}⟩ |
⟨μ A B C (⊥ # D) = μ A B C D⟩ |
⟨μ A B ((p → q) # C) [] = μ A B C [p] ∪ μ A B (q # C) []⟩ |
⟨μ A B C ((p → q) # D) = μ A B (p # C) (q # D)⟩ |
⟨μ A B [] [] = (if set A ∩ set B = {} then {A} else {})⟩
by pat-completeness simp-all
```

The last two arguments are the two sides of a sequent and the first two are lists of propositional symbols that we have so far encountered positively and negatively, respectively. The first two cases of the function make this clear. The return value of the prover is a set of counterexamples: lists of propositional symbols that falsify the formula when they are interpreted as true and any other symbols interpreted as false. The third case returns no such lists because if we encounter falsity on the left-hand side, i.e. as an assumption, then the sequent holds trivially. Similarly, the fourth case simply drops falsity from the right-hand side. The next two cases implement the standard sequent calculus rules for implication (cf. fig. 1). The last case checks whether a counterexample exists and returns the information necessary to build one if so. The final line uses a standard method to convince Isabelle/HOL that our functional program covers all cases for the input (exhaustiveness).

5.1 Termination

To see why μ terminates, note that we always remove at least one constructor from either side of the sequent before recursing. Isabelle/HOL is not sophisticated enough to figure this out automatically, but we can assist it by giving an explicit decreasing measure that sums the sizes of formulas in the two lists:

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

In this way, we can have the computer check an otherwise informal argument about our functional program. Students can even try out different measures and study the cases that fail.

In Agda we take a very different approach after Bove and Capretta [6] and reify the call graph as an inductive accessibility predicate:

```
data mpAcc : (a b : List Id) (c d : List Form) → Set where
  mp-fls-l : ∀ a b c → mpAcc a b (falsity :: c) []
  mp-fls-r : ∀ a b c d → mpAcc a b c d → mpAcc a b c (falsity :: d)
  mp-pro-l : ∀ x a b c → mpAcc (x :: a) b c [] → mpAcc a b (pro x :: c) []
  mp-pro-r : ∀ x a b c d → mpAcc a (x :: b) c d → mpAcc a b c (pro x :: d)
  mp-imp-l : ∀ p q a b c → mpAcc a b c [ p ] → mpAcc a b (q :: c) [] → mpAcc a b (p ⇒ q :: c) []
  mp-imp-r : ∀ p q a b c d → mpAcc a b (p :: c) (q :: d) → mpAcc a b c (p ⇒ q :: d)
  mp-basic : ∀ a b → mpAcc a b [] []
```

We give a constructor for each case of the recursive function. Values of type mpAcc a b c d are call graphs for the function when run on those arguments. Since the datatype is inductive, to have a value of it means that every branch of the recursion eventually reaches the base case, i.e. that the function terminates. As such, we seek to prove that the type is inhabited for all choices of a, b, c and d.

To start out, we give an explicit size function like the one provided for us in Isabelle:

```
size : Form → ℕ
size falsity = 1
size (pro _) = 1
size (p ⇒ q) = 1 + size p + size q
```

We proceed by well-founded recursion which requires us to bundle the arguments into a tuple. The following abbreviation makes this simpler:

```
mk-mpAcc : Args → Set
mk-mpAcc (a , b , c , d) = mpAcc a b c d
```

We define the same measure as in Isabelle:

```
SeqTo‌ℕ : Seq → ℕ
SeqTo‌ℕ (l , r) = sum (map size l) + sum (map size r)
```

We use it to prove the lemma of interest (nested inside ArgsTo‌ℕ):

```
mpAccTotal : (x : Args) → mk-mpAcc x
mpAccTotal = wfRec _ _ go
  where
  open WF.InverseImage {A = Args} {_¡_ = _¡_} ArgsTo‌ℕ using (wellFounded)
  open WF.All (wellFounded ¡-wellFounded) using (wfRec)
  go : (x : Args) → (∀ y → y ¡Args x → mk-mpAcc y) → mk-mpAcc x
  go (a , b , falsity :: c , []) rec = mp-fls-l a b c
  go (a , b , c , falsity :: d) rec = mp-fls-r a b c d
    (rec (a , b , c , d) (suc+suc (sum (map size c)) (sum (map size d))))
```

```
go (a , b , pro x :: c , []) rec = mp-pro-l x a b c
  (rec (x :: a , b , c , []) (s≤s ≤-refl))
go (a , b , c , pro x :: d) rec = mp-pro-r x a b c d
  (rec (a , x :: b , c , d) (suc+suc (sum (map size c)) (sum (map size d))))
go (a , b , p ⇒ q :: c , []) rec = mp-imp-l p q a b c
  (rec (a , b , c , [ p ]) (s≤s (mp1 (size p) (size q))))
  (rec (a , b , q :: c , []) (s≤s (mp2 (size p) (size q))))
go (a , b , c , p ⇒ q :: d) rec = mp-imp-r p q a b c d
  (rec (a , b , p :: c , q :: d) (mp3 (size p) (size q)))
go (a , b , [] , []) rec = mp-basic a b
```

The cases are no different from the ones that arise in Isabelle/HOL, but in Isabelle/HOL we can discharge them automatically with the simplifier and here we need to prove them manually. It is simply a matter of recursing and shuffling arithmetical expressions.

Before we define the first prover in Agda we need a helper function to express the base case: evidence that the two lists of propositional symbols share a common element. We use a dependent sum constructed from a propositional symbol and a proof that it occurs in both lists:

```
Shared : (xs ys : List Id) → Set
Shared xs ys = (Σ Id λ e → e ∈ xs × e ∈ ys)
```

Whether two lists contain a shared element is also a decidable property:

```
shared : ∀ xs ys → Dec (Shared xs ys)
shared [] ys = no (λ (_ , in-[] , _) → ¬Any[] in-[])
shared (x :: xs) ys with x id∈? ys
... | no absent with shared xs ys
... | no disj = no (contraposition (Shared-contract x xs ys absent) disj)
... | yes (v , in-xs , in-ys) = yes (v , there in-xs , in-ys)
shared (x :: xs) ys | yes prf = yes (x , here refl , prf)
```

The Shared-contract lemma drops an element of the first list when given evidence that it does not occur in the second:

```
Shared-contract : ∀ x xs ys → ¬ x ∈ ys → Shared (x :: xs) ys → Shared xs ys
Shared-contract x xs ys absent (v , here px , in-ys) rewrite px = ⊥-elim (absent in-ys)
Shared-contract x xs ys absent (v , there in-xs , in-ys) = v , in-xs , in-ys
```

We then define the first prover in Agda over the mpAcc datatype:

```
μ-acc : ∀ {a b c d} → mpAcc a b c d → List (List Id)
μ-acc (mp-fls-l _ _ c) = []
μ-acc (mp-fls-r _ _ _ d s) = μ-acc s
μ-acc (mp-pro-l x _ _ c s) = μ-acc s
μ-acc (mp-pro-r x _ _ _ d s) = μ-acc s
μ-acc (mp-imp-l p q _ _ c s t) = μ-acc s ++ μ-acc t
μ-acc (mp-imp-r p q _ _ _ d s) = μ-acc s
μ-acc (mp-basic a b) = if ⌊ shared a b ⌋ then [] else [ a ]
```

In the base case we reduce the evidence produced by shared to a boolean but it is useful to compute it first: when proving lemmas about μ-acc we will need the evidence and the function will then unfold as we want it to.

We obtain the prover itself by composition with mpAccTotal:

```
μ : (a b : List Id) (c d : List Form) → List (List Id)
μ a b c d = μ-acc (mpAccTotal (a , b , c , d))
```

Isabelle/HOL automatically defines an induction principle for terminating functions that matches their recursive patterns. We can now prove things in a similar way (without reproving the termination of our lemma), by induction over the mpAcc datatype. Moreover, we can reuse the termination proof mpAccTotal for the second prover whereas we prove termination twice in Isabelle/HOL (cf. fig. 2).

5.2 Sound and Complete

In Isabelle/HOL we very easily prove the first prover sound and complete by looking at whether or not it returns a counterexample:

```
lemma sat: ‹ sc (map Pro A @ C) (map Pro B @ D) (λn. n ∈ set L) ⟹ L ∉ μ A B C D ›
  by (induct rule: μ.induct) auto

theorem main: ‹ (∀ i. sc (map Pro A @ C) (map Pro B @ D) i) ⟷ μ A B C D = {} ›
  by (induct rule: μ.induct) (auto simp: sat)
```

5.2.1 Soundness

The Agda story is more complicated. We may start on soundness by noticing that if a formula occurs on both sides of a sequent then we can find a sequent calculus derivation via the appropriate permutations:

```
shared≫ : ∀ {l r} p → (lhs : p ∈ l) (rhs : p ∈ r) → l ≫ r
shared≫ p lhs rhs with ∈-∃++ lhs — ∈-∃++ rhs
... — l1 , l2 , l-prf — r1 , r2 , r-prf rewrite l-prf — r-prf =
  per-r (per-l basic (↭-sym (shift p l1 l2))) (↭-sym (shift p r1 r2))
```

We can then prove by induction on the call graph that if no counterexamples are returned, there is a derivation (see the formalization for the proof):

```
proven-acc : ∀ {a b c d} → (s : mpAcc a b c d) (prf : μ-acc s ≡ []) → c ++ map pro a ≫ d ++ map pro b
```

We compose it with μAccTotal to obtain proven and thereby soundness:

```
sound : ∀ a b c d → (prf : μ a b c d ≡ []) → ∀ i → semantics' i (c ++ map pro a) (d ++ map pro b)
sound a b c d prf i = proper i (proven a b c d prf)
```

5.2.2 Completeness

We have shown that the formula is valid if the first prover returns an empty set. We now consider the other case: when one or more counterexamples are returned. Our falsifying interpretation is based on list membership where we reduce the Dec value to a boolean:

```
counter-sem : List Id → Interp
counter-sem a x = ⌊ x id∈? a ⌋
```

Again, calculating the evidence and then reducing it to a boolean will be useful in the proofs.

Our task is then to show that the sequent can be falsified by a returned counterexample: if we assume it is satisfied (*assm*) then we reach a contradiction:

```
counter-acc : ∀ {a b c d} → (s : mpAcc a b c d) (xs : List Id) (prf : xs ∈ μ-acc s)
  → ¬ semantics' (counter-sem xs) (c ++ map pro a) (d ++ map pro b)
counter-acc (mp-fls-l _ _ c) xs () assm
counter-acc (mp-fls-r _ b _ d s) xs prf assm = counter-acc s xs prf λ lhs →
```

```
  drop-falsity (d ++ map pro b) (counter-sem xs) (assm lhs)
counter-acc (mp-pro-l x a _ c s) xs prf assm = counter-acc s xs prf λ lhs →
  assm (All-resp-↭ (shift (pro x) c (map pro a)) lhs)
counter-acc (mp-pro-r x _ b _ d s) xs prf assm = counter-acc s xs prf λ lhs →
  Any-resp-↭ (↭-sym (shift (pro x) d (map pro b))) (assm lhs)
counter-acc (mp-imp-l p q a b c s t) xs prf assm with ∈-++⁻ (μ-acc s) prf — eval (counter-sem xs) p
... — inj₁ in-s — no pf = counter-acc s xs in-s (λ lhs → there (assm ((⊥-elim ∘ pf) All.:: lhs)))
... — inj₁ in-s — yes pt = counter-acc s xs in-s (const (here pt))
... — inj₂ in-t — _ = counter-acc t xs in-t (λ { (qt All.:: lhs) → assm ((λ _ → qt) All.:: lhs) })
counter-acc (mp-imp-r p q a b c d s) xs prf assm = counter-acc s xs prf λ { (pt All.:: lhs) →
  Any-update (p ⇒ q) q (d ++ map pro b) (assm lhs) (_$ pt) }
counter-acc (mp-basic a b) xs prf assm with shared a b
counter-acc (mp-basic a b) xs (here px) assm — no no-common rewrite px =
  counter-common a b no-common (assm (counter-lhs [] a))
```

The first case is trivially absurd since no counterexamples are returned there. In the second case we use the following lemma to drop a falsity from the right-hand side of a sequent before recursing:

```
drop-falsity : ∀ l i → (prf : Any (semantics i) (falsity :: l)) → Any (semantics i) l
drop-falsity l i (there prf) = prf
```

In the propositional cases we use permutations to shift the symbol w.r.t. the concatenation. The cases for implication are more interesting. On the left-hand side we look at *prf* to determine which branch of the recursion the returned counterexample came from and at the semantics of the antecedent p. If the counterexample came from the first recursive call (on *[p]*) then we need to recurse on this side with a proof that the sequent is satisfied by the counterexample. If p is false under the counterexample then the implication we are working on is satisfied and we can apply our assumption *assm*. If the interpretation satisfies p then the sequent we are recursing on is trivially satisfied, as p occurs on the right-hand side. Finally, if the counterexample emerged from adding q to the assumptions then we can assume that q holds, in which case the implication does too and we can again appeal to *assm*. If the implication occurs on the right-hand side then we can assume from the nature of the recursive call that p is true and need to show that q is too given our implication. The function Any-update does the "heavy" lifting:

```
Any-update : {A : Set} {P : A → Set} (v v' : A) (xs : List A)
  (prf : Any P (v :: xs)) (f : P v → P v') → Any P (v' :: xs)
Any-update v v' xs (here pv) f = here (f pv)
Any-update v v' xs (there prf) f = there prf
```

We obtain completeness by considering whether or not the first prover returns counterexamples:

```
complete : ∀ a b c d → (valid : ∀ i → semantics' i (c ++ map pro a) (d ++ map pro b)) → μ a b c d ≡ []
complete a b c d valid with case-∈ (μ a b c d)
... — inj₁ prf = prf
... — inj₂ (xs , prf) = ⊥-elim (counter a b c d xs prf (valid (counter-sem xs)))
```

If it does not then we are done and if it does then we use the counter result (derived from counter-acc) to derive a contradiction from the validity assumption; a classic technique.

6 Second Prover

We now reach the second prover, which returns true or false instead of a list of lists and whose execution is thus simpler than the first one. Our goal is to show a correspondence with the first prover and thereby easily derive completeness, which would be harder to prove directly because the prover produces less information. We give simple definitions to check membership and the existence of a common element in Isabelle/HOL and use them to define the prover (and prove termination):

```
primrec member where
⟨member - [] = False⟩ |
⟨member m (n # A) = (if m = n then True else member m A)⟩

primrec common where
⟨common - [] = False⟩ |
⟨common A (m # B) = (if member m A then True else common A B)⟩

function mp where
⟨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) [] = (if mp A B C [p] then mp A B (q # C) [] else False)⟩ |
⟨mp A B C (Imp p q # D) = mp A B (p # C) (q # D)⟩ |
⟨mp A B [] [] = common A B⟩
by pat-completeness simp-all

termination by (relation ⟨measure (λ(-,-,C,D). ∑p ← C @ D. size p)⟩) simp-all
```

To define it in Agda we first need equivalents of the functions *member* and *common*:

```
member : (v : Id) (xs : List Id) → Bool
member v [] = false
member v (x :: xs) = if v ≡ᵇ x then true else member v xs


common : (xs ys : List Id) → Bool
common [] ys = false
common (x :: xs) ys = if member x ys then true else common xs ys
```

The second prover is then extremely simple:

```
mp-acc : ∀ {a b c d} → mpAcc a b c d → Bool
mp-acc (mp-fls-l _ _ c) = true
mp-acc (mp-fls-r _ _ _ d s) = mp-acc s
mp-acc (mp-pro-l x _ _ c s) = mp-acc s
mp-acc (mp-pro-r x _ _ _ d s) = mp-acc s
mp-acc (mp-imp-l p q _ _ c s t) = mp-acc s ∧ mp-acc t
mp-acc (mp-imp-r p q _ _ _ d s) = mp-acc s
mp-acc (mp-basic a b) = common a b
```

As mentioned, we can reuse the termination proof for the first prover:

```
mp : (a b : List Id) (c d : List Form) → Bool
mp a b c d = mp-acc (mpAccTotal (a , b , c , d))
```

One disadvantage compared to the Isabelle/HOL code (cf. fig. 2) is that we build the mpAccTotal data structure explicitly to recurse on it. In Isabelle/HOL we have no such middle step.

In Isabelle/HOL we carefully set up correspondences between *member*, *common* and their set variants:

```
lemma member-iff [iff]: ⟨member m A ⟷ m ∈ set A⟩
 by (induct A) simp-all

lemma common-iff [iff]: ⟨common A B ⟷ set A ∩ set B ≠ {}⟩
 by (induct B) simp-all
```

This allows us to show the equivalence of the two provers using just the simplifier:

```
lemma mp-iff [iff]: ‹mp A B C D ⟷ μ A B C D = {}›
  by (induct rule: μ.induct) simp-all
```

The Agda versions are more complicated. For example, here we prove that if an element is a member of a list then our function member returns true:

```
∈-member : ∀ v xs → v ∈ xs → T (member v xs)
∈-member v .(x :: _) (here {x} px) rewrite to T-≡ ⟨$⟩ ≡⇒≡ᵇ v x px = tt
∈-member v .(x :: xs) (there {x} {xs} prf) = T-if-true-else-x (v ≡ᵇ x) (member v xs) (∈-member v xs prf)
```

On the whole, proving the equivalence requires lots of fiddling with the evidence:

```
μ⇔mp-acc : ∀ {a b c d} → (s : mpAcc a b c d) → μ-acc s ≡ [] ⇔ T (mp-acc s)
μ⇔mp-acc (mp-fls-l _ _ c) = record { to = Eq.const tt ; from = Eq.const refl }
μ⇔mp-acc (mp-fls-r _ _ _ d s) = μ⇔mp-acc s
μ⇔mp-acc (mp-pro-l x _ _ c s) = μ⇔mp-acc s
μ⇔mp-acc (mp-pro-r x _ _ _ d s) = μ⇔mp-acc s
μ⇔mp-acc (mp-imp-l p q _ _ c s t) = record
  { to = record { _⟨$⟩_ = λ empty → from T-∧ ⟨$⟩
        ( to (μ⇔mp-acc s) ⟨$⟩ ++-conicalˡ _ _ empty
        , to (μ⇔mp-acc t) ⟨$⟩ ++-conicalʳ _ _ empty )
        ; cong = cong _ }
  ; from = record { _⟨$⟩_ = λ both → []-++-[]
        (from (μ⇔mp-acc s) ⟨$⟩ proj₁ (to T-∧ ⟨$⟩ both))
        (from (μ⇔mp-acc t) ⟨$⟩ proj₂ (to T-∧ ⟨$⟩ both))
  ; cong = cong _ } }
μ⇔mp-acc (mp-imp-r p q _ _ _ d s) = μ⇔mp-acc s
μ⇔mp-acc (mp-basic a b) with common a b — inspect (common a) b
... — false — PE.[ eq ] rewrite shared-common a b — eq = record
  { to = record { _⟨$⟩_ = λ () ; cong = cong _ } ; from = record { _⟨$⟩_ = λ () ; cong = cong _ } }
... — true — PE.[ eq ] rewrite sym (shared-common a b) — eq =
  record { to = Eq.const tt ; from = Eq.const refl }
```

We use function equivalence, ⇔, to show that we can go back and forth between the two provers' return values. We use this result to show soundness and completeness of the second prover in the general case:

```
sound-complete' : ∀ a b c d → (∀ i → semantics' i (c ++ map pro a) (d ++ map pro b)) ⇔ T (mp a b c d)
sound-complete' a b c d = record
  { to = record { _⟨$⟩_ = λ valid → to (μ⇔mp a b c d) ⟨$⟩ complete a b c d valid ; cong = cong _ }
  ; from = record { _⟨$⟩_ = λ sc → sound a b c d (from (μ⇔mp a b c d) ⟨$⟩ sc) ; cong = cong _ } }
```

We may boil it down to just a prover for a single formula:

```
prover : Form → Bool
prover p = mp [] [] [] [ p ]
```

It is sound and complete (see the formalization for the proof):

```
sound-complete : ∀ p → (∀ i → semantics i p) ⇔ T (prover p)
```

We can use Agda to run the prover on different examples, with natural numbers as propositional symbols:

```
mp-ex1 : T (prover $ pro 0 ⇒ pro 1 ⇒ pro 0)
mp-ex1 = tt

mp-ex2 : T (prover $ ((pro 0 ⇒ pro 1) ⇒ pro 0) ⇒ pro 0)
mp-ex2 = tt
```

This concludes our Agda development and we briefly return to Isabelle/HOL in comparison.
We can define a prover in the same way in Isabelle/HOL:

definition ‹*prover p* ≡ *mp* [] [] [] [*p*]›

This simplifies the soundness and completeness statement:

corollary ‹*prover p* ⟷ (∀ *i*. *semantics i p*)›
unfolding *prover-def* **by** (*simp flip*: *main*)

In Isabelle/HOL we use the simplifier to run an example. Since the syntax is parametric over the choice of propositional symbols, we can use the unit type here where there is only one symbol:

proposition ‹*prover* (((*Pro* () → ⊥) → ⊥) → *Pro* ())›
by *code-simp*

The prover is defined as a definition instead of an abbreviation, a distinction that exists in Isabelle/HOL, to allow for code generation. We expand on this in the following section.

7 Teaching Formally Verified Functional Programming

In our teaching we primarily use Isabelle. An important aspect of this is code generation which turns the verified functional programs into code in a regular programming language, which is more familiar to the students.

We can run the Agda code via the type checker but we can also export it to Haskell or JavaScript. In Isabelle/HOL we can use *eval* instead of *code-simp* to compile an example to Standard ML and execute it as such using the integration of Standard ML in Isabelle/HOL. Isabelle/HOL also has code export features that we can use to generate standalone Standard ML, Haskell, OCaml or Scala code to compile and run on our own. Doing so turns our formalization effort into an actual executable prover in a regular programming language.

Consider for instance the SML code in fig. 3 which is the result of running the following line in Isabelle:

export_code *prover Falsity Pro Imp* **in** *SML*

This exports the *prover* program and the formula constructors. There are a few things to note:

- The `HOL` structure is used to handle polymorphic equality in the style of Haskell's dictionary translation [12].
- The auxiliary functions *member*, *common* and *mp* are not exposed outside the `Micro_Prover` structure.
- The lists in Isabelle are compiled to the native SML lists (and likewise for booleans).
- The *form* datatype is compiled as expected into an SML datatype.
- The translation preserves the structure of the Isabelle functions.

In fig. 4 we provide a `prover'` definition that fixes the equality to be SML's built-in polymorphic equality (the definition is eta-expanded due to the value restriction in SML). We then run the prover on two small examples, one that returns false as expected and one that returns true. A similar example in an online editor is shown in fig. 5.

Besides running the provers we also ask the students to use the sequent calculus in order to construct the full proofs. When using lists instead of sets we need contraction and permutation rules to assure that neither the order nor the multiplicity of occurrences of elements matters. We combine these rules into general so-called set rules and prove soundness in Isabelle/HOL. Using these rules we cover 4 example proofs in the lecture and the students construct 4 exercise proofs, with help from the teaching assistant as needed. In the mandatory assignment the students also construct 4 proofs. We mix shorter and longer proofs as shown in the following overview.

