Peirce's Alpha System

April 2021

1 Prerequisite

The project requires PySwip, SWI-PROLOG (64-bit) and Python 3 (64-bit) to run. PySwip can be installed with 'pip install pyswip'.

2 Quick Start

The user first enters the directory with the Peirce.py and peirce.pl file, then runs 'python Peirce.py'. The user then selects 'start' under 'Commands' menu, and inputs a formula using Coq syntax for connectives. Then clicks on 'Start Proof' to start proving.

3 User Interface

The GUI has a select feature. The user can select multiple graphs only in the same level, and deselect by clicking on selected graphs. For convenience, if the user wants to select all graphs in a given region, the user can select the empty space surrounding the graphs instead.
The user can initialise a graph by selecting the 'start' tab, and inputting a formula. The corresponding graph will be printed on the GUI. The connectives uses Coq's format. They are, $\tilde{}$, $/\backslash, \backslash/, ->$ and $<->$. True and false can be entered as just 'true' and 'false'. A letter represents an atomic proposition. The input does not differentiate capitalization, and all whitespaces will be removed. Thus, as an example, an input of 'T r Ue' will be parsed as the true value, and there is no difference between 'a' and 'A'. They will both be printed as 'A' on the graph. The input also accepts brackets to prioritise operations.
The user can cut or copy a selected graph from the command menu. The user can also paste cut/copied graphs into regions or cuts by selecting the region/cut and clicking 'paste' in the command menu.
There is also 'undo' and 'redo' in the command menu.
Once the user has the graph ready, the user can click on 'Start Proof' in the command menu. The command menu changes to a rules menu, containing the five inference rules, undo and redo for rules and 'stop proof' button.

To erase, the user can click on one or multiple graphs, or a region, then select erasure in the rules menu. If the selected graph has even depth, it will be erased.
To insert, the user can select either a cut or a region, then select insert rule in the rules menu. A text box to insert a formula appears, and the formula will be converted into a graph inside the selected region or inside the selected cut.
To iterate, the user must first click on the graph to iterate, then click on the iterate button. Then the user must select a region or a cut that is in a nested level of the first graph to iterate the first graph into.
To deiterate, the user first selects a graph, then selects deiterate. The user will then select the same graph that is in a nested level of the first, then click on 'Run Deiterate' button. The inner graph will then be removed.
To insert double cut, the user first selects a graph then clicks on 'Insert Double Cut' to add a double cut around the selected graph.
To remove a double cut, the user must select the outer cut of the double cut, then click on 'Remove Double Cut'.
The user can undo and redo the rules as well from the rules menu.
Once the user clicks on 'Stop Proof', the original graph at the start of the proof will be printed and the user can continue editing the original graph before the proof started.

4 Implementation

The project utilised python as the programming language, Tkinter for the GUI and PySwip with prolog for parsing and converting the input formula into conjunction and negation.
The code consists of a python file and a prolog file.
The python file contains 3 classes, Graph, Box and Atom. Both Box and Atom inherits from Graph. All of them has a list of children, which enables a tree structure, with an instantiated Graph as a root.
A Graph is represented by a rectangle with fill, Box is represented by a rectangle with no fill and just a border, and Atom is represented by a text object on the canvas.
This file also contains functions to print the graph, functions for the 5 inference rules, and editing commands such as select, cut, copy and paste.
Each graph (Graph, Box, Atom) has a top left and bottom right coordinate, to place canvas objects representing the graphs in the correct location. The function calculateCoord calculates the coordinates with DFS, the top left coordinates in pre-order manner, while the bottom right coordinates in a post-order manner. The top left coordinate takes reference of the parent graph's top left coordinate or, if its parent has multiple children, takes reference from the previous child's top left coordinate. It then adds a fixed value to the reference, and makes it it's own top left coordinate. The bottom right coordinate takes reference from the furthest right child, or if it has no children, it will reference its own top left coordinate. It will add a fixed value to the reference, and make

the new coordinate its own bottom right coordinate. A function printGraph, creates canvas objects for each graph in the tree. When creating canvas objects, each object is given an id, which is stored in each graph in the tree.
Select works by obtaining the coordinates of the click on the canvas, and finding canvas objects that are at the clicked coordinate. Since each canvas object has an id, the id of the graph can be obtained from select.
The find method conducts a DFS by finding a graph until the input id is the id of the current graph, and the graph is returned.
A function for checking if a graph is nested in another and checking if two graphs are equivalent are also added. They are used to check for the nesting and equivalence in iteration and deiteration.
A function that returns a list of the parents of a graph is made. It does a DFS traversal until the id of the graph is found, and returns the list of parents found. It is used for checking number of cuts around a given graph for erasure and insertion.
A prolog file is also given, which contains a DCG for parsing the user's input formula into compounds and(X, Y) or neg(X). The python file utilises PySwip to consult the prolog file, and provides the DCG with the user's input. The python file then receives the parse tree made of the compounds 'and' and 'neg', and uses the compound to form the graph. The graph will then be the graph on the GUI.
The prolog DCG ensures right-associativity for implication and left-associativity for conjunction, disjunction and equivalence. The operators are also prioritised, with negation having the highest, followed by conjunction and disjunction, then implication and finally equivalence. Content inside brackets are prioritised before operators. Tabling is also used to enable left recursion for left-associativity.
The prolog file also contains a goal for splitting and(X,Y) into X and Y, which is used when converting the DCG's parse tree into the graph in python.