A Brief Study of Automated Theorem Proving

Justin Privitera
Computer Science and Software Engineering Department
California Polytechnic State University
San Luis Obispo, California
Email: jprivite@calpoly.edu

***Abstract*—For my senior project, I wanted to learn about computerized theorem proving, with an end goal of building my very own theorem prover. Specifically, I wanted to learn how computerized theorem proving is typically done, what scope of problems are able to be solved, and what sort of infrastructure is needed to prove simple theorems. In the end, I successfully built a simple Automated Theorem Prover called JATP (Justin's Automated Theorem Prover) from the ground up. It is able to do the following: given a set of assumptions A and a set of conclusions B, it can discover the proof with the fewest number of steps that a mathematician would accept proves $A \rightarrow B$. I decided to narrow the scope of problems JATP would solve to proving facts about the parity of integers, which is a rather uncomplicated area, suitable for this sort of proof of concept. In this paper we will discuss briefly what computerized theorem proving entails, and in much more detail how JATP works.**

I. Introduction

What is computerized theorem proving? It is as it sounds; namely, a subfield of mathematical logic and automated reasoning that deals with proving mathematical theorems using computer programs [1]. This can be realized in a multitude of different ways. One of those ways is through what is called Intuitionistic Logic and Type Theory (ITT). ITT is an alternative foundation of mathematics [2]. There are some variations, but the core design is based on constructive logic. Constructive logic is centered around the idea that it is necessary to construct or find an example of a mathematical object in order to show its existence [3]. In classical mathematics, to prove the existence of an object, it is typically sufficient to prove that it is not possible for it not to exist. This notion of proof by contradiction is normally rejected by mathematical constructivism [4]. To realize this goal of constructivism, ITT uses what is called the Brouwer-Heyting-Kolmogorov (BHK) interpretation (of ITT) [5]. For a given logical formula, the BHK interpretation states what a valid proof of that formula should look like. For example, the following are three of the BHK rules:

- A proof of $P \wedge Q$ is a pair $< a, b >$ where a is a proof of P and b is a proof of Q.
- A proof of $P \vee Q$ is a pair $< a, b >$ where a is 0 and b is a proof of P, or a is 1 and b is a proof of Q.
- A proof of $P \rightarrow Q$ is a function f that converts a proof of P into a proof of Q.

There are additional rules that cover the quantifiers as well as negation. I chose these three rules to give a general sense of how the BHK interpretation works. The BHK interpretation has the added bonus of turning proofs into mathematical objects [6].

- A proof of $(A \wedge B) \rightarrow C$ is a function f that converts a proof of $(A \wedge B)$ into a proof of C.
- A proof of $A \wedge B$ is a pair $< a, b >$ where a is a proof of A and b is a proof of B.
- Putting it together, a proof of $(A \wedge B) \rightarrow C$ is a function f that converts a pair $< a, b >$ into a proof c of C, where a is a proof of A and b is a proof of B.

Instead of reasoning about proofs as abstract arguments, we can now look at them as functions, pairs, and other objects. This creates a very concrete notion of what constitutes a proof of a given logical formula, which opens the door to tackling the problem of theorem proving with computers.

Many computerized theorem provers are what are called proof assistants [6]. Essentially, a human will guide the search for a proof, aided by a computer. The computer will work on searching for a proof until it gets stuck, and the human will push it in the right direction. This process continues until a proof is found. Coq, Idris, Twelf, and Agda are some of the most well known proof assistants. During my time working on this project, I had the opportunity to work with two proof assistants, Proust and Agda.

Proust is a proof assistant built using Racket [6]. It is designed for educational purposes, and a large portion of "Logic and Computation Intertwined" (LaCI), by Prabhakar Ragde, is devoted to guiding the reader in adding functionality to Proust. Proust is built to be bare-bones, because the point is to teach the reader about proof assistants and ITT. By the end of LaCI, Coq and Agda are introduced. I had some time to experiment with Agda, which is a functional programming language and also a proof assistant. I also had the opportunity to read "The Little Typer" by Dan Friedman, which again discusses this idea of proofs as mathematical objects, or in this case, programs, and serves as another introduction to ITT [7].