```
structure HOL : sig
  type 'a equal
  val eq : 'a equal -> 'a -> 'a -> bool
end = struct

type 'a equal = {equal : 'a -> 'a -> bool};
val equal = #equal : 'a equal -> 'a -> 'a -> bool;

fun eq A_ a b = equal A_ a b;

end; (*struct HOL*)

structure Micro_Prover : sig
  datatype 'a form = Pro of 'a | Falsity | Imp of 'a form * 'a form
  val prover : 'a HOL.equal -> 'a form -> bool
end = struct

datatype 'a form = Pro of 'a | Falsity | Imp of 'a form * 'a form;

fun member A_ uu [] = false
  | member A_ m (n :: a) = (if HOL.eq A_ m n then true else member A_ m a);

fun common A_ uu [] = false
  | common A_ a (m :: b) = (if member A_ m a then true else common A_ a b);

fun mp A_ a b (Pro n :: c) [] = mp A_ (n :: a) b c []
  | mp A_ a b c (Pro n :: d) = mp A_ a (n :: b) c d
  | mp A_ uu uv (Falsity :: uw) [] = true
  | mp A_ a b c (Falsity :: d) = mp A_ a b c d
  | mp A_ a b (Imp (p, q) :: c) [] =
    (if mp A_ a b c [p] then mp A_ a b (q :: c) [] else false)
  | mp A_ a b c (Imp (p, q) :: d) = mp A_ a b (p :: c) (q :: d)
  | mp A_ a b [] [] = common A_ a b;

fun prover A_ p = mp A_ [] [] [] [p];

end; (*struct Micro_Prover*)
```

Figure 3: Result of exporting the second prover from Isabelle/HOL to SML

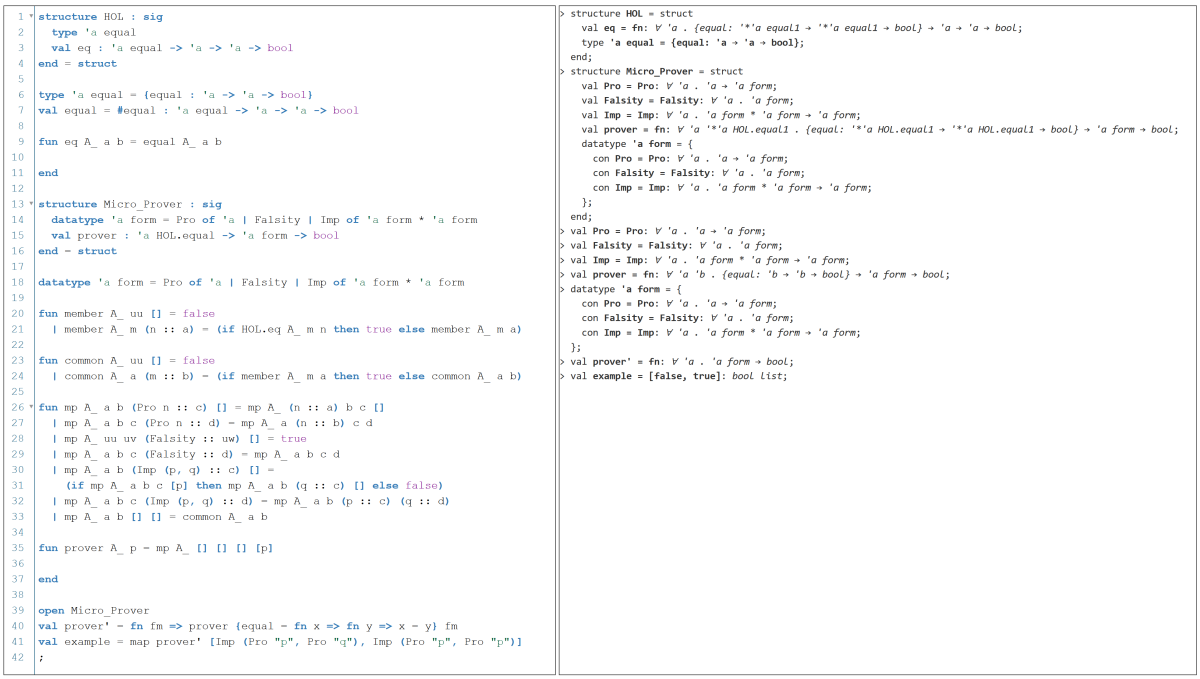
Examples	$p \to p$	1 proof step
	$p \to (p \to q) \to q$	3 proof steps
	$p \to q \to q \to p$	4 proof steps
	$p \to \neg\neg p$	4 proof steps using abbreviation for $\neg$
Exercises	$p \to q \to p$	3 proof steps
	$(p \to q \to r) \to (p \to q) \to p \to r$	9 proof steps
	$\neg p \to \neg\neg\neg p$	3 proof steps using abbreviation for $\neg$
	$p \vee \neg p$	1 proof step using abbreviation for $\neg$ and $\vee$
Assignment	$(p \to q) \to p \to q$	1 proof step
	$\neg\neg p \to p$	5 proof steps using abbreviation for $\neg$
	$p \wedge (p \to q) \to q$	7 proof steps using abbreviation for $\wedge$
	$p \wedge q \to r \to p \wedge r$	10 proof steps using abbreviation for $\wedge$

In the proof steps we count the number of left and right introduction rules as well as the left and right set rules.

The GitHub page contains a number of tautology checkers in Isabelle/HOL similar to the ones presented here but based on other fragments of propositional logic: conjunction and negation, disjunction and negation, and negation normal form (NNF). Most of them end with a code export to Haskell (this can easily be inserted if not) and we have compared the resulting code elsewhere [37].

