The Application of Computing to Mathematics and Induction Code

Ben Stefl
MAT227 - Marianne Smith

Introduction to Proof Assistants

What is a proof?

A proof is the process of validating a statement according to principles of reasoning and using other known facts.

- The two roles of a proof:
  - Convince the reader of the proof
  - Explain why the proof is correct
- The three stages of a proof:
  - **Finding a proof:** not recorded, involves experimentation
  - **Recording:** what works is recorded, saying why the conclusion holds.
  - **Presentation:** the proof may be proofread throughout the process and is communicated to others.

What is a proof?

A proof can be broken down into many very small steps that are irrefutably true in order to verify its correctness. The irreducible steps are so small that no mathematician ever does this, but with any valid proof found in a book it is able to be done. Mathematicians combine the small steps to form an abstraction that shows what the proof means. The irreducible steps are too small for a human to understand, but this can be managed by a computer, and is where proof assistants are used.

Checking Proofs with Computers

- A computer serves to check the small irreducible components of an abstracted proof as mentioned previously.
  - The “kernel” of a proof checker is it’s library of irreducible steps that can be used for proofs.
  - The first proof checkers had limited syntax and only served to verify the given proof.
- Proof checkers are only useful with the persuasion and recording of proofs.
- Each PA uniquely overcomes the challenge of establishing a language that allows the user to communicate a proof.
- The use of these programs can extend to checking if other programs or systems meet their specifications.

Checking Proofs with Computers

- **These systems usually improve with:**
  - Greater levels of abstraction to help humans understand the systems
  - Having a smaller number of irreducible steps, or a smaller kernel, which allows the system to be more easily verified.
    - You need to be certain that the program is written correctly to believe in a proof checked with it.
    - De Bruijn Criterion: A principle that guides logical software, a proof assistant satisfies this criterion if it's verification program can be checked manually. The parts of the software that generate and check the proofs are independent.
- **Important Features of Proof Assistants:**
  - All valid proof tactics work in the logical system established.
  - The user can create "proof objects" within the system and independently verify them.

2 Styles of Writing PA Code

- **Procedural**
  - Harder to interpret by humans, needs to be executed to mean anything
  - Easier/shorter to write
  - Minimal information
- **Declarative**
  - Easier to interpret by humans, can be reasonably parsed by a mathematician without executing.
  - Harder/longer to write
  - Easier to adapt and less sensitive to changes.

History of Proof Assistants

- Interest in computer automated proof systems began in the 1950s, with the interest of truly automated theorem proving.
- The goal later shifted to formalization and automated proof checking in the 1970s.
- **Automath**
  - Usually considered the first proof assistant.
  - Initiated by De Bruijn in 1967 as an attempt to formalize/unify all areas of math through a unified mathematical language.
  - The idea is that correctly written code in Automath is always mathematically correct, allowing math to be check-able by a computer.
  - Several versions of Automath have been written, and they have had great influence on later PAs, although it has a simpler kernel compared to modern PAs.
  - De Brujin implemented only a basic logical framework within Automath that included capabilities for variables, substitution, definitions, and their creation by the user.
    - The Mathematical concept of **Type theory** is foundational to this logical framework.

History of Proof Assistants

- **NuPrl**
  - Martin–löf developed constructive logic into constructive type theory, which is an alternative foundation of mathematics. The first version of this type theory was implemented into the NuPrl proof assistant.
  - NuPrl was developed by a team at Cornell University formed in 1984. It aimed to be a system where programs are created “in a mathematical fashion by interactive refinement”.
- **Coq**
  - The Coq proof assistant uses a paradigm saying that any proof can be converted into a functional program that uses abstract data types.
  - Started in 1984 by Huet and Coquand at INRIA but has been worked on by many different researchers over the years.
  - Its Original intent was to serve a checker for a certain variation upon type theory called “Calculus of Constructions”.
  - Proofs are implemented as tactics operating on goals that represent the proof, it also operates like a functional programing language in it’s own right.
  - Coq was used to formalize the four-color theorem.

History of Proof Assistants

- **Agda**
  - A functional type checker for Martin–löf's type theory was developed in 1990 by Catarina Coquand and Nordstrom, this later evolved into Agda which was written by Coquand in 1996.
  - Agda has been implemented into other languages such as C and Haskell.
- **HOL**
  - Derived from a previous language designed for the formal verification of hardware systems at Cambridge.
  - Based on simply typed lambda calculus.
- **Isabelle**
  - Developed by Lawrence Paulson in 1990.
  - Has a framework for embedding specific logic from the user (called meta-logic), without induction as to limit assumptions, which simplifies the implementation of logic.
  - Has tactics and goals similar to Coq.
  - In the HOL family of languages.

History of Proof Assistants

- **Mizar**
  - Developed by Trybulec since 1973 at the university of Białystok, it is the oldest continuously maintained PA project.
  - It began with the goal of editing and recording a mathematical library rather than proof checking.
    - The Mizar Mathematical Library is the largest repository of formalized mathematics.
    - Users can submit new articles to its Library Committee to include.
  - The Mizar project consists of it's own variation upon standard mathematical language and a program for verifying the correctness of that language.
  - It is easy for a human to read proofs in Mizar as an actual proof with the tradeoff of a large Kernel. It does not satisfy De Bruijn criterion.
  - Does "batch proof checking" where an entire file can be verified at once and the system does not stop checking if an error is detected.

