Inference Based Theorem Prover

Arman Shah shahx252@umn.edu
Ry Wiese wiese176@umn.edu

December 19, 2018

Abstract

We present two algorithms that are capable of generating mathematical proofs for basic statements in first order logic. The creation of powerful, automated theorem provers will have a tremendous impact on the future of both mathematics and computer science, expanding human knowledge to include ideas that are difficult to prove, or even difficult to conceive. Our algorithms employ the inference rules of resolution and forward chaining, but also utilize the recursive grammatical structure of logical statements in order to simplify the inference process and generate a more readable proof. In order for the theorems generated by A.I. to be meaningful, they must be readable to humans. For this reason, we have established readability as the most important metric for analyzing our algorithms. We compare these two algorithms based of the quality of the proofs generated and runtime generating each proof. Finally, we analyze and discuss our findings to determine which algorithm is superior.

1 Introduction

Many inference algorithms, such as resolution, forward chaining, and backward chaining, exist to determine whether a given quantified statement is entailed by a knowledge base. Our problem is to, given a quantified statement and a knowledge base of known theorems and axioms, generate a proof that the quantified statement is, in fact entailed. A proof is a list of statements, each of which either binds a new variable to a value or is a statement that obviously entails from a statement in the knowledge base or from a previous statement in the proof.

As stated by Howard in [8], there is a direct correspondence between algorithms and mathematical proofs. We will implement an OCaml algorithm which, on input of a knowledge base and a statement to be proved, uses this correspondence and the recursive grammar of quantified statements to parse the statement and bind the appropriate variables to values that will make the statement true. The difficult part of this problem is choosing the proper values to bind to variables. As a simple example, consider the statement

$$\forall x, (\exists y, (y^2 = x)).$$

To prove this statement, we first bind the variable x by adding the statement "given x", to the proof, adding x to our list of bound variables, and recursively proving

$$\exists y, (y^2 = x).$$

Now, we must bind the variable y by adding "let $y = \sqrt{x}$" to the proof and adding $y = \sqrt{x}$ to the knowledge base, then recursively prove

$$y^2 = x.$$

The full proof is as follows:

Given x, let $y = \sqrt{x}$. $y^2 = (\sqrt{x})^2 = x$.

Because we chose the "right" value of y, the rest of the proof follows immediately. However, for more difficult proofs and for computers that do not have the same sense of intuition as a mathematician, this choice is far from obvious. For this task, we will use a few different OCaml functions that will be called from our main parser algorithm that will implement forward chaining to infer the proper choice.

From a purely logical point of view, any inference algorithm can be modified a bit to print out its steps, thus generating a sound proof. However, such a simple approach will almost always generate proofs that are nearly incomprehensible to the human reader. We aim to devise an algorithm which uses these inference algorithms, but generates a more readable proof. For the sake of our experiment, we are defining a "readable" proof to be one that is concise (relatively few sentences in the proof) and relevant (contains no sentences which do not help establish the goal). We are choosing to use resolution and forward chaining as our inference algorithms. With resolution in particular, the requirement of having a statement in CNF form alone will have an impact on readability. Our workaround is to use our main parser algorithm to break apart the statement to be proved into smaller parts, ideally until it is only one literal, before calling resolution. We will discuss this in more detail later.

This topic is very relevant for both mathematicians and computer scientists. Having an A.I. capable of generating proofs would be revolutionary for the field of mathematics. Mathematicians would no longer spend most of their efforts meticulously crafting proofs. Instead, they would be focusing their efforts on figuring out what interesting questions they should ask, and what the potential ramifications of certain results may be.

For computer scientists, automated theorem proving could allow a software engineer to ask a personal A.I. assistant whether the problem that they want to solve is decidable. To some extent, they may also be able to prove the correctness of the software that they have created. Automated theorem proving could also bring us significantly closer to answering the question of whether $P = NP$, an answer to which would have massive ramifications for the entire field of computer science as a whole.

2 Literature Review

The theoretical basis for automated proof generation was established by Howard in [8], which establishes a direct correspondence between algorithms and mathematical proofs. The task of creating a theorem prover is, in general, an instantiation of this correspondence. Howard establishes an extension of the λ-calculus, the foundation of functional programming languages, that allows inferential logic to be expressed algorithmic-ally. Consider the Modus Ponens inference rule. Modus Ponens can be described using traditional logic as follows:

$$\frac{\Gamma \vdash (\alpha \implies \beta), \Gamma \vdash \alpha}{\Gamma \vdash \beta}.$$

Howard converts such a statement into

$$\frac{\Gamma \vdash (E_1 : \alpha \implies \beta), \Gamma \vdash E_2 : \alpha}{\Gamma \vdash E_1 E_2 : \beta},$$

which allows the allows the functions E_1 and E_2 to be defined as $(\alpha \implies \beta)$ and α, respectively, so that the application $E_1\ E_2$ gives rise to β. Such conversions exist for all forms of inference, allowing for a computation friendly generalization of traditional logic.

