Logic and Computation

to explain the principles and limits of mechanical theorem proving

Dr. Wei Chen, Huawei Cambridge Research Centre

March 2024, Cambridge, England

Logic and Computation

Propositional Logic *is decidable*:

- concepts: formulae, semantics, consequence, deduction, soundness, completeness
- systems: natural deduction, satisfiability, equivalence checking, model checking

Predicate Logic *is undecidable*:

- concepts: variables, functions, relations, quantification, substitution, unification
- systems: natural, resolution, decidable, simplex, reduction, normalisation

Intuitionistic Logic *believes in "propositions as types and programs as proofs"*:

- concepts: lambda calculus, types, lambda cube
- systems: type inference, calculus of constructions

A. Propositional Logic *is decidable*

There are confluent procedures to demonstrate automatically a propositional statement.

A1. Concepts

Formulae and Semantics

Syntax or Formulae

$$\psi \ ::= \ p \ \mid \ \diamond \psi \ \mid \ \psi \ \bullet \ \psi$$

Semantics or Interpretation

$$\mathbb{B} \ = \{ \, t, \ f \, \} \qquad [\![\, p \,]\!] : \mathbb{B} \qquad [\![\diamond]\!] : \mathbb{B} \to \mathbb{B} \qquad [\![\bullet]\!] : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \qquad [\![\, \psi \,]\!] : \mathbb{B}^n \to \mathbb{B}$$

Implication *as an example*

$$[\![\Rightarrow]\!] \, (t, f) = f$$
$$[\![\Rightarrow]\!] \, (\_, \_) = t$$

Logical Consequence

Consequence

$$\phi \models \psi$$

If $[\![\phi]\!] (\vec{x}) = t$, then $[\![\psi]\!] (\vec{x}) = t$, for all assignments $\vec{x} \in \mathbb{B}^n$.

That is, the formula $\phi \Rightarrow \psi$ is a tautology.

Example

$$p \wedge q \models p$$

$$[\![p \wedge q]\!] (t, t) = t \qquad [\![p]\!] (t, \_) = t$$
$$[\![p \wedge q]\!] (\_, \_) = f \qquad [\![p]\!] (f, \_) = f$$

Sequent and Deduction

Sequent

$$A_1, A_2, \cdots, A_n \vdash C_1, C_2, \cdots, C_m$$

The formula $A_1 \wedge A_2 \wedge \cdots \wedge A_n \Rightarrow C_1 \vee C_2 \vee \cdots \vee C_m$ is a tautology.

Deductive Proof

A list of sequents satisfying that each sequent is either an axiom or derived from previous sequents by applying inference rules.

Soundness and Completeness

Soundness and Completeness

A deduction system consists of a finite set of axioms and inference rules.

It is sound if $\phi \vdash \psi$ implies $\phi \models \psi$ for all fomulae ϕ and ψ.

It is complete if $\phi \models \psi$ implies $\phi \vdash \psi$ for all fomulae ϕ and ψ.

A2. Natural Deduction

In logic and proof theory, natural deduction is a kind of proof calculus in which logical reasoning is expressed by inference rules closedly related to "natural" way of reasoning. — Wikipedia

Gentzen System

GENTZEN SYSTEM *as an example of natural deduction*

$$\frac{}{\Gamma,\ \psi\ \vdash\ \Delta,\ \psi}\ \text{INITIAL} \qquad \frac{\Gamma\ \vdash\ \Delta,\ \psi}{\Gamma,\ \neg\psi\ \vdash\ \Delta}\ \text{NOTL} \qquad \frac{\Gamma,\ \psi\ \vdash\ \Delta}{\Gamma\ \vdash\ \Delta,\ \neg\psi}\ \text{NOTR}$$

$$\frac{\Gamma\ \vdash\ \Delta,\ \phi \qquad \Gamma,\ \psi\ \vdash\ \Delta}{\Gamma,\ \phi \Rightarrow \psi\ \vdash\ \Delta}\ \text{IMPL} \qquad \frac{\Gamma,\ \phi\ \vdash\ \Delta,\ \psi}{\Gamma\ \vdash\ \Delta,\ \phi \Rightarrow \psi}\ \text{IMPR}$$

$$\frac{\Gamma,\ \phi,\ \psi\ \vdash\ \Delta}{\Gamma,\ \phi \wedge \psi\ \vdash\ \Delta}\ \text{CONL} \qquad \frac{\Gamma\ \vdash\ \Delta,\ \phi \qquad \Gamma\ \vdash\ \Delta,\ \psi}{\Gamma\ \vdash\ \Delta,\ \phi \wedge \psi}\ \text{CONR}$$

$$\frac{\Gamma,\ \phi\ \vdash\ \Delta \qquad \Gamma,\ \psi\ \vdash\ \Delta}{\Gamma,\ \phi \vee \psi\ \vdash\ \Delta}\ \text{DISL} \qquad \frac{\Gamma\ \vdash\ \Delta,\ \phi,\ \psi}{\Gamma\ \vdash\ \Delta,\ \phi \vee \psi}\ \text{DISR}$$

