Core Logic: A Formalization of Sequent Calculus for Classical Implicational Logic

Frederik Krogsdal Jacobsen, Simon Tobias Lund, and Jørgen Villadsen

Technical University of Denmark

For use in course 02156 Logical Systems and Logic Programming 2023

1 Introduction

We present a formalization of a sequent calculus and an automatic theorem prover implementing a complete proof search strategy for classical implicational logic. The definitions and proofs are formalized in Isabelle/HOL, and are presented in figs. 1 to 3. In this paper, we will go through the formalization, explaining the fine points of each definition and proof.

```
datatype form
  = Pro nat (‹·›)
  | Imp form form (infixr ‹⟶› 100)

primrec semantics (infix ‹⊨› 50) where
  ‹I ⊨ ·n = I n› |
  ‹I ⊨ p → q = (I ⊨ p ⟶ I ⊨ q)›

abbreviation sc (‹⟦_⟧›) where ‹⟦I⟧ X Y ≡ (∀p ∈ set X. I ⊨ p) ⟶ (∃q ∈ set Y. I ⊨ q)›

inductive SC (infix ‹≫› 50) where
  Imp_L: ‹p → q # X ≫ Y› if ‹X ≫ p # Y› and ‹q # X ≫ Y› |
  Imp_R: ‹X ≫ p → q # Y› if ‹p # X ≫ q # Y› |
  Set_L: ‹X' ≫ Y› if ‹X ≫ Y› and ‹set X' = set X› |
  Set_R: ‹X ≫ Y'› if ‹X ≫ Y› and ‹set Y' = set Y› |
  Basic: ‹p # _ ≫ p # _›

function mp where
  ‹mp A B [] [] = (set A ∩ set B ≠ {})› |
  ‹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 (·n # C) [] = mp (n # A) B C []› |
  ‹mp A B C (·n # D) = mp A (n # B) C D›
  by pat_completeness simp_all

termination
  by (relation ‹measure (λ(_, _, C, D). 2 * (∑p ← C @ D. size p) + size (C @ D))›) simp_all

lemma main: ‹(∀I. ⟦I⟧ (map · A @ C) (map · B @ D)) ⟷ mp A B C D›
  by (induct rule: mp.induct) (auto 5 2)

definition prover (‹⊢›) where ‹⊢ p ≡ mp [] [] [] [p]›

theorem prover_correct: ‹⊢ p ⟷ (∀I. I ⊨ p)›
  unfolding prover_def by (simp flip: main)

export_code ⊢ in SML
```

Fig. 1. Definitions, proof of correctness for the prover, and code generation.

We begin at the top of fig. 1, with the definition of a formula, *form*. Formulas are defined as a *datatype*, which means that the type of formulas contains every formula that can be finitely constructed using the two *constructors*: *Pro* and *Imp*. The constructor *Pro* represents atomic propositions, which can take on a value of either true or false. Atomic propositions represent concrete statements such as "it is raining" or "the Earth is flat", which are either true or false. Working with abstract atomic propositions lets us state things more generally than if we worked with concrete statements. The names of atomic propositions are represented using the type of natural numbers, *nat*, and we define the syntax $\cdot$ to mark a natural number as being the name of an atomic proposition, such that we can write e.g. $\cdot 2$ for the atomic proposition named 2. We could also have used any other type (e.g. strings) for names, but natural numbers are efficient and easy to type in examples. The constructor *Imp* represents implication from one formula to another. We define the infix syntax $\rightarrow$ such that we can write e.g. $p \rightarrow q$ for the implication from some formula p to some formula q (as usual, the arrow associates to the right). Our logic is called implicational because it has no logical operators except implication.