History of Proof Assistants

- **Nqthm**
  - First version in 1973, Eventually evolved into ACL2.
  - Powerful automation features with minor tradeoffs.
  - The user can add “lemmas” which are intermediate proofs that allow the user to break up a proof into stepping stones however they see fit.
    - Lemmas are proven by the system first before moving on to the proof that they are used in.
  - Popular within computer science for verification studies.

History of Proof Assistants

- **PVS (Prototype Verification Systems)**
  - developed at SRI International since 1992.
  - Attempts to combine advantages of expressive language and automation. This makes PVS particularly applicable and easy to use.
  - The downside is that PVS's kernel is large and violates De Brujin criterion, causing occasional bugs needing to be repaired that indicate inconsistencies.
- **EA (Evidence algorithm)**
  - Developed by the Russian mathematician Glushkov starting in the early 1970s with the goal of formalizing mathematical texts, similar in spirit to Mizar.
  - Evolved into the SAD (System for Automated Deduction) PA which checks proofs written in a language called ForTheL (FORmal THEory Language), also serving a similar purpose of documentation like Mizar but had developed independently.

Problems Only Solved by Computers

The 4 Color Theorem

- Says that every planar graph allows for a proper vertex coloring with 4 colors. Meaning that there is always a way to color in any 2D “map” of contiguous regions on a plane with no more than 4 colors, and not have any two regions of the same color touching (sharing a boundary). Regions of the same color may touch on a vertex.
  - Any 2D map can be related to a planar graph, planar means that the graph can be drawn without any edges (lines) crossing.
  - The problem was first posed in 1852 by Francis Guthrie.
  - First proof was presented by Alfred Kempe in 1879, A mistake in this proof was found by Percy Heawood in 1890. Heawood modified and fixed Kempe’s proof to show that no more than 5 colors were required for any given graph.

Heawood's Contradiction Proof (5 Color Theorem)

- Start by assuming that there is a graph *G* of minimal size without a proper coloring with no more than 5 colors. "minimal size" means that there is only one node *v* that must be removed in order to allow the graph to have a proper coloring with 5 colors.
- Consider Euler's formula: $V - E + F = 2$ (holds true for any planar graph)
  - *V*: # of vertices (or nodes)
  - *E*: # of edges (lines that connect 2 nodes)
  - *F*: # of regions (called faces) into which the plane is divided by any drawing of the graph. The outside region is counted.

Heawood's Contradiction Proof (5 Color Theorem)

- It can be deduced from Euler's formula that *G* contains a vertex with at most 5 neighbors.
- If *v* is removed from the graph, there is a proper 5 color coloring of what is left (since *G* is of minimal size).
- If *v* had less than 5 neighbors, *G* would have a proper coloring since *v* could have a different color than each of its neighbors.
- You can always properly color *v* and all of its neighbors by not using at least one color for any of *v*'s neighbors. If you couldn't, than the graph wouldn't be planar.

The 4 Color Theorem

- Heinrich Heesch later proposed the method for the 4 color proof where computers were needed. The idea is that every planar graph can only be composed of various configurations out of a finitely sized list *C* of possible configurations. It was hypothesized that there is no possible way to construct a graph out of these configurations that results in more than more than 4 colors being needed.
- Heawood's solution was thought to have a simple list of 5 configurations, depending on the number of connections a node has.
- Heesch proposed that a much larger list of configurations was needed.

The 4 Color Theorem

- **In 1976, Kenneth Appel and Wolfgang Haken found the list of configurations.**
  - The problem was that contained around 1200 configurations. And for some configurations, hundreds of thousands of cases needed to be checked.
  - This was the first major theorem that was too long to be humanly checkable. The use of computing was unsurprisingly controversial, many mathematicians felt uneasy about trusting that the program was written correctly.
  - This proof did have flaws, but all were corrected by Appel and Haken by 1989.
- In 1997, Robertson, Sanders, Seymour, and Thomas developed another proof with similar principles that was more transparent. The program used was improved and corrected with their proof. This removed any doubts left.
- Although a satisfactory proof had been found, the reason the proof holds is still unknown.

Minimum Number of Clues Needed in Sudoku

- It had been conjectured previously that 17 clues was the minimum. People wondered whether or not a solvable 16-clue puzzle existed.
- **Statistical Unlikelihood of a 16-Clue Puzzle**
  - Gordon Royle maintains a list on the internet of thousands of unique filled Sudoku puzzles which anyone can contribute to. there are not many new puzzles because duplicates of previous puzzles are most often submitted. Ed Russell computed an estimate saying that there were about 34,550 possible Sudoku puzzles, and Royle's database had about 33,000 puzzles at the time.
  - An observation has been made that if a 16-clue puzzle existed, you could add a 17th clue to it in 65 different ways, yielding a result which has much more variations than the record at the time of 29 for the configuration with the most puzzles. A filled Sudoku grid with 65 17-clue puzzles is unlikely seeing that the most 17-clue puzzles on a single grid is only 29.