Terms

Let us implement Gentzen system in functional programming language Haskell.

λ-TERM $\quad t ::= x \mid t\ t \mid \lambda x\ .\ t$

DEFINE DATA TYPE `Term` *to express formulae*

```
data Term = Cst String | Var String
          | App [Term] | Abs [Term] deriving (Eq, Ord)
```

Terms

DEFINE show FUNCTION *to convert terms into strings*

```
instance Show Term where
  show (Cst s) = s
  show (Var x) = x
  show (App ts) = "(" ++ shows ts ++ ")"
  show (Abs ts) = "(abs " ++ shows ts ++ ")"

shows :: Show a => [a] -> String
shows xs = if null xs then ""
           else foldl (\y x -> y ++ " " ++ show x)
                      (show $ head xs) (tail xs)
```

Trees

DEFINE AUXILIARY DATA TYPE `Tree` *to characterise the syntax structure of terms*

```
data Tree = Tree String [Tree] deriving (Eq, Ord)

instance Show Tree where
  show = concat . showt
    where showt :: Tree -> [String]
          showt (Tree s []) = [s ++ "\n"]
          showt (Tree s ts) =
            let x = s ++ " ->\n"
                d = take (length x) (repeat ' ')
            in x : (concat $ map (\t -> map (d++) (showt t)) ts)
```

Trees

DEFINE AUXILIARY FUNCTION `token` *to split strings into tokens*

```
token :: String -> [String]
token s = let (a, t) = head (lex s)
          in if a == "" then []
             else if t == "" then [a]
                  else a : (token t)
```

Trees

DEFINE AUXILIARY FUNCTION `nest` *to replace the top-level paired-braces by brackets*

```
nest :: [String] -> [[String]]
nest xs = let (ys, xss) = rec xs 0
          in wrap ys ++ xss
  where rec :: [String] -> Int -> ([String], [[String]])
        rec [] n = ([], [])
        rec ("(":xs) n = let (ys, xss) = rec xs (n + 1)
                         in if n == 0 then ([], ys:xss)
                            else ("(":ys, xss)
        rec (")":xs) n = let (ys, xss) = rec xs (n - 1)
                         in if n == 1 then ([], wrap ys ++ xss)
                            else (")":ys, xss)
        rec (x:xs) n = let (ys, xss) = rec xs n in (x:ys, xss)
        wrap = map (\x -> [x])
```

Trees

DEFINE `tree` FUNCTION *to construct trees from strings*

```
tree :: String -> Tree
tree = mkt . map skt . nest . token
  where skt :: [String] -> Tree
        skt [] = Tree "" []
        skt [x] = Tree x []
        skt xs = let x = head xs
                 in if x == "app" || x == "abs"
                    then Tree x (map skt $ nest $ tail xs)
                    else skt ("app" : xs)
        mkt :: [Tree] -> Tree
        mkt [] = Tree "" []
        mkt [t] = t
        mkt ts = Tree "app" ts
```

Terms

DEFINE `readt` FUNCTION *to construct terms from strings via trees*

```
isvar :: String -> Bool
isvar s = s /= "" && isLower (head s)

iscst :: String -> Bool
iscst = not . isvar

readt :: String -> Term
readt = mkt . tree
  where mkt :: Tree -> Term
        mkt t = case t of
                  Tree s ts ->
                    case s of
                    "abs" -> Abs $ map mkt ts
                    "app" -> App $ map mkt ts
                    _ -> if isvar s then Var s else Cst s
```

Goals

Sequent $\Gamma \vdash \Delta$

Define data type `Goal` *to realise sequents*

```
data Goal = Goal [Term] [Term] deriving (Eq, Ord)
```