We now have a syntax for writing formulas, but we have not yet defined what the formulas actually mean. We noted above that atomic propositions can take on a value of either true or false, and we will now formalize this notion. We thus introduce the notion of an *interpretation*, I, which is a (total) function assigning truth values to each atomic proposition name (i.e. natural number). For instance, the interpretation could assign the value true to an atomic proposition representing the statement "it is raining" if it is currently raining. An interpretation gives meaning to the individual atomic propositions, but we still need to formally define the meaning of formulas. The function *semantics* defines what a formula means under a specific interpretation by performing a recursive case analysis on the constructors a formula can be made up of. The function is primitive recursive (i.e. it is possible to pre-determine an upper bound on the number of iterations it will need), and we immediately define the infix syntax $\vDash$ such that we can write e.g. $I \vDash p$ for the semantics of the formula p under the interpretation I. If the formula is an atomic proposition, the semantics of the formula is just the truth value that the interpretation assigns to that atomic proposition, which can be retrieved by applying the function I to the atomic proposition name n. If the formula is an implication from one formula to another, we delegate the semantics to the metalogic by first recursively determining the truth value of each subformula, then using the implication arrow from HOL (note the slightly longer arrow on the right in the definition of the semantics function) to determine the truth value of the implication.

We now have a way to determine the meaning of any formula in our logic under a specific interpretation: use the semantics function to calculate the truth value. We are however, often interested in more general questions about formulas, and in particular we would like to know whether some formulas evaluate to true no matter which interpretation they are evaluated under. We call such formulas *valid* and define a *sequent calculus* as a proof system with which to show that

formulas are valid. A *sequent* consists of two lists of formulas: the antecedent and the consequent. The intended meaning of a sequent is that if all of the formulas in the antecedent evaluate to true (under some interpretation), then one of the formulas in the consequent also evaluates to true. This idea is formalized in the sequent semantics *sc*, which evaluates to true if all of the formulas in the antecedent X (viewed as a set) evaluating to true (under the interpretation I) implies that at least one of the formulas in the consequent Y (viewed as a set) evaluates to true.

Like formulas, sequents are valid if their semantics evaluate to true under all interpretations. We can reason about the validity of sequents using a number of rules, which together with an axiom form the sequent calculus *SC*. The sequent calculus is an inductively defined relation, which means that the relation includes the smallest (but still infinite) set of sequents that satisfy the rules and the axiom. We immediately define the syntax $\gg$ such that we can write e.g. $X \gg Y$ for the statement that the sequent consisting of the antecedent X and the consequent Y is valid. The basic axiom of our sequent calculus (*Basic*) states that if the same formula is the head of both the antecedent and the consequent, then the sequent is valid. This should be intuitively clear, since validity means we assume that all of the formulas in the antecedent evaluate to true, and then have to find a formula in the consequent that evaluates to true.

The rule *Imp_L* states that a sequent which has an implication $p \to q$ at the head of its antecedent is valid if both the sequent which has p at the head of its consequent and the sequent which has q at the head of its antecedent, and are otherwise identical to the original sequent (without the implication), are valid. We can explain this rule by case analysis: $p \to q$ evaluates to true if p evaluates to false or if q evaluates to true. If p evaluates to false, then the validity of $X \gg p \# Y$ entails that one of the formulas in Y evaluates to true, making the original sequent valid since it also has Y as its consequent. If q evaluates to true, then the validity of $q \# X \gg Y$ entails that one of the formulas in Y evaluates to true, which again makes the original sequent valid.

The rule *Imp_R* states that a sequent which has an implication $p \to q$ at the head of its consequent is valid if the sequent which has p at the head of its antecedent and q at the head of its consequent, and is otherwise identical to the original sequent without the implication, is valid. We can explain this rule by case analysis on whether q evaluates to true. If it does, then the validity of $p \# X \gg q \# Y$ entails the validity of the original sequent, since $p \to q$ evaluates to true if both p and q evaluate to true. If q evaluates to false, we proceed by case analysis on whether p evaluates to true. If p evaluates to true, then $p \# X \gg q \# Y$ entails that one of the formulas in Y evaluates to true, making the original sequent valid. If p evaluates to false, then $p \to q$ evaluates to true, making the original sequent valid.

The rules *Set_L* and *Set_R* extend the system with so-called *structural rules* by allowing us to reorder, duplicate, and deduplicate the elements in the antecedent and the consequent of a sequent, respectively. We formalize this by stating that the validity of a sequent can be established by providing another

valid sequent which has the same antecedent (respectively consequent) when they are viewed as sets.