The thought I had when working with proof assistants was this: oftentimes, when either Proust or Agda would get stuck and need human input to solve a problem, there would be a relatively small number of options that constituted valid input. What if, instead of waiting for a human to prod it in the right direction, it tried each of the possible options until it found one that would lead to a correct proof? This was my motivating idea for developing JATP: When searching for a proof, when the program gets to a point where there are multiple paths to take, it will try each path, running each path side-by-side until one of them signals completion, and a valid proof has been found.

II. Related Work

One of the more relevant theorem provers to this topic is called Coq, released in 1989 [8]. It helps create formal proofs of mathematical assertions, and has its roots in mathematical constructivism. However, it is not an automated theorem prover, but includes tactics to help automate parts of proofs. It is both a functional programming language that implements ITT. The name is from the French word for "rooster". One of the largest results proven using Coq is the Four-Color Theorem [9].

Another proof assistant related to this subject is Agda, which is a dependently typed functional programming language, first released in 1999. It interprets mathematical propositions as types, and it is able to assist with proving such propositions as long as there is minimal branching in the proofs [10].

III. System Design

This section will discuss what JATP does and how it works. JATP is a very simple theorem prover, built using Racket, capable of producing proofs of statements about the parity of integers. These proofs will take the form of a list of steps, where each step is the result of applying a particular axiom to the conclusions reached in the previous step. The following is an example of a proof of the following statement, generated by JATP: Given $x = y$ and $y = 2 * z$ for some z, prove x is even.

```
Example 1:
   0) Given:
      x = y
      y = (* 2 z)
   1) Applying even-reverse:
      y is even
   2) Applying subst:
      x is even
```

Disambiguation:

0) This step lists the assumptions we are given to start with. In this case, this consists of the facts "x = y" and "y = 2 * z" for some z.
1) This step shows the result of applying what is called the "even-reverse" axiom. Even-reverse is shorthand for this: if a = 2 * b, then a is even. Similarly, even-forward is shorthand for this: if a is even, then a = 2 * b for some b. In this case, even-reverse was applied, so we were able to conclude that because y = 2 * z, that y must be even. JATP only prints the new statements that have been discovered, instead of printing all of the known information. At this stage, JATP "knows" about 3 statements, the two from step 0 and the newly discovered "y is even" from the current step.
2) Finally, step 2 applies "subst", which is shorthand for substitution: if a = b and a appears within another statement, then a can be replaced with b within that statement. In this case, x = y and y appears within the statement "y is even" from step 1. Applying substitution, we can replace y with x within "y is even", yielding "x is even". This was our desired conclusion from the beginning, so the proof is complete.

Now that we have seen a short example, we are ready to discuss system design in greater detail.

Input

There are three inputs to JATP. We require some sort of representation of our initial assumptions as well as a representation of our desired conclusion. The other piece is the axioms. JATP must know which axioms it has at its disposal in order for it to find a proof of the conclusion given the initial assumptions.

First we will discuss how JATP represents assumptions and conclusions. Both of these are a set of statements, which will now be defined rigorously. A statement is a relation, and there are two kinds of allowed relations.

```
parity ::= even | odd
expr ::= c | x | (+ expr expr)
       | (* expr expr)
```

where c is a constant and x is an identifier.

Relations can take the form of $expr1 = expr2$ or $expr\ is\ parity$.

Statements can be encoded quite naturally. "x is even and equal to y and y is equal to 7 + z * 3" is encoded as "x is even" and "x = y" and "y = 7 + z * 3".

The other component is the axioms. Axioms are envisioned here as functions. These functions will take a set of statements and map them to a new set of statements, where the output set represents the result of applying the axiom to the input set of facts.

In the case that an axiom is not applicable, it will reproduce the input set of assumptions with no changes. An example of this would be trying to apply even-reverse, from earlier, to this set of statements:

```
x = y + 3
y is even
```

Because none of these statements are of the form "a = (* 2 b)", even-reverse cannot be applied.

In the case that the current axiom is applicable, the output will be the result of applying the axiom. That will look like this:

```
Given this set of assumptions as input:
  x = y
  y = (* 2 z)
```

Applying "even-reverse" here will produce the following set of statements as output:

```
x = y
y = (* 2 z)
y is even
```

So axioms never take any information away; they only either add information or preserve the existing information. This example takes the inputs from Example 1. You can see above in Example 1 that after even-reverse is applied, only the final line of the outputs, "y is even", is printed. This is done to prevent printing useless information that is already known to the user.