Some of the common issues that arise when performing algorithmic inference are a result of large databases which give rise to many unnecessary and redundant information. Richardson proposes a probabilistic approach to minimizing these issues in [9], where he proposes using a Markov Logic Network to learn which statements in the knowledge base are prioritized based on their likelihood of being used in a proof. When proving a theorem, an experienced mathematician will never attempt to "brute force" the proof by looking at every possible theorem that he knows to be true. He or she will usually be able to look at a statement and know, based on experience, which theorems are important, relevant, and powerful enough to get the job done. Richardson's Markov Networks work similarly. By assigning weights to each statement in the knowledge base proportional to the probability that they will be used in a future proof, the network ensures that commonly used theorems are attempted first. This provides numerous advantages, the most obvious being that irrelevant theorems will be less likely to used as a base for inference, drastically reducing the runtime. Moreover, there are often multiple ways to prove a statement and usually some proofs will be much more clear and concise than others. One proof may require simply invoking a known powerful theorem from which the desired statement is immediately entailed, while another may require pages worth of statements before arriving at the same conclusion. In a worst case situation, the longer proof will be essentially reproving a more powerful and well known theorem as a part of the original proof. In theory, that powerful statement will be assigned a higher probability and will be selected first for inference so that both the runtime and the complexity of the argument will be reduced.

In [11], Slaney provides an improvement to the traditional forward chaining approach of proof generation in an algorithm called SCOTT. He describes forward chaining as a "blind" process, which does not use information contained in the goal statement other than to check whether a newly inferred statement unifies with the goal. The improvement he provides is to combine forward chaining with the Finder algorithm, which works rather differently from most other theorem proving algorithms. Instead of repeatedly applying inference rules in order to deduce a goal state, it generates models of the theory in which the goal statement (potentially) exists, and tests whether the statement is consistent with the model. In the case of a non-consistent statement, the model is adjusted and the process is repeated. Intuitively, instead of testing whether the statement can be deduced by the theory, it figures out what needs to be true of the theory in order for the statement to be true. A false statement may be determined false, for example, by showing that it can only be true if a certain statement, which is known to be false, is present in the knowledge base. By combining this algorithm, SCOTT is able to determine what statements are necessary to prove the goal, and using those statements as starting points for the forward chaining search.

Prolog is a logic based programming language which is often the first choice for creators of theorem provers. It is a powerful language whose design is well suited for proof generation. It does, however, have many limitations. It is limited to knowledge bases containing Horn clauses, those that contain at most one positive literal. Moreover, it is not built to handle first order logic. Manthey overcomes these limitations in his SATCHMO theorem prover, described in [6], by extending Prolog to handle first order logic and implementing a model-elimination reduction rule as well as an occurs check. Together, these extensions allow for a complete algorithm, even without the Horn clause restriction. Stickel takes a similar approach in [12], with the addition of an iterative deepening version of Prolog's depth first search algorithm, making his prover both complete and efficient.

Nevins uses similar techniques, particularly forward and backward chaining, to implement a theorem prover for Plane Geometry in [7]. His paper addresses the difficulty of proofs which require the introduction of new points on an existing geometric figure. This difficulty arises due to the fact that any line segment in a figure contains an infinite number of points. In general, it is much more difficult to construct an algorithm that can perform inference efficiently over infinite domains. It is straightforward to argue about the finitely many line segments in a figure, the finitely many already labeled points in a figure, but the infinite number of points that have not yet been label prevent an obstacle. This obstacle is similar to that of first order inference over infinite domains, such as the integers. Nevins' algorithm is unable to solve this problem, and his problem space is restricted to proofs that do not require the introduction of new points. However, this issue serves to further motivate the creation of an algorithm capable of deduction over infinite domains. Such an algorithm would have benefits beyond first order inference.

In 2011, Asperti wrote about the interactive theorem prover [1]. Essentially, the research helps the user create a proof. It is mainly used for verifying correctness of the users proof, but can also simplify what the user is writing. Another interesting area in the research is the product can correct for when the user makes a logical paradox while trying to prove something. The paper doesn't go into detail on how they accomplish this, but rather explain what they can do. Another problem is this relies heavily on a user to create a proof, while our problem is to automatically do that.

In 2007, Bonichon wrote about their automated theorem prover, Zenon, for first order logic [3]. While testing, Zenon was shown to be able to find proofs for many problems. The problems that Zenon couldn't find proofs for were mainly because of the time limit (5 minutes) and size limit (400 megabytes). Zenon uses the tableau method, but it is mentioned in [3] that it isn't that efficient compared to other methods, like resolution.

In 2018, Kusumoto proposed deep reinforcement learning algorithm as an automated theorem prover for propositional logic [5]. Instead of treating the problem as a planning problem, they treat it as a search problem. Neural Networks are used to figure out which states should be explored while searching. After applying four approximate policy iterations to allow the algorithm to learn, it solved around 86% of theorems provided. The main issue with the approach in [5] is the need for large amounts of data to train the deep reinforcement learning algorithm.

One of the algorithms we use to generate proofs is resolution. There has been plenty of work done to help us implement this algorithm. In fact, Robinson in [10] writes on a theoretical bases on how to resolution while keeping the explanation relevant to computing. He mentions that first order logic was made to proof algorithms in computers, but resolution allows for simplistic approach using inference instead of deduction. The rest of the paper essentially describes how inference and resolution work, providing many theorems and lemmas. This paper gave us a strong foundation in order to implement resolution.