This completes the definition of our sequent calculus. Note that we have so far only informally argued why the calculus seems appropriate as a proof system for validity. We will return to the formal proof later.

2 An automatic prover

The sequent calculus can be used to prove the validity of formulas as we will see in section 4, but the proofs quickly become large and unwieldy. Since propositional logic is decidable, it is possible to design a program which can decide whether any formula is valid or not. In this section, we will go through a simple implementation of such a program, namely the function *mp*. This function essentially implements an ordering on the rules of the sequent calculus SC by iterating through the antecedent and consequent and reducing them to lists of names of atomic propositions. The function is defined recursively by pattern matching and takes four arguments:

A: a list of atomic proposition names found in the antecedent
B: a list of atomic proposition names found in the consequent
C: an antecedent
D: a consequent

The intention is that only C and D are non-empty when calling the function, and that A and B are only used in the recursive steps.

The first case of the function is matched when both C and D are empty, and simulates the rule *Basic* from the sequent calculus by checking whether the lists of atomic proposition names A and B have any elements in common, returning true if they do and false otherwise. This is the only case that terminates the function, just as *Basic* is the only axiom in the proof system.

The next case is matched when an implication is at the head of the antecedent and the consequent is empty. This case simulates the rule *Imp_L* and similarly recurses into two cases where the two sides of the implication are added to the consequent and the antecedent, respectively. Requiring that the consequent is empty before matching on the antecedent imposes an order on the function cases, simplifying the termination argument later. The third case is matched when an implication is at the head of the consequent, and simulates the rule *Isa_R* quite directly.

The fourth is matched when an atomic proposition is at the head of the antecedent. The name of the atomic proposition is added to the list of names of atomic propositions A. The fifth case is similar, but for the consequent, and instead adds the name to B.

The *mp* does not obviously terminate. This means that we can not rely on Isabelle/HOL's usual automation for defining the function like we did with the primitive recursive *semantics* function. Instead, we use the *function* keyword when defining the function to manually take of the the necessary arguments.

The first thing we need to prove is that the patterns we have provided in the cases are actually enough to match every possible argument. It is easy to see that our patterns do in fact match every possible argument, and the proof can be completed by the automatic *pat_completeness* tactic.

Next, we need to prove that the function eventually terminates for every possible argument, and this time the argument is less trivial. We will prove termination by defining a *measure* (a function mapping the arguments of *mp* to a natural number) and proving that this measure decreases whenever the function recurses. Since the measure decreases with every recursion, it will eventually reach zero, at which point the function must terminate, since the measure can no longer decrease any further. The patterns defining *mp* only match on the contents of C and D, so the measure does not need to depend on A and B. We define the measure as two times the number of implications in C and D plus the length of the concatenation of C and D. In the first case of *mp* this measure is zero, since both lists are empty. The second and third cases both decrease the number of implications by one in each recursive call. The third case, however, also increases the length of the concatenation of C and D by one in its recursive call, which is why we multiply the number of implications by two in the measure such that it will decrease overall. The fourth and fifth cases both decrease the length of the concatenation of C and D by one in their recursive calls. The calculation can be done automatically in every case by the simplification tactic *simp_all* once the measure has been defined. The definition of the measure itself relies on the fact that Isabelle/HOL automatically defines a *size* function for every datatype which counts the number of recursive constructors applied. For lists, this means that the size function returns the number of elements added to the list, which corresponds to the length of the list. For formulas, this means that the size function returns the number of implications in the formula.

We have now proven that the function *mp* terminates for every possible argument, but it remains to prove that the function actually returns true if and only if the sequent it is given is valid. The lemma *main* states a slightly more general result, namely that the prover returns true if and only if the sequent with the concatenation of A (with the atomic proposition names turned into atomic propositions by applying $\cdot$) and C as antecedent and the concatenation of B and D as consequent is valid. The proof is by induction on the cases of *mp*, and each case can be proven by the automatic tactic *auto*, which in this case needs two arguments to increase the bound on the number of irreversible ("unsafe") steps to try while proving the cases. The proof of each case is similar to the reasoning in the previous description of how the cases of *mp* simulate the rules.

