Classical and Constructive Logic Interactive Theorem Prover

The project is designed as an educational tool to help introduce students to the logic realm. A theorem can be proved by clicking on formulas, the rule is directly applied according to the connective and a theorem is proven once there is no more goals to check. We took an imperative approach using OOP to implement the prover, instead of functional programming.

Implementation

There are three independent components to the prover.

Types

We have two main types: propositions defined by a state (T/F) and a name and formulas defined by a connective (Or/And/Imply/Not) and sub formulas, a list of formula objects

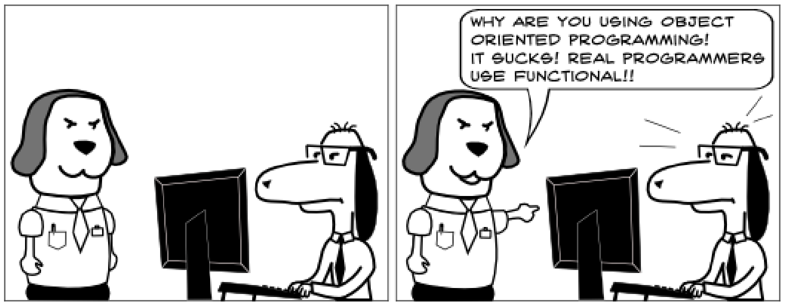
Logic

Collection of rules that take in a sequent and returns the new sequents after a rule is applied. It also contains axioms that given a sequent returns true or false if an axiom is found. Since the logic is independent, adapting the prover to other logics is simple.

Engine

Given a list of goals (sequents) to prove, the engine takes as input from the user which formula to break down and recursively continues the process until no more goals are left.

Functional vs. Imperative


WHY ARE YOU USING OBJECT ORIENTED PROGRAMMING! IT SUCKS! REAL PROGRAMMERS USE FUNCTIONAL!!

http://geekofficedog.blogspot.com

+

- Flexibility of types, namely objects in OOP; let us deal with tricky concepts like sub formulas which can be propositions.
- Each object created is unique, making comparisons straightforward –we do not look at the name. Hence, we face no problem in creating fresh variables.
- Modularity offered by imperative programming gives more room to improve and debug each component.

−

- Lack of efficient recursion makes it hard to implement inherently recursive rules like substitution.
- Error is more likely due to the lack of rigid type checking

OOP to introduce First-Order Logic

The following are changes made to types

Formulas	Quant	Connective	Sub formulas		
Props	Quant	Name	State	isVar	Input

Quant are a new class defined by a quantifier (forall/exists) and the variables applied to. Propositions are extended to cover variables and functions. If the input, a list of variables, is empty then the object is either a proposition or a variable; if isVar is true then it is a variable, else a proposition. The changes did not affect the other modules and the transition was seamless. The main challenge was implementing substitution without mapping.

Samar Rahmouni
Prof. Giselle Reis
https://github.com/natvern/CCLITP