Bacchus wrote in [2], that a backwards chaining algorithm has a benefit over forward chaining algorithm in that backwards chaining will never consider actions that aren't relevant to goal, but forward chaining may take require backtracking when a wrong action is taken. On the other hand, a negative of backwards chaining is that it will have less knowledge of the world compare to forward chaining. backwards chaining will never consider actions that aren't relevant to achieving the goal. The real interesting addition is that they restrict the domain of the search space by evaluating with first order linear temporal logic. The only differences between first order logic and first order linear temporal logic is that the operations until, always, eventually, and next are added. Adding these operations allows for the domain of the search space of actions to reduce and become more goal driven, solving the downside of forward chaining. The main reason this modification to forward chaining is not applicable to our approach is that in [2] forward chaining was for planning, so they assume that the only positive literals are the goal's set of literals and the rest are negative literals. For generating a valid proof, we can't just assume the goal has the only positive literals and therefore this modification isn't valuable to us.

In [4], Fink writes about working both ways. The article talks about having two incomplete plans: head plan and tail plan. The head plan goes from initial state to some current state and gets modified by taking an action from tail plan. The tail plan in just a backward chain from the goal state to the current state. The algorithm, named Prodigy, makes a decision of when to move an action from tail plan to head plan or just backward chain. The general idea of Prodigy is that the algorithm uses the tail plan as a smaller search space to see if the current state can apply any of those actions, otherwise the algorithm will chose to backwards chain to grow the search space. This is apparently more efficient than just backwards chaining. The paper brings up the idea of mixing backwards chaining and forward chaining and [4] describes their solution as backwards chaining with elements of forward chaining.

3 Approach

As stated by Howard in [8], there is a direct correspondence between algorithms and mathematical proofs. We will implement an OCaml algorithm which, on input of a knowledge base and a statement to be proved, uses this correspondence and the recursive grammar of quantified statements to parse the statement, bind the appropriate variables, and then prove the core statement using either resolution or forward chaining.

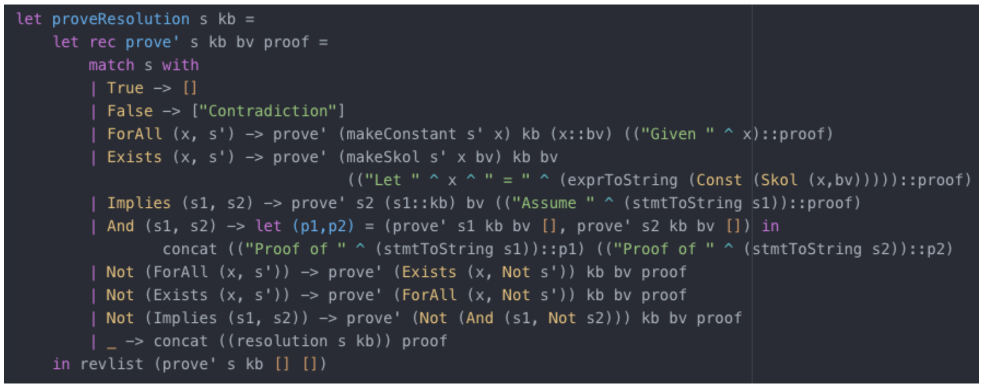
We have chosen OCaml because it is a powerful language for pattern matching which makes it very easy to represent logical statements. For example, the statement $\forall x, (\exists y, (y^2 = x))$ can be expressed in OCaml as

$$ForAll(''x'', Exists(''y'', Equals(Times(Var''y'', Var''y''), Var''x''))).$$

This allows us to use pattern matching to guide the proof, as shown in Figure 1.

There are a few known techniques for making readable proofs. When trying to prove a statement of the form $\forall x, s'$, we can say "Given x," and then proceed to prove s' recursively. When trying to prove a statement of the form $\exists x, s'$, we can assign a value to x by saying "Let $x = ...$" and then recursively proving s'. When proving a statement of the form $p \implies q$, we can say "Assume p", add p to the knowledge base, and then recursively prove q. We have compiled these rules into a top level algorithm which parses our statement and builds up a start to the proof before passing it off to resolution or forward chaining. This top level algorithm is shown in Figure‘1 for resolution. A nearly identical function **proveFC** exists for forward chaining, differing only in that the **forwardChain** algorithm is called in the second to last line, instead of **resolution**.