Finally, in the case that the axiom is applicable to the set of statements multiple times, the output will be multiple sets of statements, where each is the result of applying the axiom once. For example:

```
Given this set of assumptions as input:
  x is even
  y is even
  z is even
```

Applying even-forward to this set of statements will yield 3 different sets of statements:

```
set1:
  x is even
  y is even
  z is even
  x = 2 * a
set2:
  x is even
  y is even
  z is even
  y = 2 * a
set3:
  x is even
  y is even
  z is even
  z = 2 * a
```

This is done so that axioms can be applied precisely where they need to be applied in order to find the shortest proof of the given conclusion using the given assumptions.

Process

We will now discuss how proofs are discovered. The goal is to discover some composition of axioms (remember, in this context, axioms are viewed as functions from sets of statements to sets of statements) that will yield a correct proof, or will map the set of statements representing the given

assumptions to the set of statements representing the desired conclusion. To do this, JATP builds a tree, where each node of the tree contains a set of statements, as well as a String containing the name of the axiom used to arrive at its set of facts. To start, the given statements are packed into a node, which acts as the root of the tree.

```
def reach-conclusion(index, tree)
   curr = get_node_by_index(index, tree)
   if conclusions is a subset of
       curr.statements
      return path_to_root(index, tree)
   else
      new_children = apply_axioms(axioms,
          curr)
      return reach_conclusion(index + 1,
          tree.add_leaves(new_children))
```

At this point, recursive calls simultaneously build the tree and search for the conclusion in the following manner: For each node in the tree, JATP will first check whether the set of statements within that node contains the desired set of conclusions. If so, then we are done. Otherwise JATP applies applicable axioms to the set of statements within the current node, yielding new sets of statements, which are stored in new children of the current node. The tree is updated and the next node is visited, repeating these steps until the conclusion is found. The nodes are visited in the order of Breadth First Search. This ensures that the first time the conclusion is found, the path from the node containing the conclusion to the root of the tree is the shortest possible path.

Now we will discuss how these paths become proofs. When JATP has found the answer, it traces a path back to the root of the tree, making a list of each of the sets of statements as well as the Strings with the names of the axioms applied to reach those sets of statements. Once this list is formed, it is a simple matter of formatting it for printing, which is discussed below.

Output

At this point, our path is represented as a list of pairs, where each pair contains a set of statements and the name of an axiom used to reach that set of statements. To output a proof which looks like the examples shown above, the list must be reversed, so that the root node's statements are the first item in the list. Then each pair in the list can be printed, with the axiom name followed by the relevant statements from the set of statements. Because each axiom only adds one statement to the set of statements, it suffices to only print the most recently added statement. Sets of statements are represented internally as lists, so it is a simple matter to find the most recently added item, since it will be the last item in the list. This produces proofs formatted as seen above and in the examples to follow:

```
Example 2:
Given x = (+ y z) for some y even and z even,
    prove x is even.
0) Given:
   x = (+ y z)
   y is even
   z is even
1) Applying even-forward:
   (* 2 b) = z
2) Applying even-forward:
   (* 2 c) = y
3) Applying subst:
   (+ (* 2 c) z) = x
4) Applying subst:
   (+ (* 2 c) (* 2 b)) = x
5) Applying factor:
   (* 2 (+ c b)) = x
6) Applying even-reverse:
   x is even
```

This first example demonstrates a more complicated proof. The only axiom we have not yet seen is factor, which factors expressions.

```
Example 3:
Given x = (+ 1 (+ (+ 2 3) 4)), show x = (+ (+
    1 2) (+ 3 4))
0) Given:
   x = (+ 1 (+ (+ 2 3) 4))
1) Applying assoc:
   (+ 1 (+ 2 (+ 3 4))) = x
2) Applying assoc:
   (+ (+ 1 2) (+ 3 4)) = x
```

This next example illustrates another new axiom, associativity. JATP is able to push the numbers around to match the form it needs to.

```
Example 4:
Given x = (+ y z) for some y even and z odd,
    prove x is odd.
0) Given:
   x = (+ y z)
   y is even
   z is odd
1) Applying even-forward:
   (* 2 a) = y
2) Applying odd-forward:
   (+ (* 2 c) 1) = z
3) Applying subst:
   (+ (* 2 a) z) = x
```

```
4) Applying subst:
   (+ (* 2 a) (+ (* 2 c) 1)) = x
5) Applying assoc:
   (+ (+ (* 2 a) (* 2 c)) 1) = x
6) Applying factor:
   (+ (* 2 (+ a c)) 1) = x
7) Applying odd-reverse:
   x is odd
```

This last example shows the longest proof so far, incorporating many of the previously seen axioms.