Minimum Number of Clues Needed in Sudoku

- **The Hitting-Set Algorithm**
  - In 2013, Cornell researchers Gary McGuire, Bastian Tugemann, and Gilles Civario used computation to prove that a minimum of 17 clues are needed to complete a 9 by 9 Sudoku puzzle.
  - There is a special Sudoku grid (shown on the right) which had been suspected by some to contain a 16-clue puzzle. It held a record for having 29 known 17-clue puzzles, the most out of any known grid. This grid was what prompted the Cornell team to develop the first version of their algorithm.

6	3	9	2	4	1	7	8	5
2	8	4	7	6	5	1	9	3
5	1	7	9	8	3	6	2	4
1	2	3	8	5	7	9	4	6
7	9	6	4	3	2	8	5	1
4	5	8	6	1	9	2	3	7
3	4	2	1	7	8	5	6	9
8	6	1	5	9	4	3	7	2
9	7	5	3	2	6	4	1	8

Minimum Number of Clues Needed in Sudoku

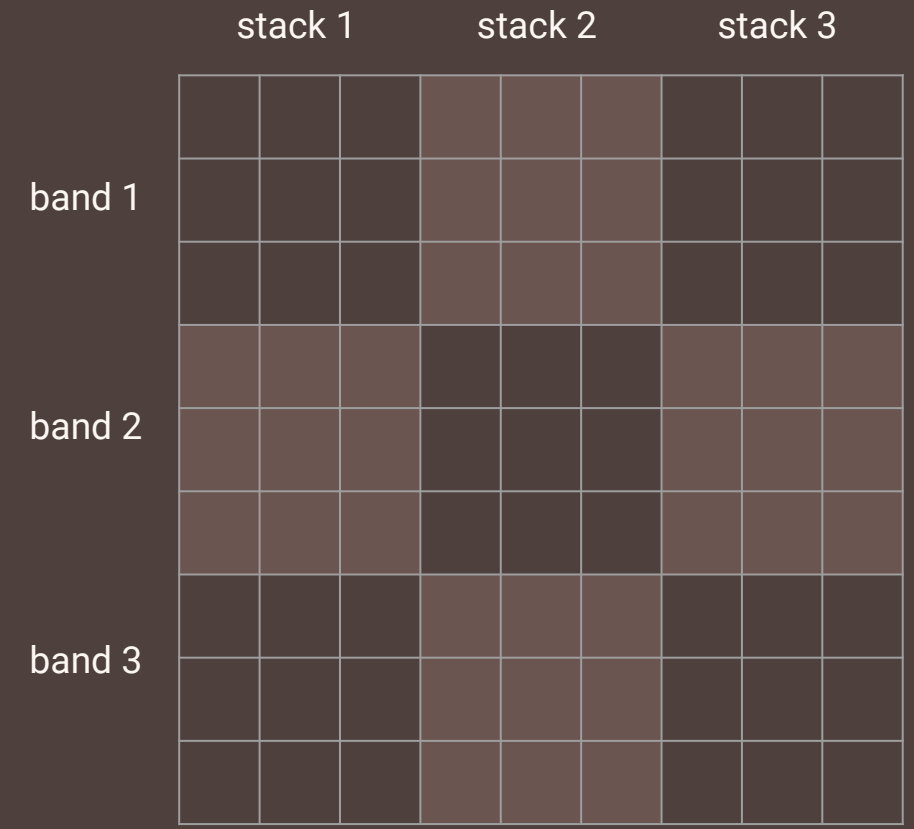
- **2005:** development began on a checker, which when given a filled Sudoku grid and a number of clues , the output is every puzzle with clues where the specified grid is the solution. This checker takes about half an hour to check one grid.
- **2009:** Development began on a new version of the checker which only took seconds to check a grid. The team later got access to a few super computers including the JUGENE supercomputer in Julich, which was Europe's fastest supercomputer at the time to use the new algorithm.
    - This algorithm uses the concept of an "unavoidable set" and that for any given Sudoku grid, it's clues must lead only to one possible solution, or else the clues are not a valid puzzle of the grid.

Minimum Number of Clues Needed in Sudoku

- **Unavoidable Sets**
  - As stated in *There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration: "In general, a subset of a sudoku solution grid is called unavoidable if it is possible to permute the contents of its cells, leaving the other cells unchanged, such that a different grid results"*
  - Since it is possible to permute an unavoidable set and still have a valid, but different, solution grid then unavoidable sets must be specified in order to only have one possible solution grid.
  - An Unavoidable set holds at least 1 clue for every puzzle of the grid so Sudoku puzzle must contain at least 1 clue from each of it's unavoidable sets. If a puzzle does not contain a clue from one of it's unavoidable sets, then there are multiple different solutions for that puzzle.
  - There is usually more than 1 unavoidable set in any given solution grid.
  - An unavoidable set has a degree *m* which is the minimum number of clues in the unavoidable set needed to uniquely obtain it when solving the puzzle.