This process of parsing through the statement is essentially performing some of the first steps of converting a statement into CNF form. Saying "Given x' allows us to drop the universal quantifier. Saying "Let x= ..." is a way of making explicit the process of Skolemization. We are replacing the existentially bound x with a new value, the result of applying a Skolem function to x depending on which variables are universally bound. The makeSkol function shown in the algorithm essentially uses the forward chaining algorithm (shown later and not repeated here) to obtain the proper value for x in order to make the proof true. Breaking up an $p \implies q$ is done instead of replacing the statement with



```
let proveResolution s kb =
    let rec prove' s kb bv proof =
        match s with
        | True -> []
        | False -> ["Contradiction"]
        | ForAll (x, s') -> prove' (makeConstant s' x) kb (x::bv) (("Given " ^ x)::proof)
        | Exists (x, s') -> prove' (makeSkol s' x bv) kb bv
                                    (("Let " ^ x ^ " = " ^ (exprToString (Const (Skol (x,bv))))::proof)
        | Implies (s1, s2) -> prove' s2 (s1::kb) bv (("Assume " ^ (stmtToString s1))::proof)
        | And (s1, s2) -> let (p1,p2) = (prove' s1 kb bv [], prove' s2 kb bv []) in
                concat (("Proof of " ^ (stmtToString s1))::p1) (("Proof of " ^ (stmtToString s2))::p2)
        | Not (ForAll (x, s')) -> prove' (Exists (x, Not s')) kb bv proof
        | Not (Exists (x, s')) -> prove' (ForAll (x, Not s')) kb bv proof
        | Not (Implies (s1, s2)) -> prove' (Not (And (s1, Not s2))) kb bv proof
        | _ -> concat ((resolution s kb)) proof
    in revlist (prove' s kb [] [])
```

Figure 1: Top level function that simplifies a statement and establishes the first part of a proof before calling the resolution algorithm.

$\sim p \vee q$. The resolution algorithm requires the supplied statement to be in CNF form (and hence makes that conversion as an initial step). CNF statements are known to be fairly unreadable for humans, so an effect of this is that an algorithm which only uses resolution will generate a very difficult to follow proof. Our top level algorithm reduces the complexity of the statement to be proved using resolution or forward chaining, and hence maximizes the readability of the proof.

Our algorithms for resolution and forward chaining have been adapted to OCaml and modified to allow the continued modification of the proof, which is represented as a list of strings, where each string represents a line in the proof. The resolution algorithm relies on a few helper functions, which we will briefly describe below:

resolveLit$(literal, clause, proof)$ takes as input a literal, a clause, and a proof, and returns a list of all possible clauses that can be formed by resolving $clause$ with $literal$.

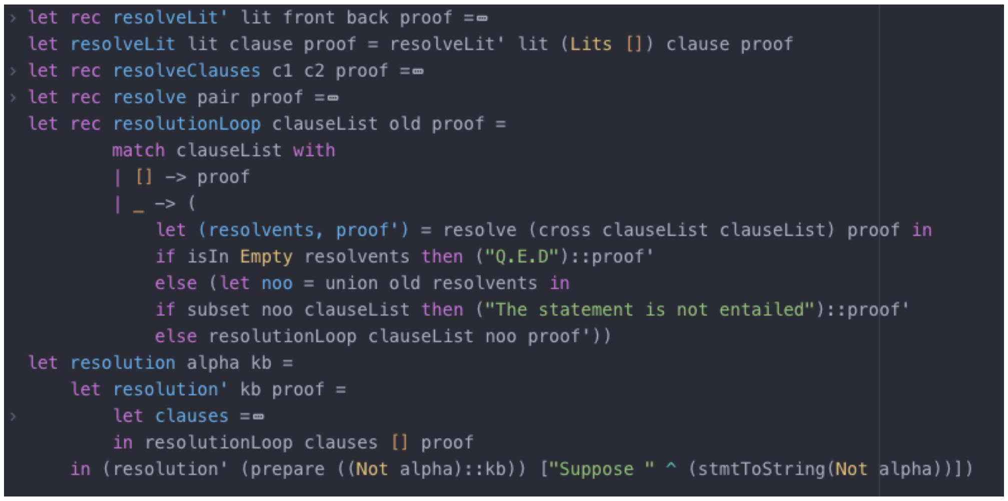
resolveClauses$(clause1, clause2, proof)$ returns a list of possible clauses which can be resolved from $clause1$ and $clause2$ as well as an updated proof. It calls **resolveLit**$(l, clause2, proof)$ for each literal in clause1, and then constructs the possible resolved clauses from the results. If a new clause can be added, a description of how it was derived is added to the initial proof.

resolve$(pairs, proof)$ accepts a list of all possible pairs of clauses in the knowledge base, and returns the result of calling r**esolveClauses** on each pair, along with an updated proof.

resolutionLoop$(clauseList, old, proof)$ performs the main tasks of the resolution algorithm. It calls $resolve$ on the set of all pairs of clauses and the current proof, and stores the list of resolvents as *resolvents* and the new proof as *proof′*. It then tests if the *Empty* clause is in *resolvents*, in which case a contradiction has been found and the proof can be returned (with "Q.E.D" appended to signify that the proof is complete). Otherwise, it combines *resolvents* with the *old* resolvents and runs the algorithm again until the *Empty* clause has been found or until nothing new can be resolved.

resolution($alpha, kb$) is essentially a wrapper function which adds $\sim alpha$ to the knowledge base, converts it to CNF form, and calls **resolutionLoop**.