IV. Discussion, Optimization, and Testing

The approach taken by JATP is reminiscent of that of the BHK interpretation discussed earlier [5]. Consider once again what constitutes a proof of an implication in BHK: A proof of $P \rightarrow Q$ is a function f that transforms a proof of P into a proof of Q. In our case, we are given a set of assumptions A and a set of conclusions B, and the task of JATP is to find a proof of $A \rightarrow B$. A valid proof of $A \rightarrow B$, in my system, is considered to be a function that maps A to B, or transforms A into B. This function is a composition of smaller functions, which are represented internally as axioms, discussed previously. While not identical to the BHK interpretation of logic, JATP has not strayed too far away. JATP has the added bonus of producing results that look much more familiar to those comfortable with proofs in classical mathematics.

Optimization

The largest problem in JATP is the vastness of the search space. Every time a new axiom is added, the search tree expands dramatically, leading to much longer runtimes. JATP becomes able to prove more advanced things, but at great cost.

The next logical step was to optimize JATP. The guiding principle behind JATP is to produce proofs that look similar to what a mathematician might produce. But the methodology is obviously very different; a human will use intuition to determine the best sequence of steps that might lead to the right answer, while JATP tries every combination of all available axioms until it comes up with the best solution. To optimize JATP, I attempted to bridge the gap. In the context of parity and integers, there are certain steps a human will take first in almost every proof. More specifically, a human will attempt to unpack definitions. For example, if a human saw the statements ”x is even” and ”x = y”, their first step would likely be to conclude that ”x = 2 * a”. Given information is almost always useful, and anything that can be unpacked right away ought to be. With this guiding idea, I turned to implementation. JATP now takes a fourth input, a list containing a subset of the axioms. This input is referred to internally as the ”cheats”. These axioms are to be applied straight away before beginning the main recursion, as described earlier. Every axiom of the cheats that can be applied will be, and only one is applied at a time. The result is a tree in which each node has only one child, except for the singular leaf node. A better way to describe this state is that the tree is simply a linked list.

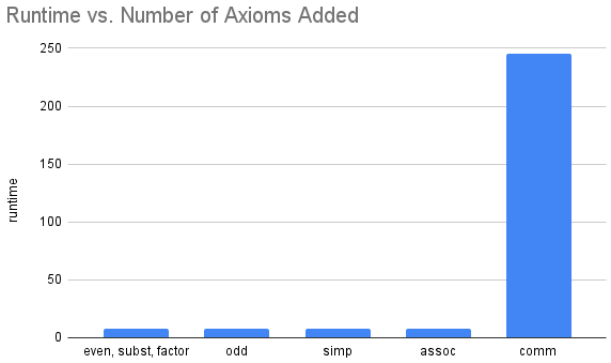

Fig. 1. A plot of runtime in seconds versus axioms. On the far left JATP is running with just 4 axioms, the evenness axioms, substitution, and factorization. What is being run are 8 fairly straightforward proofs, which only require the first 4 axioms listed in the prior sentence. As we move to the right, new axioms are added. The most significant data point is the rightmost point, as commutativity is an expensive axiom since it is applicable very often.

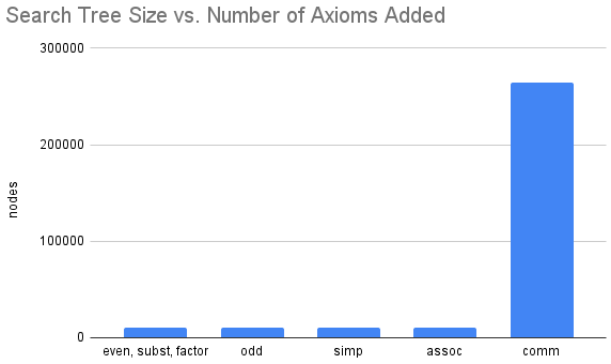

Fig. 2. A plot of search tree size in number of nodes versus axioms. On the far left JATP is running with just 4 axioms, the evenness axioms, substitution, and factorization. The same set of proofs is being run as in the prior chart. As we move to the right, new axioms are added. The most significant data point is the rightmost point, as commutativity is an expensive axiom since it is applicable very often. The search tree size very closely resembles the runtime.