Minimum Number of Clues Needed in Sudoku

- **Equivalence Transformations**
  - There are certain transformations that can be performed on a sudoku grid that do not cause that grid to become a different puzzle.
    1. swapping each instance of a digit with another digit
    2. swapping 2 bands or stacks
    3. swapping 3 rows within a single band or swapping 3 columns within a given stack
    4. Transposing the entire grid (flipping it on the diagonal)
  - **band:** each “block” of 3 rows
  - **stack:** each “block” of 3 columns



Minimum Number of Clues Needed in Sudoku

Finding unavoidable sets of a given grid:

1. Generate all grids that are equivalent to the given grid.
2. Compare the unavoidable sets of each of those grids.
3. If a match can be found between grids, apply equivalence transformations to the equivalent grid to convert it to the given grid.

Minimum Number of Clues Needed in Sudoku

- **Finding every possible combination of 16 clues in a given grid that intersect all the unavoidable sets in that grid:**
  - You don't actually need to find these clues, just find out if all of the unavoidable sets' degrees in the grid add to <17 to know the minimum number of clues needed to solve.
  - Multiple degree 1 unavoidable sets can be combined into that single set to speed up the algorithm.
  - You only need 1 unavoidable set for a unique puzzle, because multiple smaller unavoidable sets can be grouped together forming a "clique".
  - Any unavoidable set of a degree greater than 1 that can be made just by combining first degree sets is a "trivial" unavoidable set. Any unavoidable set that is irreducible in this manner, or not made up of first degree sets, is "nontrivial".

Minimum Number of Clues Needed in Sudoku

- **2007:** this problem had been mentioned in several publications by this point.
- **2007 - 2009:** A team at the university of Graz used a computer search to prove that there are no solvable puzzles with only 11 or 12 clues, they did not manage to do their planned search for 16 clue puzzles.
- **mid-2010:** Mladen Dobrichev writes an improved and open-source version of the Cornell team's checker.
- **late-2010:** A team at National Chiao Tung University used BONIC to search for 16-clue puzzles with an even more optimized (faster by a factor of 129) version of the Cornell checker. They managed to check over 1.45 billion cases, 26.5% of all cases that needed to be considered.

Induction Proofs and Program

What is Induction?

Induction is a method of proof in mathematics usually used to prove statements that are true for an infinite subset of or all natural numbers. $P(n)$ is the statement (also called a proposition) to be proven for some set of numbers n. Think of it as a function but evaluating to true or false depending on its truthfulness. There are 2 basic steps after the statement is established:

1. **Base case:** This is the first value for which the statement holds. Prove that $P(1)$ (when $n = 1$) is true, or starting at any other natural number as necessary.
2. **Inductive case:** Prove that $P(k) \rightarrow P(k+1)$ for all cases greater than the base case; In other words, prove that if $P(k)$ is true for some number k, then $P(k+1)$ must also be true. This will usually be done by starting at $P(k)$ and using algebra to imply $P(k+1)$. The truthfulness of $P(k)$ is assumed since the goal is to prove implication. $P(k)$ is called the inductive hypothesis.

Once this is done, it is concluded that $P(n)$ is true for all n greater than or equal to the base case.

Induction Code

Task: Prove 6 theorems using induction and create a program that displays the work done using GUI.