The code has been shown in Figure 2 below:



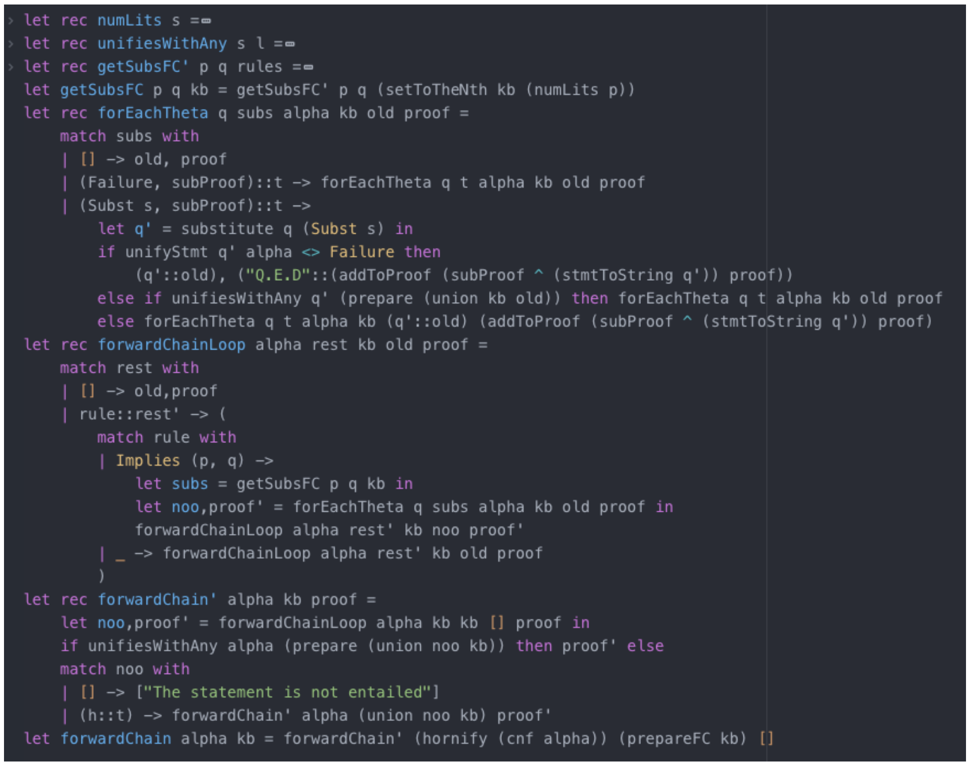
```
› let rec resolveLit' lit front back proof =⋯
  let resolveLit lit clause proof = resolveLit' lit (Lits []) clause proof
› let rec resolveClauses c1 c2 proof =⋯
› let rec resolve pair proof =⋯
  let rec resolutionLoop clauseList old proof =
          match clauseList with
          | [] -> proof
          | _ -> (
              let (resolvents, proof') = resolve (cross clauseList clauseList) proof in
              if isIn Empty resolvents then ("Q.E.D")::proof'
              else (let noo = union old resolvents in
              if subset noo clauseList then ("The statement is not entailed")::proof'
              else resolutionLoop clauseList noo proof'))
  let resolution alpha kb =
      let resolution' kb proof =
›         let clauses =⋯
          in resolutionLoop clauses [] proof
      in (resolution' (prepare ((Not alpha)::kb)) ["Suppose " ^ (stmtToString(Not alpha))])
```

Figure 2: Resolution algorithm, modified to return a proof. The bodies of some helper functions have been excluded.

We have also adapted the forward chaining algorithm into OCaml, as shown below in Figure 3.

getSubsFC($predicate, q, kb$) returns a list of substitutions that unify *predicate* with the knowledge base.

forwardChainLoop($alpha, rest, kb, old, proof$) recurses through the list of rules



```
› let rec numLits s =…
› let rec unifiesWithAny s l =…
› let rec getSubsFC' p q rules =…
  let getSubsFC p q kb = getSubsFC' p q (setToTheNth kb (numLits p))
  let rec forEachTheta q subs alpha kb old proof =
      match subs with
      | [] -> old, proof
      | (Failure, subProof)::t -> forEachTheta q t alpha kb old proof
      | (Subst s, subProof)::t ->
          let q' = substitute q (Subst s) in
          if unifyStmt q' alpha <> Failure then
              (q'::old), ("Q.E.D"::(addToProof (subProof ^ (stmtToString q')) proof))
          else if unifiesWithAny q' (prepare (union kb old)) then forEachTheta q t alpha kb old proof
          else forEachTheta q t alpha kb (q'::old) (addToProof (subProof ^ (stmtToString q')) proof)
  let rec forwardChainLoop alpha rest kb old proof =
      match rest with
      | [] -> old,proof
      | rule::rest' -> (
          match rule with
          | Implies (p, q) ->
              let subs = getSubsFC p q kb in
              let noo,proof' = forEachTheta q subs alpha kb old proof in
              forwardChainLoop alpha rest' kb noo proof'
          | _ -> forwardChainLoop alpha rest' kb old proof
          )
  let rec forwardChain' alpha kb proof =
      let noo,proof' = forwardChainLoop alpha kb kb [] proof in
      if unifiesWithAny alpha (prepare (union noo kb)) then proof' else
      match noo with
      | [] -> ["The statement is not entailed"]
      | (h::t) -> forwardChain' alpha (union noo kb) proof'
  let forwardChain alpha kb = forwardChain' (hornify (cnf alpha)) (prepareFC kb) []
```