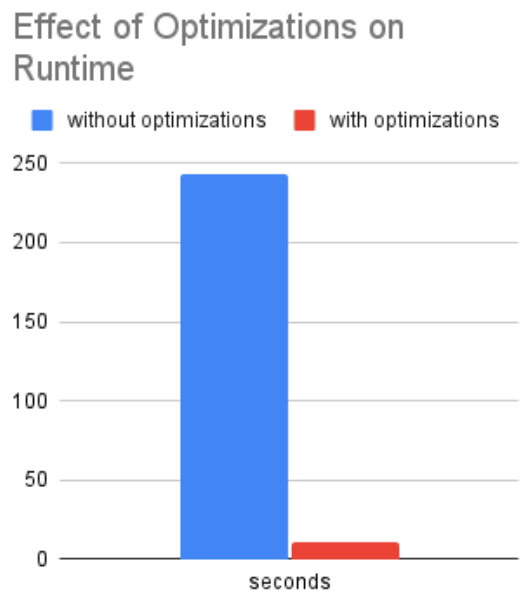

Fig. 3. Adding the optimizations described in this section drastically reduced runtime.

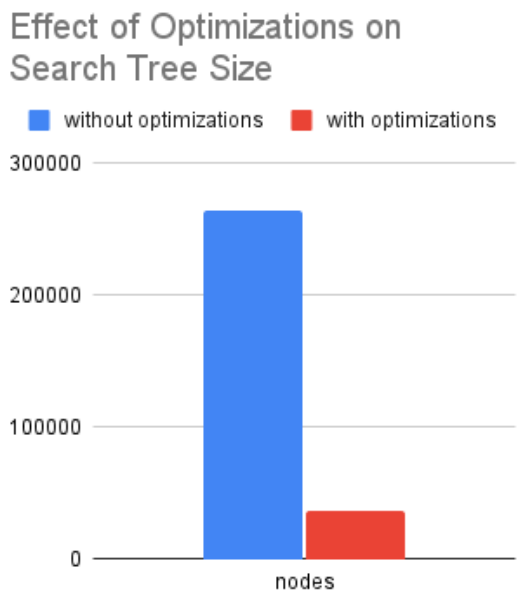

Fig. 4. Adding the optimizations described in this section drastically reduced the search tree size.

```
def cheat(index, cheats, tree)
  curr = get_node_by_index(index, tree)
  if any_axioms_applicable(cheats, curr)
    new_child =
        apply_first_applicable_axiom(cheats,
        curr)
    return cheat(index + 1,
        tree.add_leaves(new_child))
  else
    return tree
```

At this point, the tree is given to the main recursive function, reach-conclusion, described previously. Then the process continues as normal. The only difference is that the tree should have a long trunk of nodes each with singular children. Because all this work has been done to unpack definitions, in theory, finding proofs should be faster.

Testing and Validation

Aside from unit testing the various components, the main way of testing JATP is to ask it to prove things that are known to be provable. The idea behind JATP is to produce proofs that look correct from the standpoint of a mathematician, so the best way of checking for correctness of results is to look through the proofs JATP produces and verify that they indeed are correct. So testing comes down to either comparing proofs JATP produces to known correct proofs it has output previously as well as coming up with new problems for it to solve, and verifying that those solutions are also correct.

V. Limitations and Future Work

JATP is a proof-of-concept, and as such has several severe limitations. Mainly, the consequence of a vast search space is that many facts which are true take an unreasonable amount of time and computational power to prove, and so the problems that can be solved using JATP are rather small. Were I to continue there are several approaches I would take to continue optimizing JATP:

- ”cheat” on conclusions as well as initial assumptions. In the optimization section of this paper I discussed what that means for initial assumptions. If JATP could both work forwards from assumptions and backwards from conclusions and meet somewhere in the middle, that would cut down significantly on search time.
- Support normal forms for expressions [7]. Confining myself to binary expressions was costly, as axioms like commutativity and associativity are very often applicable, and grow the search tree very quickly. If expressions instead had variable numbers of elements and were represented as normal forms, it would be much easier

to manipulate them and find answers quickly, instead of randomly applying axioms.
- Ideally, I would expand functionality so that it is possible to prove things not just about integers and parity, but about other mathematical objects such as sets or groups.