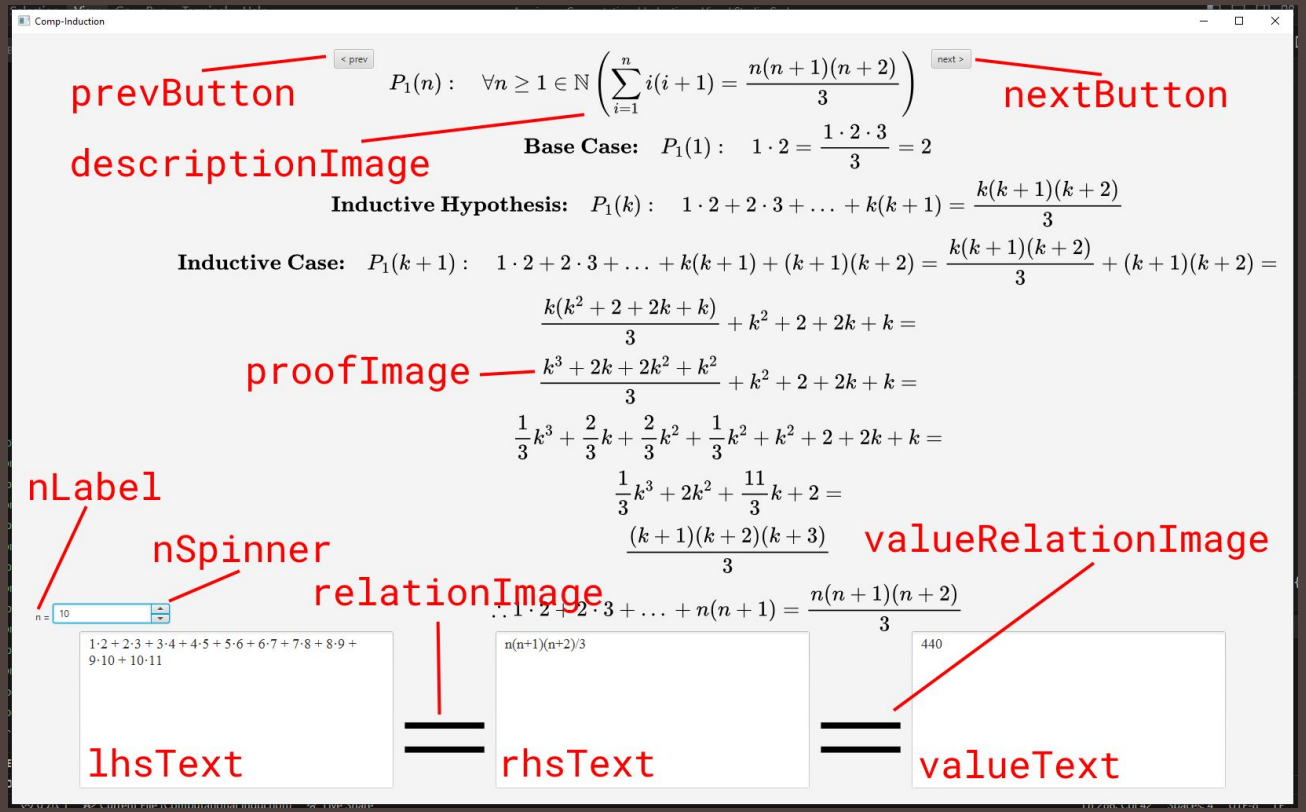
1. $(1 \cdot 2) + (2 \cdot 3) + \ldots + n(n+1) = \dfrac{n(n+1)(n+2)}{3}$
2. $1 + 4 + 7 + \ldots + (3n-2) = \dfrac{n(3n-1)}{2}$
3. $2 \cdot 2^1 + 3 \cdot 2^2 + 4 \cdot 2^3 + \ldots + (n+1) \cdot 2^n = n \cdot 2^{n+1}$
4. $1^2 + 2^2 + 3^2 + \ldots + n^2 = \dfrac{n(n+1)(2n+1)}{6}$
5. $\forall n \geq 4 (n! > 2^n)$
6. $2 + 4 + 6 + \ldots + 2n = \dfrac{n(2n+2)}{2}$

Links

Source Code: https://github.com/Ben-dev-0/Computational-Induction

Demo: https://youtu.be/EfvMGHWJdnM

Component Name Reference



Proposition Objects (App.java)

```
// P_1(n): (1*2) + (2*3) + (3*4) + ... + n(n+1) =
// n(n+1)(n+2)/3
Proposition p1 = new Proposition(
	Proposition.Relation.EQUALITY,
	"+i·i+1",
	"n(n+1)(n+2)/3",
	n -> {
		long sum = 0;

		for (int i = 1; i <= n; i++) {
			sum += i * (i + 1);
		}

		return sum;
	},
	n -> {
		return (n * (n + 1) * (n + 2)) / 3;
	}
);
```

IntegerExpression interface:

```
public interface IntegerExpression {
    long expression(int n);
}
```

Proposition objects store the data needed for each proposition (or theorem) to be calculated within the program. They are declared as seen on the left and have the following fields:

- **`relation: Relation`**
  - Stores the relation between the left and right sides of the equation which determines whether it is true at n or some range. `EQUALITY` and `LESS_THAN` are used in this program.
- **`lhsText, rhsText: String`**
  - Used to display the left side and right side expressions.
- **`lhs, rhs: IntegerExpression`**
  - Both store a lambda expression which implements the `IntegerExpression` interface, they are used to calculate the value of both sides of each equation.

Images (App.java)

```
// image resources
Image[] descriptionImages = {
        new Image("p1.png"),
        new Image("p2.png"),
        new Image("p3.png"),
        new Image("p4.png"),
        new Image("p5.png"),
        new Image("p6.png"),
};

Image[] proofImages = {
        new Image("p1_proof.png"),
        new Image("p2_proof.png"),
        new Image("p3_proof.png"),
        new Image("p4_proof.png"),
        new Image("p5_proof.png"),
        new Image("p6_proof.png"),
};

Image[] relationImages = {
        new Image("equals.png"),
        new Image("not_equal.png"),
        new Image("less_than.png"),
        new Image("greater_than.png"),
        new Image("less_than_or_equal.png"),
        new Image("greater_than_or_equal.png"),
        new Image("empty.png")
};
```

Three Arrays of `Image` objects are used to organize images used in the program.

- **descriptionImages**
  - Displays each theorem
- **proofImages**
  - Displays each proof
- **relationImages**
  - Has a set of images containing the the symbols =,≠,<,>,≤,≥, and a blank image. They appear between text fields.