```
open Micro_Prover;
val prover' = fn fm => prover {equal = fn x => fn y => x = y} fm;
prover' (Imp (Pro "p", Pro "q")); (* false *)
prover' (Imp (Pro "p", Pro "p")); (* true *)
```

Figure 4: Two examples of using the prover in SML



```
structure HOL : sig
  type 'a equal
  val eq : 'a equal -> 'a -> 'a -> bool
end = struct

type 'a equal = {equal : 'a -> 'a -> bool}
val equal = #equal : 'a equal -> 'a -> 'a -> bool

fun eq A_ a b = equal A_ a b

end

structure Micro_Prover : sig
  datatype 'a form = Pro of 'a | Falsity | Imp of 'a form * 'a form
  val prover : 'a HOL.equal -> 'a form -> bool
end = struct

datatype 'a form = Pro of 'a | Falsity | Imp of 'a form * 'a form

fun member A_ uu [] = false
  | member A_ m (n :: a) = (if HOL.eq A_ m n then true else member A_ m a)

fun common A_ uu [] = false
  | common A_ a (m :: b) = (if member A_ m a then true else common A_ a b)

fun mp A_ a b (Pro n :: c) [] = mp A_ (n :: a) b c []
  | mp A_ a b c (Pro n :: d) = mp A_ a (n :: b) c d
  | mp A_ uu uv (Falsity :: uw) [] = true
  | mp A_ a b c (Falsity :: d) = mp A_ a b c d
  | mp A_ a b (Imp (p, q) :: c) [] =
    (if mp A_ a b c [p] then mp A_ a b (q :: c) [] else false)
  | mp A_ a b c (Imp (p, q) :: d) = mp A_ a b (p :: c) (q :: d)
  | mp A_ a b [] [] = common A_ a b

fun prover A_ p = mp A_ [] [] [] [p]

end

open Micro_Prover
val prover' = fn fm => prover {equal = fn x => fn y => x = y} fm
val example = map prover' [Imp (Pro "p", Pro "q"), Imp (Pro "p", Pro "p")]
;
```

```
> structure HOL = struct
    val eq = fn: ∀ 'a . {equal: '*'a equal1 → '*'a equal1 → bool} → 'a → 'a → bool;
    type 'a equal = {equal: 'a → 'a → bool};
  end;
> structure Micro_Prover = struct
    val Pro = Pro: ∀ 'a . 'a → 'a form;
    val Falsity = Falsity: ∀ 'a . 'a form;
    val Imp = Imp: ∀ 'a . 'a form * 'a form → 'a form;
    val prover = fn: ∀ 'a '*'a HOL.equal1 . {equal: '*'a HOL.equal1 → '*'a HOL.equal1 → bool} → 'a form → bool;
    datatype 'a form = {
      con Pro = Pro: ∀ 'a . 'a → 'a form;
      con Falsity = Falsity: ∀ 'a . 'a form;
      con Imp = Imp: ∀ 'a . 'a form * 'a form → 'a form;
    };
  end;
> val Pro = Pro: ∀ 'a . 'a → 'a form;
> val Falsity = Falsity: ∀ 'a . 'a form;
> val Imp = Imp: ∀ 'a . 'a form * 'a form → 'a form;
> val prover = fn: ∀ 'a 'b . {equal: 'b → 'b → bool} → 'a form → bool;
> datatype 'a form = {
    con Pro = Pro: ∀ 'a . 'a → 'a form;
    con Falsity = Falsity: ∀ 'a . 'a form;
    con Imp = Imp: ∀ 'a . 'a form * 'a form → 'a form;
  };
> val prover' = fn: ∀ 'a . 'a form → bool;
> val example = [false, true]: bool list;
```

Figure 5: Prover example in the online editor `https://sosml.org/`

A possible future direction is to consider a prover for a larger set of propositional connectives and see how this impacts the formalization. Defining connectives in terms of each other leads to fewer cases to consider, but also makes the prover work on different syntax than the user may be expecting. Starting from a fuller set of connectives could make this gap smaller. It may also serve as a starting point for considering other logics than classical logic, where the connectives are not necessarily definable in terms of each other.

For negation normal form (NNF), however, it does not make sense to consider any more connectives. We include the prover based on NNF in fig. 6. The datatype now only has two constructors, *Atom* and *Op*, but each includes a boolean field. For *Atom* the boolean specifies whether or not the propositional symbol is negated and for *Op* we use true to denote conjunction and false to denote disjunction. The semantics function, now dubbed *val*, makes this clear. Since the syntax is simpler, so is the underlying sequent calculus and prover *cal*. Similarly, the shift from explicit functions on lists to built-in operations on sets helps the automation. Thus, Isabelle/HOL can more easily verify the prover's soundness and completeness. The translation to, say, Haskell becomes less straightforward and now relies on a provided set library, but fig. 6 showcases how strong a meta-language Isabelle/HOL is. We can verify a terminating, sound and complete prover in less than twenty lines of code, including the definitions of the syntax and semantics and an example.

For future work we want to formalize a similar prover in Agda, but we also want to emphasize the value of our current approach, where we formalize the proof system as well as the prover. Defining the sequent calculus as an inductive datatype makes it less abstract, say, in a teaching situation: a proof is just a value of this type! And having a concrete representation allows us to show properties like soundness or weakening as functions that take sequent calculus proofs as arguments.