If I decided instead to start from scratch instead of using what I have, it is likely I would build a new ATP more closely inspired by ITT, perhaps using Agda or Proust as a base.

VI. Conclusion

In conclusion, throughout the duration of my senior project I gained an understanding of computerized theorem proving. I had the opportunity to learn about Intuitionistic Type Theory and Logic, as well as see it in action while working with Proust and Agda. I then gained experience with theorem provers by building my own, which helped me to understand both what sorts of problems can be solved and what sort of infrastructure is required to solve them.

References

[1] Plaisted, David A. "Automated theorem proving." Wiley interdisciplinary reviews. Cognitive science vol. 5,2 (2014): 115-28. doi:10.1002/wcs.1269

[2] Moschovakis, Joan, "Intuitionistic Logic", The Stanford Encyclopedia of Philosophy (Winter 2018 Edition), Edward N. Zalta (ed.)

[3] Constructive logic. Encyclopedia of Mathematics. http://encyclopediaofmath.org/index.php?title=Constructive_logic&oldid=46487

[4] Gronwald, Jan. "Beginner's Guide to Mathematical Constructivism." Medium, Cantor's Paradise, 23 Jan. 2021, www.cantorsparadise.com/beginners-guide-to-mathematical-constructivism-4015ca66825d.

[5] "The Brouwer-Heyting-Kolmogorov Interpretation." Open Logic Project Builds, 21 June 2021, builds.openlogicproject.org/content/intuitionistic-logic/introduction/bhk-interpretation.pdf.

[6] Ragde, Prabhakar. Logic and Computation Intertwined. 2020.

[7] Friedman, Daniel P., et al. The Little Typer. The Massachusetts Institute of Technology, 2018.

[8] "The Coq Proof Assistant." A Short Introduction to Coq — The Coq Proof Assistant, coq.inria.fr/a-short-introduction-to-coq.

[9] "The Four Color Theorem." UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN College of Liberal Arts amp; Sciences Department of Mathematics, 12 Dec. 2011, faculty.math.illinois.edu/ lfolwa2/Four

[10] Bove, Ana, et al. A Brief Overview of Agda - A Functional Language with Dependent Types. www.cse.chalmers.se/ ulfn/papers/tphols09/tutorial.pdf.

Appendix A: Axioms

- even-forward: The forward direction for the evenness definition.
  - a is even $\rightarrow a = (* \ 2 \ b)$
- even-reverse: The reverse direction for the evenness definition.
  - $a = (* \ 2 \ b) \rightarrow a$ is even
- odd-forward: The forward direction for the oddness definition.
  - a is odd $\rightarrow a = (+ \ (* \ 2 \ b) \ 1)$
- odd-reverse: The reverse direction for the oddness definition.
  - $a = (+ \ (* \ 2 \ b) \ 1) \rightarrow a$ is odd
- subst: Substitution.
  - $(a = b) \wedge (a \in S \ \exists$ statement $S), \rightarrow$ any instance of $a \in S$ can be replaced with b
- comm: Commutativity. Elements can be swapped in binary expressions.
  - $(+ \ a \ b) = x \rightarrow (+ \ b \ a) = x$
  - $(* \ a \ b) = x \rightarrow (* \ b \ a) = x$
- assoc: Associativity.
  - $(+ \ a \ (+ \ b \ c)) \rightarrow (+ \ (+ \ a \ b) \ c)$
  - $(* \ a \ (* \ b \ c)) \rightarrow (* \ (* \ a \ b) \ c)$
- factor:
  - $(+ \ (* \ a \ b) \ (* \ a \ c)) \rightarrow (* \ a \ (+ \ b \ c))$
  - $(+ \ (* \ a \ b) \ a) \rightarrow (* \ a \ (+ \ b \ 1))$
- simp: Simplification. Consider n1, n2 to be integers.
  - $(+ \ n1 \ n2) = x \rightarrow n1 + n2 = x$
  - $(* \ n1 \ n2) = x \rightarrow n1 * n2 = x$

Appendix B: Source Code

Source code for this project can be found at https://github.com/JustinPrivitera/ATP2