GUI Components (App.java)

```
// ImageViews
ImageView descriptionImage = new ImageView();
ImageView proofImage = new ImageView();
ImageView relationImage = new ImageView();
ImageView valueRelationImage = new ImageView();

// label
Label nLabel = new Label("n = ");

// TextAreas
Font font = new Font("Times new Roman", 18);

TextArea lhsText = new TextArea("");
lhsText.fontProperty().set(font);
lhsText.wrapTextProperty().set(true);
lhsText.prefHeightProperty().set(200);
lhsText.prefWidthProperty().set(400);
lhsText.setEditable(false);

TextArea rhsText = new TextArea("");
rhsText.fontProperty().set(font);
rhsText.setEditable(false);
rhsText.prefHeightProperty().set(200);
rhsText.prefWidthProperty().set(400);

TextArea valueText = new TextArea("");
valueText.fontProperty().set(font);
valueText.setEditable(false);
valueText.prefHeightProperty().set(200);
valueText.prefWidthProperty().set(400);
```

- **ImageView**
  - Displays an image held by and Image object
- **Label**
  - Label that displays non-editable text
- **TextArea**
  - Area of wrapping text separated by a border, may be editable. The `TextAreas` `lhsText`, `rhsText`, and `valueText` are created and will be indirectly updated according to input.

PropositionsListDisplayer Object (App.java)

```
// root node & object containing root node
PropositionsListDisplayer<BorderPane> pld =
new PropositionsListDisplayer<BorderPane>(
     new BorderPane(),      //root node
     lhsText,               //TextArea
     rhsText,               //TextArea
     descriptionImage,      //ImageView
     proofImage,            //ImageView
     relationImage,         //ImageView
     valueRelationImage,    //ImageView
     descriptionImages,     //Image array
     proofImages,           //Image array
     relationImages,        //Image array
     p1,p2,p3,p4,p5,p6      //Proposition
);
```

This creates a `PropositionsListDisplayer` object called `pld` which holds various GUI components that interact with the `Proposition` objects also contained within `pld`.

The arguments highlighted at the bottom are the `Proposition` objects, which are a vararg parameter in the constructor. `PropositionsListDisplayer` inherits from a class called `PropositionsList` which contains a field for an `ArrayList` of propositions as well as a way to return their references.

Fields of the PropositionsListDisplayer Class

```
public class PropositionsListDisplayer <T extends Pane> extends PropositionsList {
     T root; //can be any root node which extends Pane
     int currentPage; //stores the current page/proposition being viewed

     // Images
     Image[] descriptions;
     Image[] proofs;
     Image[] relations;

     // Page-specific components
     ImageView descriptionImageView, proofImageView, relationImageView, valueRelationImageView;
     TextArea lhsTextArea, rhsTextArea;

     // lhs/rhs texts are declared here because they change from user input
     String[] lhsTexts;
     String[] rhsTexts;
...
```

updatePropositionText() (PLD class)

Updates the LhsTextArea of the pld object upon new values of n or on going to a different page.

- Use the lhsText of the Proposition to display each term in the sequence. The first char in this string is the operator, most of the time this is a '+'.
- Use simplifyExpression() (from the Arithmetic class) to write a condensed form of each term and append them to each other to write a sequence.
- Set the text of both TextAreas.

```
public void updatePropositionText(int n) {
     String lhsTerm = this.propositions.get(this.currentPage).getLhsText().substring(1);
     char lhsOperand = this.propositions.get(this.currentPage).getLhsText().charAt(0);
     String newLhsText = Arithmetic.simplifyExpression(lhsTerm, 1);

     for (int i = 2; i <= n; i++) {
          newLhsText += " " + lhsOperand + " " + Arithmetic.simplifyExpression(lhsTerm, i);
     }

     this.lhsTextArea.setText(newLhsText);
     this.rhsTextArea.setText(this.propositions.get(this.currentPage).getRhsText());
}
```

setCurrentPage() (PLD class)

```
public void setCurrentPage(int currentPage) {
      if (currentPage < 0 || currentPage >= this.propositions.size()) {
            return;
      }

      this.descriptionImageView.setImage(descriptions[currentPage]);
      this.proofImageView.setImage(proofs[currentPage]);

      this.currentPage = currentPage;
}
```

- If in range, set the page to the inputted one.
- Update `descriptionImageView` and `proofImageView` to the image associated with the current page.

Arithmetic.java

- Has one public method called `simplifyExpression()` and 2 private helper methods `operationStep()` and `nextOperationIndex()`. `simplifyExpression()` inputs a string of a mathematical expression with integers and optional variables, then outputs a partially simplified version, meaning parenthesis are not taken into account and are used to separate outputs in the resulting string. Exponent operations were also not implemented but they could easily be implemented by adding another statement to `simplifyExpression()`.
- '~' is used for subtraction instead of '-', since '-' must be used for reading negative integers. This helped to simplify the logic by allowing the program to easily distinguish between subtraction and negative integers.
- The inputted string can have any number of variables, the method reads each alphabetic character in the string as its own variable. The values of the variables are inputted as a vararg in the order that the variables first appear in the string.
    - Ex. "-2x~3xy+z"

```
simplifyExpression("-2x~3xy+z", -2, 1, 4);
//x=-2, y=1, z=4, returns "14"
```

simplifyExpression()

```
public static String simplifyExpression(String expression, int... varInputs) {
	String result = expression.replace(" ","");
	HashSet<Character> varsSet = new HashSet<Character>();
	Character[] varsArray;
	int currentColonIndex;

	//assign variables
	for (int i = 0; i < result.length(); i++) {
		if (Character.isAlphabetic(result.charAt(i))) {
			varsSet.add(result.charAt(i));
		}
	}

	varsArray = varsSet.toArray(new Character[varsSet.size()]);

	//substitution
	for (int i = 0; i < varsArray.length; i++) {
		result = result.replace(varsArray[i]+"",":"+varInputs[i]+":");
	}
```

