university of groningen / faculty of science and engineering / bernoulli institute

Towards Automated Natural Deduction in Prolog

Bachelor's Project Thesis

Flip Lijnzaad
Supervisor: Dr. B.R.M. Gattinger

Abstract: We present a theorem prover for propositional logic that uses Fitch-style natural deduction to find formal proofs that are similar to how a human would make them. We chose the logic programming language Prolog to implement this system because the logical representation is a natural choice for a theorem prover, and because the language has a powerful reasoning engine. Prolog employs an unbounded depth-first search strategy in its proof search, and the search space of natural deduction proofs is infinite; to combat this issue, the prover uses iterative deepening to arrive at a shortest proof. Several heuristics are put into place to cut unneeded branches of the search tree and thus improve the performance of the system. Lastly, a parsing system that uses Prolog and Python turns the proof into LaTeX code to display the full proof in Fitch format, and provides other functionalities for input and output. The theorem prover was tested with 9 exercises in natural deduction from *Introduction to Logic* examinations, ranging from 5 to 9 proof steps needed to solve. The prover was able to solve 6 out of the 9 exercises within a time limit of 10 minutes per proof. The prover was not able to prove 3 out of these 9 exercises because of reasons other than computational limitations; this suggests an incompleteness in the search strategy of the system.

1 Introduction

Automated theorem proving – proving logical theorems in an automated fashion – is a branch of artificial intelligence that has been developing since the advent of modern computing.

The course *Introduction to Logic* at the University of Groningen teaches natural deduction to first-year bachelor students in a style adapted from the format created by Frederic Fitch (see Section 2). The examinations of this course contain exercises in natural deduction, which form the direct motivation for this research: we would like to be able to automatically generate model answers for these examinations using a theorem prover.

The goal of this research is to develop a theorem prover for propositional logic that uses natural deduction to find formal proofs that are similar to how a human would make them (i.e. *natural* natural deduction). Our 'natural natural deduction' system should output proofs in LaTeX code, such that these proofs could be used in the model answers for examinations of *Introduction to Logic*.

There are a number of different proof systems for propositional logic, any of which could be implemented in an automated fashion. However, some proof systems are more practically suited for automated theorem-proving than others. Tableaux, for instance, are an attractive choice, because they adhere to the *subformula property*. The subformula property states that any application of a rule in the proof results in a conclusion that is a subformula of its premises (Bolotov, Bocharov, Gorchakov, & Shangin, 2005). Because of this subformula property, the search space of tableaux is finite: one may only break down formulas, and there are finitely many ways to break down a formula.

Natural deduction proofs on the other hand do *not* adhere to this subformula property, which is highlighted by Bolotov et al. (2005) as one of the challenges of implementing a natural deduction theorem prover. Almost[1] all of the introduction rules of natural deduction allow one to introduce a new 'compound' formula in the proof. Because of this, the conclusion to an application of one of these introduction rules is not a subformula of the premises (but rather a superformula), therefore violating the subformula property. The search space for natural deduction proofs is infinite because of these introduction rules. Fitting (1996) even goes as far to say that natural deduction is "inappropriate for automated theorem proving. (...) Heuristics are a necessity, not a nicety.".

Despite the disadvantages of using natural deduction for automated theorem-proving, Pelletier (1998) has been succesful in designing an efficient

[1] $\bot$ Intro is the exception.

theorem prover for first-order logic using natural deduction. Their 'THINKER' system, implemented in Spitbol and later in C, uses the Fitch format as described by Kalish, Montague, and Mar (1980) (an introductory logic textbook for university students), and is designed such that the proofs mimic those of 'a good student'. Pelletier makes a distinction between direct and indirect methods of theorem-proving depending on which method the theorem prover uses internally. A direct theorem prover for natural deduction would internally represent and solve the proofs using natural deduction, whereas an indirect theorem prover might prove the theorem using some other representation internally, and then convert this internal proof to a natural deduction proof afterwards. THINKER has a post-processor which removes logically unnecessary lines from the final proof. Internally, the system uses a goal stack to keep track of subgoals; initially, this goal stack contains only the conclusion of the proof. Moreover, it keeps a list of the preceding lines that are available at this point in the proof; initially this list contains the premises of the proof. Proven lines and subproofs are added to a final proof. In proving a goal, the prover continually either sets a new subgoal based on heuristics, or starts a new subproof.

Other than the "traditional" proof systems such as tableaux and natural deduction, the logic programming language Prolog (see Section 3) and its inference system may also be used as the basis for automated theorem-proving. The programming language may be used 'directly' as a representation language, i.e. as not only the metalanguage but also the object language; Prolog may also be used only as the metalanguage for a theorem prover.

Stickel (1992) developed a theorem prover for first-order logic that uses Prolog as its object language. This theorem prover necessitates translating the premises and conclusion to specific normal forms and skolemizing them and in general restricting the format of the formulas, such that they can be represented as Horn clauses in Prolog. These formulas are then translated into Prolog clauses which are solved by Prolog. Some modifications and extensions to Prolog's reasoning mechanism are made to ensure that the system is sound, and depth-bounded search is implemented such that the system is complete. Moreover, some changes are made to the reasoning mechanism to support non-Horn clauses.