Having a prover for sequents, we can easily define a prover for individual formulas by simply letting the antecedent be empty and the consequent contain only the formula. We call this the *prover*, and define the syntax $\vdash$ such that we can write e.g. $\vdash p$ for the result of applying the prover to the formula p. The theorem *prover_correct* specializes the lemma *main* to say that the prover returns true for a formula if and only if that formula is valid. The theorem follows

immediately from reading lemma *main* in from right to left and the definition of the prover.

Now that we have a prover which we know to be correct, we might want to use it. We can of course run the prover directly Isabelle/HOL using the rewriting system, but we will get results much faster by exporting the prover to a regular programming language, then compiling it and running it like any other program. This will also allow us to run the prover without Isabelle. To do this, we ask Isabelle to export code for the prover into Standard ML (SML), which will automatically generate code for all necessary definitions. The proofs will of course not be translated, since Standard ML is just a programming language, not a proof assistant. If we trust the translation process, however, we can conclude that the correctness theorem will also hold from the exported program, since it is generated from the Isabelle definitions.

3 Relating the calculus and the prover

So far we have defined a sequent calculus and a prover, and shown that the prover works correctly. We have yet, however, to show that the sequent calculus is appropriate as a proof system for validity. Since we already know that the prover works correctly, we can show this by relating the prover and the sequent calculus.

```
lemma MP: ‹mp A B C D ⟹ set X ⊇ set (map · A @ C) ⟹ set Y ⊇ set (map · B @ D) ⟹ X ≫ Y›
proof (induct A B C D arbitrary: X Y rule: mp.induct)
  case (1 A B)
  obtain n where ‹n ∈ set A› ‹n ∈ set B›
    using 1(1) by auto
  then have ‹set (·n # X) = set X› ‹set (·n # Y) = set Y›
    using 1(2,3) by auto
  then show ?case
    using Set_L Set_R Basic by metis
next
  case (2 A B p q C)
  have ‹set (map · A @ C) ⊆ set X› ‹set (map · B) ⊆ set (p # Y)›
    using 2(4,5) by auto
  moreover have ‹set (map · A @ C) ⊆ set (q # X)› ‹set (map · B) ⊆ set Y›
    using 2(4,5) by auto
  ultimately have ‹(p → q) # X ≫ Y›
    using 2(1-3) Imp_L by simp
  then show ?case
    using 2(4) Set_L by fastforce
next
  case (3 A B C p q D)
  have ‹set (map · A @ C) ⊆ set (p # X)› ‹set (map · B @ D) ⊆ set (q # Y)›
    using 3(3,4) by auto
  then have ‹X ≫ (p → q) # Y›
    using 3(1,2) Imp_R by simp
  then show ?case
    using 3(4) Set_R by fastforce
qed simp_all

theorem OK: ‹(∀I. ⟦I⟧ X Y) ⟷ X ≫ Y›
  by (rule, use MP main[of ‹[]› _ ‹[]› _] in simp, induct rule: SC.induct) auto

corollary ‹[] ≫ [p] ⟷ (∀I. I ⊨ p)›
  using OK by force
```

Fig. 2. Proof of correctness for the sequent calculus.

We first prove lemma *MP*: if *mp* returns true, then there is a derivation in the sequent calculus for a sequent which contains at least the arguments given to *mp*. The proof is by induction on the cases of *mp*, keeping the larger sequent for the sequent calculus general. We need to state the lemma using a potentially larger sequent for the sequent calculus derivation to obtain a strong enough induction hypothesis for the proof.

For the first case, where both C and D are empty, we first note that A and B have an element in common since *mp* returned true. This means that the larger antecedent X and the larger consequent Y both contain that element, so adding it again does nothing when viewing them as sets. Knowing this, we use *Set_L* and *Set_R* to move the common element to the front of X and Y respectively, then conclude using the *Basic* rule.