Goals

DEFINE show FUNCTION *to convert goals into strings*

```
instance Show Goal where
  show (Goal xs ys) =
    let hs = showi "A" xs
        ks = showi "C" ys
        m = maxs $ map length (hs ++ ks)
    in if m == 0 then ""
       else let s = take m (repeat '-')
            in "\n" ++ concat hs ++ s ++ "\n" ++ concat ks
    where maxs :: [Int] -> Int
          maxs = foldl (\y x -> if y >= x then y else x) 0

showi :: Show a => String -> [a] -> [String]
showi s xs = map (\(n,x) -> s ++ show n ++ ": " ++ show x ++ "\n")
                 (zip [1..length xs] xs)
```

Axioms

$$\frac{}{\Gamma,\ \psi\ \vdash\ \Delta,\ \psi}\ \textsc{initial}$$

DEFINE `initial` FUNCTION *to realise the axiom* INITIAL

```
cap :: Eq a => [a] -> [a] -> [a]
cap xs = foldl (\zs y -> if y `elem` xs then zs ++ [y] else zs) []

initial :: Goal -> [Goal]
initial (Goal xs ys) = if null $ cap xs ys
                       then [Goal xs ys] else []
```

Inference Rules

$$\frac{\Gamma \vdash \Delta, \psi}{\Gamma, \neg\psi \vdash \Delta}\ \text{NOTL}$$

DEFINE `notl` FUNCTION *to realise the inference rule* NOTL

```
notl :: Goal -> [Goal]
notl (Goal xs ys) =
  let p x = case x of
            App [Cst "~", t] -> False
            _ -> True
      f x = case x of
            App [Cst "~", t] -> [t]
            _ -> []
      hs = filter p xs
      ks = concat $ map f xs
   in [Goal hs (ys ++ ks)]
```

Inference Rules

$$\frac{\Gamma,\ \psi\ \vdash\ \Delta}{\Gamma\ \vdash\ \Delta,\ \neg\psi}\ \textsc{notr}$$

DEFINE `notr` FUNCTION *to realise the inference rule* NOTR

```
notr :: Goal -> [Goal]
notr (Goal xs ys) =
  let p x = case x of
            App [Cst "~", t] -> False
            _ -> True
      f x = case x of
            App [Cst "~", t] -> [t]
            _ -> []
      hs = filter p ys
      ks = concat $ map f ys
  in [Goal (xs ++ ks) hs]
```

Inference Rules

$$\frac{\Gamma \vdash \Delta, \phi \qquad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \Rightarrow \psi \vdash \Delta}\ \text{IMPL}$$

DEFINE `impl` FUNCTION *to realise the inference rule* IMPL

```
impl :: Goal -> [Goal]
impl (Goal xs ys) =
  let f x = case x of
              App [Cst "=>", ta, tb] -> [(x, ta, tb)]
              _ -> []
      ps = concat $ map f xs
  in if null ps then [Goal xs ys]
     else let (t, ta, tb) = head ps
              hs = rm t xs
          in [Goal (tb:hs) ys, Goal hs (ta:ys)]
```

Inference Rules

$$\frac{\Gamma,\ \phi\ \vdash\ \Delta,\ \psi}{\Gamma\ \vdash\ \Delta,\ \phi \Rightarrow \psi}\ \text{IMPR}$$

DEFINE `impr` FUNCTION *to realise the inference rule* IMPR

```
impr :: Goal -> [Goal]
impr (Goal xs ys) =
  let p x = case x of
            App [Cst "=>", ta, tb] -> False
            _ -> True
      f x = case x of
            App [Cst "=>", ta, tb] -> [(ta, tb)]
            _ -> []
      (hs, ks) = unzip $ concat $ map f ys
  in [Goal (xs ++ hs) (filter p ys ++ ks)]
```

Inference Rules

$$\frac{\Gamma,\ \phi,\ \psi\ \vdash\ \Delta}{\Gamma,\ \phi \wedge \psi\ \vdash\ \Delta}\ \textsc{conl}$$

DEFINE `conl` FUNCTION *to realise the inference rule* CONL

```
conl :: Goal -> [Goal]
conl (Goal xs ys) =
  let f x = case x of
              App [Cst "/\\", ta, tb] -> [ta, tb]
              _ -> [x]
  in [Goal (concat $ map f xs) ys]
```

Inference Rules

$$\frac{\Gamma \vdash \Delta, \phi \qquad \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \phi \wedge \psi}\ \text{CONR}$$