Figure 3: Resolution algorithm, modified to return a proof. The bodies of some helper functions have been excluded.

in the knowledge base, finds the list of substitutions, *subs*, that allow the predicate of the rule to be unified with known statements in the knowledge base (using **getSubsFC**). For each element of *subs*, the **forEachTheta** function is called.

forEachTheta$(q, subs, alpha, kb, oldproof)$ recurses through each substitution in *subs*. If q', the result of substituting q with the current element of *subs*, unifies with *alpha* then the proof is complete and we return this result, along with "Q.E.D", added to the proof. Otherwise, we add q' to the list of new statements (unless it is already in it).

forwardChain$(alpha, kb)$ is, again, a wrapper function whose job is to repeatedly call **forwardChainLoop** until an answer is found or nothing new can be added.

4 Experiment Design

To test which algorithm, proveResolution or proveFC, was better at creating proofs, 15 different statements where ran and the proofs were recorded. From the proof itself, we calculated the percent of unnecessary lines in the proof, the total amount of lines need to prove the statement, and the minimum lines the proof needs to make sense. Run time was, also, gathered to see how long the algorithm took in order to generate a proof.

A common knowledge base was used for all the proofs. So, all statements we wanted to prove were using the same knowledge base. The knowledge base includes axioms of equality and less than, but also sentences we decided were true in the scope of the problem. Our knowledge base includes:

1. $\forall x$ x = y
2. $\forall x, y$ x = y $\implies$ y = x
3. $\forall x, y, z$ x = y $\wedge$ y = z $\implies$ x = z
4. $\forall x$ 0 = 0 * x
5. $\forall x$ $\neg$(x < x)
6. $\forall x, y$ x < y $\implies$ $\neg$(y < x)
7. $\forall x, y, z$ x < y $\wedge$ y < z $\implies$ x < z
8. $\forall x$ 0 < x $\vee$ 0 = x
9. $\forall x$ x < x + 1
10. $\forall x$ x < x * x
11. $\forall x \exists y$ x < y
12. Ry = Arman
13. Parker = Arman
14. $\forall x$ x < Ry

Our 15 statements that we used to test the algorithms will be referred to by their number and are listed below:

1. $Parker = Arman$
2. $\exists x x = Arman$
3. $\forall x Parker = x \implies x = Ry$
4. $\neg(Ry < Arman)$
5. $\forall x, y \exists z x + y < z$
6. $\forall x \exists y x = y$
7. $\forall x \exists y x * x < y$
8. $\forall x x * x < x * x * x$
9. $Arman * Arman < Ry * Ry$
10. $Arman * Parker < Ry * Parker$
11. $\forall x, y \exists z z = x + y$
12. $\exists x \forall y x < y \vee x = y$
13. $\forall x \exists y x * (1 + 1) < y$
14. $\neg \exists x \forall y y < x$
15. $\neg \forall x x < x * x$

For example, consider statement 3, $\forall x[(Parker = x) \implies (x = Ry)]$. Our **proveResolution** algorithm generated the following proof. It is worth noting that our algorithm does include functions that make these lines more readable by reversing the CNF conversion process and by making the substitutions used in resolution.

Proof. Given x.
Assume $Parker = x$.
Suppose $\neg(x = Ry)$.
$\neg(x = Ry)$ and $(Ry = x) \implies (x = Ry)$ therefore $\neg(Ry = x)$.
$\neg(x = Ry)$ and $\forall y, (x = y) \implies \neg(y = Ry) \vee (x = Ry)$ therefore $\forall y, ((x = y) \implies \neg(y = Ry))$.
$Parker = x$ and $(Parker = x) \implies (x = Parker)$ therefore $x = Parker$.
$Parker = x$ and $\forall f, [(f = Parker) \implies (\neg(Parker = x) \vee (f = x))]$ therefore $\forall f, [(f = Parker) \implies (f = x)]$.
$Parker = x$ and $\forall z, [(Parker = x) \implies \neg(x = z) \vee (Parker = z)]$ therefore $\forall z, [(x = z) \implies (Parker = z)]$.
Using y/x and c/Ry, $x = Ry$ and $\neg(x = Ry)$.
Contradiction.
Q.E.D.

$\square$

It is clear that the lines
$\neg(x = Ry)$ and $(Ry = x) \implies (x = Ry)$ therefore $\neg(Ry = x)$
and
$Parker = x$ and $(Parker = x) \implies (x = Parker)$ therefore $x = Parker$
are not relevant to the proof, and there are a total of 11 lines, so the percentage of useless lines in this proof is 18.2%.

5 Results

After running both algorithms we assessed the quality of the proofs as seen in Table 1. The table shows what statement we are running the algorithms on, the total number of lines in the generated proof, the number of lines that were irrelevant in the proof, and the percentage of those useless lines. Statement 12 doesn't have any values for proveFC because it determined there was no entailment of that statement in the knowledge base. This is because Statement 12 is not a horn clause and therefor proveFC can't compute a proof. Statement 6 on proveFC did not generate a correct proof either, so that data may not be useful.