For the second case, where the head of the antecedent is an implication and D is empty, we first note that the antecedent is included in X and that the consequent is included in the larger consequent $p \# Y$. Conversely, we next note that the antecedent is also included in the larger antecedent $q \# X$ and that the consequent is included in Y. From these two observations, we can instantiate the two induction hypotheses such that they fit rule *Imp_L* to obtain a derivation of $(p \to q) \# X \gg Y$. Finally, since $p \to q$ is already in X, we can use *Set_L* to deduplicate X and obtain a derivation of $X \gg Y$.

For the third case, where the head of the consequent is an implication, we first note that the antecedent is included in the larger antecedent $p \# X$ and that the consequent is included in the larger consequent $q \# Y$. From this observation, we can instantiate the induction hypothesis such that it fits rule *Imp_R* to obtain a derivation of $X \gg (p \to q) \# Y$. Finally, since $p \to q$ is already in Y, we can use *Set_R* to deduplicate Y and obtain a derivation of $X \gg Y$.

The cases for atomic propositions are trivial since they just move the atomic propositions from one part of the larger sequent to another. This concludes the proof of lemma *MP*.

We now prove our main theorem *OK*, which states that a derivation of $X \gg Y$ can be constructed if and only if X and Y form a valid sequent. We prove each direction individually. If we know that X and Y form a valid sequent, we can use the fact that *mp* returns true for valid sequents (lemma *main*) and lemma *MP* to obtain a derivation of $X \gg Y$. For the other direction, where we have a derivation of $X \gg Y$, we proceed by induction on the structure of this derivation. Each case follows immediately from the induction hypothesis.

To round off, we specialize theorem *OK* to single formulas by simply instantiating the theorem with an empty antecedent and a single-formula consequent and applying the definitions of validity for sequents and formulas.

4 Examples

We now know that our sequent calculus is an appropriate proof system for validity of formulas, so we can try to prove some examples ourselves instead of using the prover.

```
proposition ‹[] ≫ [p → p]›
proof -
  from Imp_R have ?thesis if ‹[p] ≫ [p]›
    using that by force
  with Basic show ?thesis
    by force
qed

proposition ‹[] ≫ [p → (p → q) → q]›
proof -
  from Imp_R have ?thesis if ‹[p] ≫ [(p → q) → q]›
    using that by force
  with Imp_R have ?thesis if ‹[p → q, p] ≫ [q]›
    using that by force
  with Imp_L have ?thesis if ‹[p] ≫ [p, q]› and ‹[q, p] ≫ [q]›
    using that by force
  with Basic show ?thesis
    by force
qed

proposition ‹[] ≫ [p → q → q → p]›
proof -
  from Imp_R have ?thesis if ‹[p] ≫ [q → q → p]›
    using that by force
  with Imp_R have ?thesis if ‹[q, p] ≫ [q → p]›
    using that by force
  with Imp_R have ?thesis if ‹[q, q, p] ≫ [p]›
    using that by force
  with Set_L have ?thesis if ‹[p, q] ≫ [p]›
    using that by force
  with Basic show ?thesis
    by force
qed
```

Fig. 3. Examples of "manual" proofs using the sequent calculus.

Our first example is the formula $p \to p$. To prove the validity of the formula, we place it alone in the consequent of a sequent with an empty antecedent. Since we have an implication in the consequent, we apply rule *Imp_R* to split the implication, moving the left side of the implication into the antecedent. This means that formula p is now the head of both parts of the sequent, so we can apply rule *Basic* to finish the proof that $p \to p$ is a valid formula.

Our second example is the formula $p \to (p \to q) \to p$. We first apply rule *Imp_R* twice to split the two outermost implications, thus moving p and $p \to q$ into the antecedent and leaving q in the consequent. Since we now have an implication in the head of the antecedent, we apply rule *Imp_L* to split it, thus splitting the proof in two branches. Both branches now have the same formula at the head of both their antecedent and their consequent, and we can apply rule *Basic* in both branches to finish the proof.

Our final example is the formula $p \to q \to q \to p$. We first apply *Imp_R* thrice to split the two outermost implications, thus moving p and two q's into the antecedent and leaving p alone in the consequent. We now have p in both the antecedent and the consequent, but in the antecedent it is not at the head of the list. We thus use rule *Set_L* to move p to the head of the antecedent (and remove the duplicate q), then use *Basic* to finish the proof.