DEFINE `conr` FUNCTION *to realise the inference rule* CONR

```
conr :: Goal -> [Goal]
conr (Goal xs ys) =
  let f x = case x of
              App [Cst "/\\", ta, tb] -> [(x, ta, tb)]
              _ -> []
      ps = concat $ map f ys
  in if null ps then [Goal xs ys]
     else let (t, ta, tb) = head ps
              hs = rm t ys
          in [Goal xs (ta : hs), Goal xs (tb : hs)]
```

Inference Rules

$$\frac{\Gamma,\ \phi \vdash \Delta \qquad \Gamma,\ \psi \vdash \Delta}{\Gamma,\ \phi \vee \psi \vdash \Delta}\ \text{DISL}$$

DEFINE `disl` FUNCTION *to realise the inference rule* DISL

```
disl :: Goal -> [Goal]
disl (Goal xs ys) =
  let f x = case x of
              App [Cst "\\/", ta, tb] -> [(x, ta, tb)]
              _ -> []
      ps = concat $ map f xs
  in if null ps then [Goal xs ys]
     else let (t, ta, tb) = head ps
              hs = rm t xs
          in [Goal (ta : hs) ys, Goal (tb : hs) ys]
```

Inference Rules

$$\frac{\Gamma \vdash \Delta, \phi, \psi}{\Gamma \vdash \Delta, \phi \vee \psi}\ \text{DISR}$$

DEFINE `disr` FUNCTION *to realise the inference rule* DISR

```
disr :: Goal -> [Goal]
disr (Goal xs ys) =
  let f x = case x of
              App [Cst "\\/", ta, tb] -> [ta, tb]
              _ -> [x]
  in [Goal xs (concat $ map f ys)]
```

Extension: Reflexivity

$$\frac{}{\Gamma \vdash \Delta,\ t = t}\ \text{REFL}$$

DEFINE `refl` FUNCTION *to realise the axiom* REFL

```
refl :: Goal -> [Goal]
refl (Goal xs ys) =
  let p y = case y of
            App [Cst "=", ta, tb] -> ta == tb
            _ -> False
  in if null $ filter p ys then [Goal xs ys] else []
```

Extension: Logical Equivalence

$$\frac{\Gamma,\ \phi \Rightarrow \psi,\ \psi \Rightarrow \phi\ \vdash\ \Delta}{\Gamma,\ \phi \Leftrightarrow \psi\ \vdash\ \Delta}\ \textsc{biml}$$

DEFINE `biml` FUNCTION *to realise the axiom* BIML

```
biml :: Goal -> [Goal]
biml (Goal xs ys) =
  let p x = case x of
            App [Cst "<=>", ta, tb] -> False
            _ -> True
      f x = case x of
            App [Cst "<=>", ta, tb] ->
              [(App [Cst "=>", ta, tb], App [Cst "=>", tb, ta])]
            _ -> []
      (hs, ks) = unzip $ concat $ map f xs
  in [Goal (filter p xs ++ hs ++ ks) ys]
```

Extension: Logical Equivalence

$$\frac{\Gamma \vdash \Delta,\ \phi \Rightarrow \psi \qquad \Gamma \vdash \Delta,\ \psi \Rightarrow \phi}{\Gamma \vdash \Delta,\ \phi \Leftrightarrow \psi}\ \text{BIMR}$$

DEFINE `bimr` FUNCTION *to realise the axiom* BIMR

```
bimr :: Goal -> [Goal]
bimr (Goal xs ys) =
   let f x = case x of
               App [Cst "<=>", ta, tb] ->
                [(x, App [Cst "=>", ta, tb], App [Cst "=>", tb, ta])]
               _ -> []
       ps = concat $ map f ys
   in if null ps then [Goal xs ys]
      else let (t, ta, tb) = head ps
               hs = rm t ys
           in [Goal xs (ta:hs), Goal xs (tb:hs)]
```

Tactics

TACTIC: a command to transform a list Γ of goals into another list Δ of goals such that the conjunction of goals in Δ implies the conjunction of goals in Γ.

```
type Tacs = [(String, [Goal] -> [Goal])]
```

Tactics

```
axms :: Tacs
axms = [("initial", initial . head),
        ("refl", refl . head)]

infs :: Tacs
infs = [("notl", notl . head),
        ("notr", notr . head),
        ("impl", impl . head),
        ("impr", impr . head),
        ("conl", conl . head),
        ("conr", conr . head),
        ("disl", disl . head),
        ("disr", disr . head),
        ("biml", biml . head),
        ("bimr", bimr . head)]
```