- Remove all spaces from the input.
- Iterate through the input adding all alphabetic characters to a HashSet, then turn the HashSet into an array in order to substitute corresponding values from varInputs.
- `currentColonIndex` is used later.
- Substitute occurrences of variables with their corresponding value, separated from surrounding text using colons.

simplifyExpression()

```
currentColonIndex = result.indexOf(":");

while (currentColonIndex >= 0) {
        if (
                currentColonIndex != 0 && currentColonIndex != result.length()-1 &&
                Character.isDigit(result.charAt(currentColonIndex - 1)) &&
                (Character.isDigit(result.charAt(currentColonIndex + 1))
                || result.charAt(currentColonIndex + 1) == '-')
        ) {
                result = result.replaceFirst(":","*");
        }
        else {
                result = result.replaceFirst(":","");
        }

        currentColonIndex = result.indexOf(":");
}

result = result.replace("~-","+");                          //remove double negatives
result = operationStep(result,((x, y) -> {return x*y;}),'*'); //multiply
result = operationStep(result,((x, y) -> {return x/y;}),'/'); //divide
result = operationStep(result,((x, y) -> {return x+y;}),'+'); //add
result = operationStep(result,((x, y) -> {return x-y;}),'~'); //subtract

return result;
}
```

- Iterate through each occurrence of a ‘:’ in the input.
- Check if the current colon is between 2 numbers
  - If yes, replace it with a ‘*’ for multiplication
  - If not, remove it.
- Remove double negatives and apply each operation using `operationStep()`.

operationStep()

```
private static String operationStep(
String expression,
IntegerOperation operation,
char connective)
{...}
```

IntegerOperation interface:

```
public interface IntegerOperation {
    int calculate(int x, int y);
}
```

- This method loops through each occurrence of the provided char `connective` in `expression`, and applies the lambda expression `operation` accordingly. The `IntegerOperation` interface is used for this method.
- `calculate()` inputs 2 integers and calculates an operation with them. The `operation` parameter specifies a lambda expression which implements the `IntegerOperation` interface.

operationStep()

```
private static String operationStep(String expression, IntegerOperation operation,
char connective) {
	String result = expression;
	int nextOperationIndex = nextOperationIndex(result,connective);

	while (nextOperationIndex > 0) {
		int leftIndex = nextOperationIndex(result,connective) - 1;
		int rightIndex = nextOperationIndex(result,connective) + 1;
		String left = "";
		String right = "";

		//left side
		while (leftIndex >= 0 &&
		(Character.isDigit(result.charAt(leftIndex)) ||
		result.charAt(leftIndex) == '-')) {
			left = (result.charAt(leftIndex)+"").concat(left);

			if (result.charAt(leftIndex) == '-') {
				leftIndex = -1;
			}
			else {
				leftIndex--;
			}
		...
	}
```

- Use `nextOperationIndex()` to find the index of the next operand in the input, iterate through all characters that are equal to `connective`.
- For each operand, get the 2 integers that the operand is between by iterating through each char away from the current operand until a non-number is reached, and storing the sequence of chars in a string. A separate while loop does this for both sides.

operationStep()

```
//left side
while (leftIndex >= 0 && (Character.isDigit(result.charAt(leftIndex)) ||
result.charAt(leftIndex) == '-')) {
      left = (result.charAt(leftIndex)+"").concat(left);

      if (result.charAt(leftIndex) == '-') {
            leftIndex = -1;
      }
      else {
            leftIndex--;
      }
}

//right side
while (rightIndex < result.length() &&
(Character.isDigit(result.charAt(rightIndex)) ||
result.charAt(rightIndex) == '-')) {
      right += result.charAt(rightIndex);
      rightIndex++;
}
```

left side and right side loops.

operationStep()

```
result = result.replace(
       left + connective + right,
       operation.calculate(Integer.parseInt(left),
       Integer.parseInt(right)) + ""
);

nextOperationIndex =
nextOperationIndex(result,connective);
}

return result;
```

- Replace the sequence of chars iterated through with the result of the operation, specified by the inputted lambda expression.
- Re-assign `nextOperationIndex`, looking for the next operator of the specified character.
- Return the result when there are no characters left in the string equal to `connective`.

nextOperationIndex()

```
private static int nextOperationIndex(String s, char connective) {
      for (int i = 1; i < s.length() - 1; i++) {
            if (Character.isDigit(s.charAt(i-1)) && s.charAt(i) ==
            connective && (Character.isDigit(s.charAt(i+1)) ||
            s.charAt(i+1) == '-')){
            return i;
            }
      }

      return -1;
}
```

- This was decidedly used instead of indexOf(), before using ‘~’ for subtraction to avoid `StringIndexOutOfBoundsExceptions` since the operand may be the first character.
- Likely not needed anymore since ‘-’ is not an operator, so indexOf() could theoretically be used instead.

