CoreCalc: A Core Sequent Calculus Tool

Jørgen Villadsen - Technical University of Denmark

We present a tool for teaching classical propositional logic using the sequent calculus.

The tool is a concise formalization in the Isabelle proof assistant.

The formalization shows that the proof system is sound and complete.

The tool has been used for several years as a gentle introduction to propositional logic and automated reasoning in courses for computer science students.

Thanks to Simon Tobias Lund for help with the formalization

Introduction – Intuitionistic Implicational Logic

```
datatype form = Pro string | Imp form form

notation Imp (infixr ‹→› 0)

inductive Axiomatics (‹⊢›) where
  AK: ‹⊢ (p → q → p)› |
  AS: ‹⊢ ((p → q → r) → (p → q) → p → r)› |
  MP: ‹⊢ q› if ‹⊢ (p → q)› and ‹⊢ p›

theorem ‹⊢ (p → p)›
  using MP MP AS AK AK .

theorem ‹⊢ (p → p)›
  using AK AS MP by blast
```

```
(* Sequent Calculus – Simplified Isabelle Syntax *)

datatype form =
  Pro nat (·)

function mp where
  mp A B [] [] = (set A ∩ set B ≠ {})
  mp A B (·n # C) [] = mp (n # A) B C []
  mp A B C (·n # D) = mp A (n # B) C D

definition prover (⊢) where
  ⊢ p  =  mp [] [] [] [p]

primrec semantics (infix ⊨ 50) where
  I ⊨ ·n  =  I n

inductive SC (infix ➤ 50) where
  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 # _

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

theorem
  (∀I. I ⊨ p)  ⟷  ⊢ p
  (∀I. I ⊨ p)  ⟷  [] ➤ [p]        (∀I. ⟦I⟧ X Y)  ⟷  X ➤ Y
```

```
(* Sequent Calculus – Simplified Isabelle Syntax *)

datatype form =
  Pro nat (·)
  Neg form (~)

function mp where
  mp A B (~ p # C) [] = mp A B C [p]
  mp A B C (~ p # D) = mp A B (p # C) D

  mp A B [] [] = (set A ∩ set B ≠ {})
  mp A B (·n # C) [] = mp (n # A) B C []
  mp A B C (·n # D) = mp A (n # B) C D

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

primrec semantics (infix ⊨ 50) where
  I ⊨ ·n  =  I n
  I ⊨ ~ p  =  (¬ I ⊨ p)

inductive SC (infix ⩾ 50) where
  Neg_L:  ~ p # X ⩾ Y  if  X ⩾ p # Y
  Neg_R:  X ⩾ ~ p # Y  if  p # X ⩾ 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 # _

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

theorem
  (∀I. I ⊨ p)  ⟷  ⊢ p
  (∀I. I ⊨ p)  ⟷  [] ⩾ [p]        (∀I. ⟦I⟧ X Y)  ⟷  X ⩾ Y
```

```
(* Sequent Calculus – Simplified Isabelle Syntax *)

datatype form =
  Pro nat (·)

  Imp form form (infixr → 100)

function mp where

  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 [] [] = (set A ∩ set B ≠ {})
  mp A B (·n # C) [] = mp (n # A) B C []
  mp A B C (·n # D) = mp A (n # B) C D

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

primrec semantics (infix ⊨ 50) where
  I ⊨ ·n  =  I n

  I ⊨ p → q  =  (I ⊨ p ⟶ 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 # _

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

theorem
  (∀I. I ⊨ p)  ⟷  ⊢ p
  (∀I. I ⊨ p)  ⟷  [] ➤ [p]        (∀I. ⟦I⟧ X Y)  ⟷  X ➤ Y
```

```
(* Sequent Calculus – Simplified Isabelle Syntax *)

datatype form =
  Pro nat (·)
  Neg form (∼)
  Imp form form (infixr → 100)

function mp where
  mp A B (∼ p # C) [] = mp A B C [p]
  mp A B C (∼ p # D) = mp A B (p # C) D
  mp A B (p → q # C) [] = (mp A B C [p] ∧ mp A B (q # C) [])
  mp A B C (p → q # D) = mp A B (p # C) (q # D)
  mp A B [] [] = (set A ∩ set B ≠ {})
  mp A B (·n # C) [] = mp (n # A) B C []
  mp A B C (·n # D) = mp A (n # B) C D

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

theorem
  p ∨ q  ⟷  ¬ p ⟶ q
  p ∧ q  ⟷  ¬ (p ⟶ ¬ q)

primrec semantics (infix ⊨ 50) where
  I ⊨ ·n  =  I n
  I ⊨ ∼ p  =  (¬ I ⊨ p)
  I ⊨ p → q  =  (I ⊨ p ⟶ I ⊨ q)

inductive SC (infix ⩾ 50) where
  Neg_L:  ∼ p # X ⩾ Y  if  X ⩾ p # Y
  Neg_R:  X ⩾ ∼ p # Y  if  p # X ⩾ Y
  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 # _

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

theorem
  (∀I. I ⊨ p)  ⟷  ⊢ p
  (∀I. I ⊨ p)  ⟷  [] ⩾ [p]        (∀I. ⟦I⟧ X Y)  ⟷  X ⩾ Y
```

Reference

On Verified Automated Reasoning in Propositional Logic: Teaching Sequent Calculus to Computer Science Students

Simon Tobias Lund and Jørgen Villadsen

Vietnam Journal of Computer Science

Vol. 11, No. 4 (2024) 621–644

DOI: 10.1142/S2196888824500064

Conclusion – Summary & Future Work

As the complexity of software systems is ever increasing, so is the need for practical tools for formal verification.

Among these are automatic theorem provers, capable of solving various reasoning problems automatically, and proof assistants, capable of deriving more complex results when guided by a mathematician/programmer.

In this paper we consider using the latter to build the former.

In the proof assistant Isabelle/HOL we combine functional programming and logical program verification to build a theorem prover for propositional logic.

We also consider how such a prover can be used to solve a reasoning task without much mental labor.

The development is extended with a formalized proof system for writing machine-checked sequent calculus proofs.

We consider how this can be used to teach computer science students about logic, automated reasoning and proof assistants.