Interactive Theorem Provers

DEFINE `itp` FUNCTION *to allow users to prove a statement by applying tactics*

```
itp :: Tacs -> [Goal] -> IO ()
itp tacs [] = putStrLn "\n  Q. E. D. \n"
itp tacs (g:gs) =
  do putStr "\n# CURRENT GOAL "
     putStrLn ("(" ++ show (length gs) ++ " OTHER GOALS TO PROVE) #")
     putStr (show g)
     putStr "\n > "
     s <- getLine
     case s of
       "quit" -> putStrLn "\n  BYE-BYE  \n"
       "next" -> itp tacs (gs ++ [g])
       _ -> let hs = concat $ map (\(t,f) -> if s == t then [f] else []) tacs
            in if null hs
               then do putStrLn "\n!!! INVALID !!!\n"
                       itp tacs (g:gs)
               else let ks = head hs $ [g]
                    in if ks == [g]
                       then do putStrLn "\n!!! NO EFFECT !!!\n"
                               itp tacs (ks ++ gs)
                       else itp tacs (ks ++ gs)
```

Interactive Theorem Provers

DEMO: *to demonstrate that*

$$p \wedge q \Rightarrow p \vee q$$

```
$ ghci gentzen.hs
GHCi, version 9.6.1: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Main             ( gentzen.hs, interpreted )
Ok, one module loaded.
ghci> itp (axms ++ infs) [head gs]

# CURRENT GOAL (0 OTHER GOALS TO PROVE) #

--------------------------
C1: (=> (/\ p q) (\/ p q))

 > impr

# CURRENT GOAL (0 OTHER GOALS TO PROVE) #

A1: (/\ p q)
------------
C1: (\/ p q)

 > conl

# CURRENT GOAL (0 OTHER GOALS TO PROVE) #

A1: p
A2: q
------------
C1: (\/ p q)

 > disr

# CURRENT GOAL (0 OTHER GOALS TO PROVE) #

A1: p
A2: q
------
C1: p
C2: q

 > initial

  Q. E. D.
```

Automation: Simplification

CONFLUENCE: **the order of applying tactics in Gentzen system doesn't matter.**

In computer science, **confluence** is a property of rewriting systems, describing which terms in such a system can be rewritten in more than one way, to yield the same result. — Wikipedia

Automation: Simplification

DEFINE `simple` FUNCTION *to apply tactics until the proof is done or there is no change*

```
fix :: Eq a => (a -> a) -> a -> a
fix f x = let y = f x in if x == y then x else fix f y

simple :: [Goal] -> [Goal]
simple gs =
  let axms = [initial, refl]
      infs = [notl, notr, impl, impr, conl, conr,
              disl, disr, biml, bimr]
      tacs = axms ++ infs
  in fix (concat . map (apply tacs)) gs
  where apply :: [Goal -> [Goal]] -> Goal -> [Goal]
        apply [] g = [g]
        apply (t:ts) g = let gs = t g
                         in if gs == [g] then apply ts g else gs
```

Tactics

ADD `simple` TACTIC *to extend the list of tactics*

```
autos :: Tacs
autos = [("simple", simple)]
```

Automation: Simplification

DEMO: *to demonstrate that*

$$p \wedge q \Rightarrow p \vee q$$

$$\neg\neg p \Leftrightarrow p$$

$$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$$

$$p \vee \neg q$$

$$p \Rightarrow p \wedge q$$

```
ghci> itp (axms ++ infs ++ autos) gs

# CURRENT GOAL (4 OTHER GOALS TO PROVE) #

--------------------------
C1: (=> (/\ p q) (\/ p q))

 > simple

# CURRENT GOAL (3 OTHER GOALS TO PROVE) #

----------------------
C1: (<=> (~ (~ p)) p)

 > simple

# CURRENT GOAL (2 OTHER GOALS TO PROVE) #

---------------------------------------
C1: (<=> (~ (\/ p q)) (/\ (~ p) (~ q)))

 > simple

# CURRENT GOAL (1 OTHER GOALS TO PROVE) #

-----------------
C1: (\/ p (~ p))

 > simple

# CURRENT GOAL (0 OTHER GOALS TO PROVE) #

--------------------
C1: (=> p (/\ p q))

 > simple

# CURRENT GOAL (0 OTHER GOALS TO PROVE) #

A1: p
------
C1: q

 > quit

  BYE-BYE
```