```
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
```

Figure 6: A prover based on negation normal form and sets in Isabelle/HOL

In summary, we emphasize the following observations:

- Both proof assistants are clearly up to the task of formalizing these micro provers but the resulting formalizations differ greatly in style.
- The simplifier (*simp*) and its extension into the classical reasoner (*auto*) of Isabelle/HOL [40] are so powerful that by defining the lemmas just right there is hardly any proving left for us to do.
- The Agda experience is very different: we need to manually prove even the simple arithmetic identities that arise in the termination proof but this also leaves more clues to the workings of the proof.
- Yet, the Agda formalization has a certain simplicity to it because everything is "just" datatypes and corresponding functions.
- These functions may return complicated pieces of evidence for a given proposition but it still feels a lot like functional programming.
- Working in Isabelle/HOL feels more like theorem proving with its special *intelligible semi-automated reasoning* language, Isar for short [41], in which we write our proofs.

At no point during the Agda formalization did we miss *proof by contradiction* or similar classical techniques. Rather, we occasionally felt inclined to move away from booleans entirely in favor of something more constructive and we want to investigate this direction further in future work. We are also interested in formalizing the same provers in other proof assistants like Coq or Lean with more available automation compared to Agda. Yet another possibility is to take advantage of the generic Isabelle framework and use a constructive type theory like Isabelle/CTT instead of Isabelle/HOL.

8 Conclusion

We have presented a formalization of a decision procedure for propositional logic with termination, soundness and completeness proofs, contrasting the meta-languages of Isabelle/HOL and Agda.

The Isabelle/HOL formalization was useful in our Agda work but exactly because the automation is so powerful, we sometimes found ourselves manually expanding the Isabelle/HOL proof to better understand the details and translate them. The Agda version is almost the opposite since every proof term is written out in the text. In both formalizations, however, the proof state is mostly hidden and it is not always clear what goal a particular expression solves.

We particularly appreciate the Isabelle/HOL approach and its powerful function package when we need to prove termination. In Isabelle/HOL the termination proof is independent of the function specification but supplying a termination proof makes an induction principle and code generation available. In Agda the specification of our provers is entangled with the termination proof and if we want to export code for our provers we need to verify that it is optimized away. A future direction is to see whether we can define the provers in a structurally recursive manner so that Agda can verify their termination automatically. Perhaps sized types can provide a solution [15] but a different calculus is likely necessary.

As mentioned we have used the Isabelle/HOL formalization in our 2020 and 2021 course on automated reasoning and functional programming. The time/memory performance of the micro provers is in no way optimal but we find it a good starting point for student projects to experiment with formalizations and perhaps improve the decision procedures. Even though the students are familiar with functional programming beforehand, our approach using formalized provers, and other programs and algorithms, introduces the students to formally verified functional programming in a proof assistant.

Acknowledgements We thank Jens Carl Moesgård Eschen, Frederik Krogsdal Jacobsen and Alexander Birch Jensen for comments on the paper and Uma Zalakain and Guillaume Allais for hints with the Agda formalization.

References

[1] Armin Biere, Marijn Heule & Hans van Maaren (2009): *Handbook of Satisfiability*. IOS Press.

[2] William Billingsley & Peter Robinson (2007): *Student Proof Exercises Using MathsTiles and Isabelle/HOL in an Intelligent Book*. *J. Autom. Reason.* 39(2), pp. 181–218.

[3] Jasmin Christian Blanchette (2019): *Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk)*. In Assia Mahboubi & Magnus O. Myreen, editors: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, ACM, pp. 1–13.

[4] Jasmin Christian Blanchette, Andrei Popescu & Dmitriy Traytel (2017): *Soundness and completeness proofs by coinductive methods*. *Journal of Automated Reasoning* 58(1), pp. 149–179.

[5] Sebastian Böhne & Christoph Kreitz (2017): *Learning how to Prove: From the Coq Proof Assistant to Textbook Style*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 6th International Workshop on Theorem proving components for Educational software, ThEdu@CADE 2017, Gothenburg, Sweden, 6 Aug 2017*, *EPTCS* 267, pp. 1–18. Available at `https://doi.org/10.4204/EPTCS.267.1`.

[6] Ana Bove & Venanzio Capretta (2005): *Modelling general recursion in type theory*. *Mathematical Structures in Computer Science* 15(4), pp. 671–708.

[7] Joachim Breitner (2016): *The Incredible Proof Machine (Invited Talk)*. In Gilles Dowek, Daniel R. Licata & Sandra Alves, editors: *Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2016, Porto, Portugal, June 23, 2016*, ACM, p. 5:1.

[8] Joachim Breitner (2016): *Visual Theorem Proving with the Incredible Proof Machine*. In Jasmin Christian Blanchette & Stephan Merz, editors: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, *Lecture Notes in Computer Science* 9807, Springer, pp. 123–139.

[9] The Agda Developers (2020): *The Agda Wiki*. `https://wiki.portal.chalmers.se/agda/pmwiki.php`.

[10] Asta Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull & Jørgen Villadsen (2020): *Teaching a Formalized Logical Calculus*. In: *Proceedings of the 8th International Workshop on Theorem proving components for Educational software (ThEdu'19)*, pp. 73–92.

[11] Kurt Gödel (1929): *Über die Vollständigkeit des Logikkalküls*. Ph.D. thesis, University of Vienna.

[12] Florian Haftmann & Tobias Nipkow (2010): *Code Generation via Higher-Order Rewrite Systems*. In Matthias Blume, Naoki Kobayashi & Germán Vidal, editors: *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, *Lecture Notes in Computer Science* 6009, Springer, pp. 103–117. Available at `https://doi.org/10.1007/978-3-642-12251-4_9`.