Besides determining the quality of the proofs, we gathered runtimes of how long each proof took to make for each algorithm in seconds. This is shown in Figure 4. Again, statement 6 shouldn't really be taken into account since the proof wasn't correct and statement 12 returned that the statement didn't entail in the knowledge base and is potentially a reason why it took so long to run.

Table 1: Data of Quality of Proofs for proveResolution and proveFC

	proveResolution			proveFC		
Statement #	Total # of Lines	# of Useless Lines	% of Useless Lines	Total # of Lines	# of Useless Lines	% of Useless Lines
1	7	2	28.6%	4	2	50.0%
2	6	0	0.0%	4	0	0.0%
3	11	2	18.2%	12	4	33.33%
4	9	3	33.3%	5	0	0.0%
5	9	1	11.1%	4	2	25.0%
6	8	0	0.0%	5	2	20.0%
7	8	1	12.5%	7	2	28.6%
8	7	0	0.0%	6	2	33.3%
9	6	0	0.0%	5	2	40.0%
10	6	0	0.0	5	1	20.0%
11	7	1	14.3%	8	2	25.0%
12	8	1	12.5%	-	-	-
13	7	0	0.0%	7	1	14.3%
14	5	0	0.0%	7	2	28.6%
15	5	0	0.0%	6	2	33.3%
Average	7.3	0.73	10.0%	6	1.71	28.5%

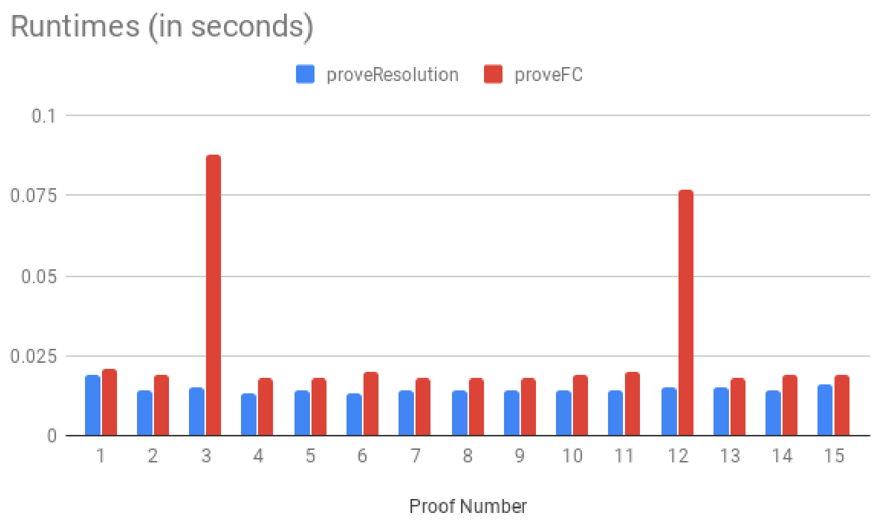


Figure 4: Runtime in seconds of how long it took each algorithm to generate a proof.

6 Analysis

The proveResolution algorithm tends to write longer proofs than proveFC. The average number of lines for a proof generated by proveResolution was 7.3, as opposed to 6 for proveFC. A main reason the proofs are long with proveResolution is that the proof isn't direct. Meaning, proveResolution is proving by contradiction and is a more roundabout way to prove than proveFC seeing if it is possible. Proving by contradiction automatically adds a line in the proof stating a contradiction has been found. Another reason the proofs may be longer is due to the need to be in CNF. If $p \longrightarrow q$, then with proveFC we take p and try unify till we get q. The resolution algorithm converts that to $\neg p \vee q$ and requires the algorithm eliminate more terms to get to a contradiction.

Although proveResolution has longer proofs, the amount of useless lines is lower compared to proveFC. Proofs made with proveResolution had, on average, a useless line percentage rate of only 10%, while proveFC had almost three times as many useless lines. In terms of readability, having irrelevant information is a lot more confusing than a lengthy/non-concise proof. Although the resolution based proofs are a bit wordier, they contain less useless and confusing information, so we consider it to be superior to the forward chaining version. It is possible that some of these irrelevant statements could be predicted and removed. For example, proveFC always decided to reverse the order of $Ry = Arman$ and $Parker = Arman$ to $Arman = Ry$ and $Arman = Parker$, even when the statement had no references to these names. In order to cut down on useless lines, we could write an algorithm to specifically look for these types of statements in the proof.

It is also worth noting that since proveFC generated shorter proofs, improving the algorithm to eliminate irrelevant information would likely mean that the generated proofs are more readable than those created by proveResolution. However we are still limited by the fact that forward chaining is only complete for horn clauses. A possible solution would be to write a top level algorithm that tests to see if a statement can be proved with an improved version of proveFC, and use proveResolution if it cannot.

Except in 2 extreme cases, the runtimes of proveResolution and proveFC were fairly close. Resolution seemed to be a bit faster than forward chaining, and proveFC had, on average, about 1.25 times the runtime of proveResolution. For proofs as simple as the ones we tested, this difference in runtime is negligible. However, for more complicated statements that we do not have the structure to represent at this time, this difference in speed could prove to be advantageous. It is also worth noting that, for sentences 3 and 12, the runtimes of proveFC were 5-6 times greater than those of proveResolution. Although these occurences are fairly infrequent, and as in the case of statement 12, more likely to happen for a statement that is not comprised of horn clauses, this could present a problem when more complicated statements are introduced. It is possible that forward chaining may not be a viable solution to finding proofs of such statements.