N Value Spinner

```
Spinner<Integer> nSpinner =
new Spinner<Integer>(new
SpinnerValueFactory.IntegerSpinnerValueFactory(1,99,1));
nSpinner.valueProperty().addListener(event -> {
     int spinnerValue = nSpinner.getValue();

     pld.updatePropositionText(spinnerValue);
     updateRelationSymbols(pld,spinnerValue,relationImages,
     valueText);
});
```

This spinner is used to select a value of n as input to each proposition. A spinner is a GUI component that is used to select a value from a predefined range. A `SpinnerValueFactory` is used to define possible spinner values in JavaFX.

- Create a spinner and define a range of 1-99 with a step size of 1.
- Update the proposition text with the spinner value.
- Update the relation symbol images with the spinner value.

Next/Last page buttons

```
Button nextButton = new Button("next >");
nextButton.setOnAction(event -> {
	pld.setCurrentPage(pld.getCurrentPage() + 1);
	pld.updatePropositionText(nSpinner.getValue());

	updateRelationSymbols(pld,nSpinner.getValue(),relationImages,valueText);

	if (pld.propositions.get(pld.getCurrentPage()).getRelation() ==
	Proposition.Relation.EQUALITY) {
		valueText.setText(
		pld.propositions.get(pld.getCurrentPage()).getLhsAtN(
		nSpinner.getValue())+"");
	}
	else {
		valueText.setText(
		"LHS = " +
		pld.propositions.get(pld.getCurrentPage()).getLhsAtN(nSpinner.getValue(
		)) +
		"\nRHS = " +
		pld.propositions.get(pld.getCurrentPage()).getRhsAtN(nSpinner.getValue(
		))
		);
	}
});
```

Buttons that appear on the top of the window to go to the next/last proposition.

- Set the current page and update `TextAreas`
- Set relation symbols according to calculated values.
- If the proposition is an `EQUALITY` relation, set the `valueText` to the value of the left side. If not then write both sides in `valueText`.

updateRelationSymbols() (App.java)

```
public static void updateRelationSymbols(
PropositionsListDisplayer pld, int spinnerValue, Image[] relationImages, TextArea valueText) {
        long lhsValue = pld.getProposition(pld.getCurrentPage()).getLhsAtN(spinnerValue);
        long rhsValue = pld.getProposition(pld.getCurrentPage()).getRhsAtN(spinnerValue);

        if (pld.getProposition(pld.getCurrentPage()).getRelation() ==
        Proposition.Relation.EQUALITY) {
                valueText.setText(lhsValue+"");
                pld.getRelationImageView().setImage(relationImages[0]);
                pld.getValueRelationImageView().setImage(relationImages[0]);
        }
        else {
                valueText.setText("LHS = " + lhsValue + "\nRHS = " + rhsValue );
                pld.getValueRelationImageView().setImage(relationImages[6]);

                if (lhsValue == rhsValue) {
                        pld.getRelationImageView().setImage(relationImages[0]);
                }
                else if (lhsValue > rhsValue) {
                        pld.getRelationImageView().setImage(relationImages[3]);
                }
                else if (lhsValue < rhsValue) {
                        pld.getRelationImageView().setImage(relationImages[2]);
                }
        }
}
```

This method is used to update the relation symbols inside event handlers.

- get the value of both sides of the equation (usually equal).
- If the proposition's relation is `EQUALITY`, then set the images to equality signs.
- If not, display the values of both sides and set the relation images accordingly, depending on which side is greater.

Note that even though a `long` is used to store the values, integer overflow still occurs so accuracy is not guaranteed for larger values of n.

Sources

Geuvers, J. H. "Proof Assistants: History, Ideas and Future." Sadhana (Bangalore), vol. 34, no. 1, 2009, pp. 3-25. https://www.ias.ac.in/describe/article/sadh/034/01/0003-0025

Mohar, Bojan. "V.12 The Four-Color Theorem." *The Princeton Companion to Mathematics* , edited by Timothy Gowers, et al., Princeton University Press, 1st edition, 2010. *Credo Reference*, https://ez2.maricopa.edu/login?url=https://search.credoreference.com/content/entry/prcm/v_12_the_four_color_theorem/0?institutionId=5365. Accessed 23 Feb. 2023.

McGuire, Gary. “There Is No 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem.” *arXiv.org*, 1 Jan. 2012, arxiv.org/abs/1201.0749.

Harrison, John, et al. “History of Interactive Theorem Proving.” *Elsevier eBooks*, Elsevier BV, Jan. 2014, pp. 135–214. https://doi.org/10.1016/b978-0-444-51624-4.50004-6.

Kavvos, Alex, editor. “De Bruijn Criterion.” *PLS Lab*, 3 Dec. 2022, www.pls-lab.org/en/de_Bruijn_criterion. Accessed 14 Apr. 2023.

Levin, Oscar. “Induction.” Discrete Mathematics: An Open Introduction, 3rd Edition, 9 Nov. 2022, discrete.openmathbooks.org/dmoi3/sec_seq-induction.html. Accessed 30 Apr. 2023.