[13] Leon Henkin (1947): *The Completeness of Formal Systems*. Ph.D. thesis, Princeton University.

[14] Martin Henz & Aquinas Hobor (2011): *Teaching Experience: Logic and Formal Methods with Coq*. In Jean-Pierre Jouannaud & Zhong Shao, editors: *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, *Lecture Notes in Computer Science* 7086, Springer, pp. 199–215.

[15] John Hughes, Lars Pareto & Amr Sabry (1996): *Proving the correctness of reactive systems using sized types*. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 410–423.

[16] Alexander Birch Jensen, John Bruntse Larsen, Anders Schlichtkrull & Jørgen Villadsen (2018): *Programming and verifying a declarative first-order prover in Isabelle/HOL*. *AI Communications* 31(3), pp. 281–299.

[17] S. C. Kleene (1967): *Mathematical Logic*. Wiley, London.

[18] Maria Knobelsdorf, Christiane Frede, Sebastian Böhne & Christoph Kreitz (2017): *Theorem Provers as a Learning Tool in Theory of Computation*. In Josh Tenenberg, Donald Chinn, Judy Sheard & Lauri Malmi, editors: *Proceedings of the 2017 ACM Conference on International Computing Education Research, ICER 2017, Tacoma, WA, USA, August 18-20, 2017*, ACM, pp. 83–92.

[19] Ramana Kumar, Rob Arthan, Magnus O Myreen & Scott Owens (2016): *Self-formalisation of higher-order logic*. *Journal of Automated Reasoning* 56(3), pp. 221–259.

[20] James Margetson & Tom Ridge (2004): *Completeness theorem*. *Archive of Formal Proofs*. `http://isa-afp.org/entries/Completeness.html`, Formal proof development.

[21] Julius Michaelis & Tobias Nipkow (2018): *Formalized Proof Systems for Propositional Logic*. In A. Abel, F. Nordvall Forsberg & A. Kaposi, editors: *23rd Int. Conf. Types for Proofs and Programs (TYPES 2017)*, *LIPIcs* 104, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 6:1–6:16.

[22] Tobias Nipkow, Manuel Eberl & Maximilian P. L. Haslbeck (2020): *Verified Textbook Algorithms - A Biased Survey*. In Dang Van Hung & Oleg Sokolsky, editors: *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, *Lecture Notes in Computer Science* 12302, Springer, pp. 25–53.

[23] Tobias Nipkow & Gerwin Klein (2014): *Concrete Semantics - With Isabelle/HOL*. Springer.

[24] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer.

[25] Lawrence C. Paulson (2014): *A Machine-Assisted Proof of Gödel's Incompleteness theorems for the Theory of Hereditarily Finite Sets*. *Rev. Symb. Log.* 7(3), pp. 484–498.

[26] Lawrence C. Paulson (2015): *A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle*. *J. Autom. Reason.* 55(1), pp. 1–37.

[27] Henrik Persson (1996): *Constructive completeness of intuitionistic predicate logic*. Licenciate thesis, Chalmers University of Technology.

[28] Andrei Popescu & Dmitriy Traytel (2019): *A Formally Verified Abstract Account of Gödel's Incompleteness Theorems*. In Pascal Fontaine, editor: Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings, Lecture Notes in Computer Science 11716, Springer, pp. 442–461.

[29] Prabhakar Ragde (2016): *Proust: A Nano Proof Assistant*. In Johan Jeuring & Jay McCarthy, editors: Proceedings of the 4th and 5th International Workshop on Trends in Functional Programming in Education, TFPIE 2016, Sophia-Antipolis, France, and University of Maryland, College Park, MD, USA, June 2, 2015, and June 7, 2016, *EPTCS* 230, pp. 63–75. Available at `https://doi.org/10.4204/EPTCS.230.5`.

[30] Tom Ridge & James Margetson (2005): *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.

[31] Anders Schlichtkrull (2018): *Formalization of the resolution calculus for first-order logic*. Journal of Automated Reasoning 61(1-4), pp. 455–484.

[32] Anders Schlichtkrull, Jasmin Blanchette, Dmitriy Traytel & Uwe Waldmann (2020): *Formalizing Bachmair and Ganzinger's Ordered Resolution Prover*. J. Autom. Reason. 64(7), pp. 1169–1195.

[33] Anders Schlichtkrull, Jasmin Christian Blanchette & Dmitriy Traytel (2019): *A verified prover based on ordered resolution*. In Assia Mahboubi & Magnus O. Myreen, editors: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019, ACM, pp. 152–165.

[34] Anders Schlichtkrull, Jørgen Villadsen & Andreas Halkjær From (2018): *Students' Proof Assistant (SPA)*. In Pedro Quaresma & Walther Neuper, editors: Proceedings 7th International Workshop on Theorem proving components for Educational software, THedu@FLoC 2018, Oxford, United Kingdom, 18 july 2018, *EPTCS* 290, pp. 1–13. Available at `https://doi.org/10.4204/EPTCS.290.1`.

[35] Natarajan Shankar (1985): *Towards mechanical metamathematics*. Journal of Automated Reasoning 1(4), pp. 407–434.

[36] Jørgen Villadsen (2020): *A Micro Prover for Teaching Automated Reasoning*. In: Seventh Workshop on Practical Aspects of Automated Reasoning (PAAR 2020) — Presentation Only / Online Papers, pp. 1–12. Available at `http://paar2020.gforge.inria.fr/`.

[37] Jørgen Villadsen (2020): *Tautology Checkers in Isabelle and Haskell*. In Francesco Calimeri, Simona Perri & Ester Zumpano, editors: Proceedings of the 35th Edition of the Italian Conference on Computational Logic (CILC 2020), Rende, Italy, 13-15 October 2020, *CEUR Workshop Proceedings* 2710, CEUR-WS.org, pp. 327–341. Available at `http://ceur-ws.org/Vol-2710/paper-21.pdf`.

[38] Jørgen Villadsen, Anders Schlichtkrull & Andreas Halkjær From (2018): *A Verified Simple Prover for First-Order Logic*. In Boris Konev, Josef Urban & Philipp Rümmer, editors: Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning (PAAR 2018) co-located with Federated Logic Conference 2018 (FLoC 2018), Oxford, UK, 19 July 2018, *CEUR Workshop Proceedings* 2162, CEUR-WS.org, pp. 88–104. Available at `http://ceur-ws.org/Vol-2162/paper-08.pdf`.

[39] Philip Wadler & Wen Kokke (2019): *Programming Language Foundations in Agda*. Available at `http://plfa.inf.ed.ac.uk/`.

[40] Makarius Wenzel (2020): *The Isabelle/Isar Reference Manual*. Part of the Isabelle distribution.

[41] Markus Wenzel (1999): *Isar - A Generic Interpretative Approach to Readable Formal Proof Documents*. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring & Laurent Théry, editors: Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings, Lecture Notes in Computer Science 1690, Springer, pp. 167–184.

[42] Richard Zach (1999): *Completeness before Post: Bernays, Hilbert, and the development of propositional logic*. Bulletin of Symbolic Logic 5(3), pp. 331–366.