It is unclear why the runtime for proveFC on statement 3 was significantly larger than the rest, but it is worth noting that, for both resolution and forward chaining, this statement generated the longest proof. It is likely that the high runtime resulted from the algorithm started chaining extra sentences that didn't pertain to proof. It is also possible that the time complexity of proveFC is more complex than that of proveResolution, so slightly longer proofs result in significantly longer runtimes. Overall, the runtime is pretty short. Using OCaml versus an object-oriented programming language allowed the computer to run the algorithms faster.

7 Conclusion

In this paper we have constructed two algorithms, proveResolution and proveFC, which use the inference rules of resolution and forward chaining to construct mathematical proofs. Before employing their respective inference algorithms, these algorithms take advantage of the grammatical structure of logical statements in order to generate a more readable proof and to reduce the complexity of the statement that must be proved using the actual inference algorithm.

After comparing the readability and runtimes of our algorithms, we conclude that, as of now, proveResolution seems to be a superior algorithm. Even though the proof length is longer, the overall quality of the proofs generated by proveResolution is better than those generated by proveFC. Also, proveResolution was always faster than proveFC when generating a proof. However, the results of our experiment have given us insight into ways to improve proveFC in order to make the proofs more readable.

However, proveFC is not complete for all statements, and this is a problem which we cannot improve. This being the case we admit that even a highly powerful forward chaining algorithm must be used in conjunction with resolution in order to guarantee the completeness of our algorithms. For the time being, the proveResolution algorithm is the better algorithm while trying to generate proofs.

8 Future Work

Due to the short time frame of the project, we chose to focus our efforts on developing correct and efficient algorithms for inference, as this has been the focus of the course. This has given us less time to invest in constructing OCaml types and patterns capable of representing more complex mathematics. For the sake of this project, we have limited our proofs to those in the theory of the natural numbers under multiplication and addition. Moving forward, we would like to improve our algorithms to handle more complicated pattern matching to perform inference on negative integers, real numbers, and more complex operations. We would also like to engineer a way to represent sets in OCaml, which would significantly improve the maturity of our algorithms. According to the axioms of specification, a set is represented as a free variable, the superset to which the free variable belongs, and a logical statement that contains the free variable. For example, the set of all even integers can be represented as $\{z \in \mathbb{Z} | z\ mod\ 2 \equiv 0\}$. In OCaml we could represent this as the tuple $(z, \mathbb{Z}, (z\ mod\ 2 \equiv 0))$, but building such a language, as well as machinery to parse it, would have been too time consuming for the scope of this project.

We would also like to implement a Markov Logic Network as described in [9]. This would allow our algorithm to "learn" which knowledge base statements are most likely to be used in a proof, and pick them first. This would have a huge improvement on runtime as well as readability, as less irrelevant statements will be checked or added to the proof. To construct a robust version of this logic network, we may consider representing our knowledge base as a graph,rather than a list, where each node represents a statement. This structure would allow for a more expressive way of representing the relationships between knowledge base statements and allow for more traditional search techniques to be used within the algorithm. This would also make it easier to construct a path to the goal answer, and then reconstruct that path to generate the proof, which would cut down on the inclusion of irrelevant statements.

9 Contributions

The work for this project was divided pretty evenly. We worked together to construct the algorithms using a whiteboard, and then translated our pseudo-code into OCaml together. Ry took the lead on writing the paper while Arman took the lead on performing the experiment (which we designed together), however we were working next to each other the entire time and were able to talk about all aspects of the project while we were doing it.

References

[1] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. The matita interactive theorem prover. In *International Conference on Automated Deduction*, pages 64–69. Springer, 2011.

[2] F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. In *Proceedings of the 3rd European Workshop on Planning*, pages 141–153, 1995.

[3] R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 151–165. Springer, 2007.

[4] E. Fink and M. Veloso. Formalizing the prodigy planning algorithm. *New Directions in AI Planning*, pages 261–271, 1996.

[5] M. Kusumoto, K. Yahata, and M. Sakai. Automated theorem proving in intuitionistic propositional logic by deep reinforcement learning. *arXiv preprint arXiv:1811.00796*, 2018.

[6] R. Manthey and F. Bry. Satchmo: A theorem prover implemented in prolog. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 415–434, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[7] A. J. Nevins. Plane geometry theorem proving using forward chaining. *Artificial Intelligence*, 6(1):1–23, 1975.

[8] C.-H. L. Ong and C. A. Stewart. A curry-howard foundation for functional computation with control. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 215–227, New York, NY, USA, 1997. ACM.

[9] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1):107–136, Feb 2006.

[10] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[11] J. Slaney. Scott: A model-guided theorem prover. In *IJCAI*, volume 93, pages 109–114. Citeseer, 1993.

[12] M. E. Stickel. A prolog technology theorem prover: Implementation by an extended prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, Dec 1988.