Beckert and Posegga (1995) developed a tableau-based theorem prover in Prolog. Like the other Prolog theorem provers mentioned before, it uses Prolog as the object language and it relies on normal forms of formulas as input. In this case though, the reliance on skolemized negation normal form is merely for simplicity's sake and can easily be extended to all first-order logical formulas. Their reliance on skolemized negation normal form enables a 'lean' system in which not all tableau rules need to be implemented. As with the theorem prover by Stickel (1992), completeness is achieved through using depth-bounded depth-first search rather than Prolog's own unbounded depth-first search.

For our theorem prover, we would like to use Prolog's reasoning engine as a basis. However, we will not use Prolog as the object language for this system following the initial motivation for this research: unlike the previous research mentioned before, we are interested in the full proof that results from a theorem prover, and not just a simple `true` or `false`. Like Pelletier (1998), our theorem prover should be a direct theorem prover, in the sense that it will generate natural deduction proofs and also internally reason using the natural deduction rules. In constrast to Pelletier's THINKER, the system should use the Fitch format as detailed in Section 2 rather than that of Kalish et al. (1980), which is considerably different.

This paper explains the workings of the natural natural deduction theorem prover and motivates the choices made in the design of the system. First, Section 2 gives a brief description of natural deduction and the Fitch format. We also establish some notational conventions. Section 3 serves as an introduction to Prolog and explains the Prolog concepts necessary for understanding the program, as well as elaborating on the choice for Prolog for implementing this system. Section 4 gives an in-depth explanation of the inner workings of the program. Moreover, it outlines and motivates certain design decisions. In Section 5 we evaluate the performance of the system by testing it with a number of queries. Finally in Section 6, we reflect upon this project and consider topics for further research.

The full code of the natural natural deduction system can be found at `https://github.com/flijnzaad/natural-natural-deduction`.

2 Natural deduction

Natural deduction is a syntactic proof method that is meant to mirror our own informal (hence *natural*) reasoning strategies. It was created by Gentzen and by Jaśkowski in the 1930s. For a brief history of natural deduction, see Pelletier (1999).

Since the purpose of this project is to automatically produce model answers for the *Introduction to Logic* examinations, our theorem prover and this paper will adhere to the conventions used in the book that is used in that course (Barker-Plummer, Barwise, & Etchemendy, 2011). We will use capital

Operator	Symbol	Prolog code
conjunction	$\wedge$	`and/2`
disjunction	$\vee$	`or/2`
negation	$\neg$	`neg/1`
implication	$\rightarrow$	`if/2`
bi-implication	$\leftrightarrow$	`iff/2`
contradiction	$\bot$	`contra/0`

Table 2.1: The logical operators, their symbols and the Prolog functors (or atom, in the case of $\bot$) that represent them in the natural natural deduction system.

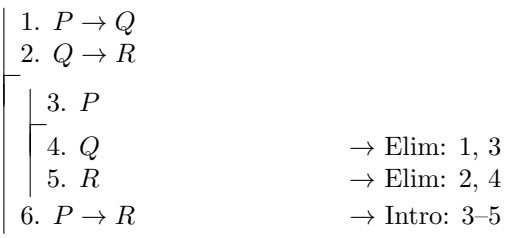

Figure 2.2: A natural deduction proof in the Fitch system that proves the transitivity of implication using a subproof.

letters to represent propositional atoms, and we will use the symbols in Table 2.1 for the operators.

In the theorem prover, we will use natural deduction in the notational system created by Frederic Fitch (Fitch, 1952) with slight modifications by Barker-Plummer et al. (2011). See Figure 2.2 for an example of a natural deduction proof in the Fitch system[2]. In a Fitch-style formal proof, the lines of the proof are numbered and displayed as a list next to a vertical line. The premises of the proof are displayed above a horizontal bar (the "Fitch bar") in the proof. Each line of the proof is justified by the use of rules that introduce or eliminate the connectives in Table 2.1. In other words, there is a $\otimes$ Intro and $\otimes$ Elim rule for each $\otimes \in \{\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \bot\}$. Moreover, one may reiterate a line using Reit as the justification. Each justification cites the line numbers of the relevant lines. Certain introduction and elimination rules require the use of *subproofs* to make temporary assumptions which are discharged after the subproof. Subproofs are proofs within a proof and are indicated by an indent, vertical line and the Fitch bar beneath the premises. For example, Figure 2.2 uses a subproof in order to prove $P \rightarrow R$ using the $\rightarrow$ Intro rule.

[2]This proof has been generated by the natural natural deduction system; see Listing F.1 and Listing F.2 for the code.

3 Prolog

Prolog (short for *programming with logic*) is a high-level logic programming language based on first-order logic. It is used by making declarative programs known as *knowledge bases*, about which the user can ask the system questions, known as queries. Based on the declarative facts and rules one writes, Prolog is able to make logical deductions and return an answer to the user's query.

This implementation of a reasoning engine makes Prolog an attractive language and a natural choice for developing a theorem prover: we write a knowledge base containing the basic natural deduction inference rules, and a recursive rule that 'links' these inference rules together. We then query this knowledge base with a set of premises and a conclusion, and the system should produce a full proof for the argument.

There is a great number of different implementations of the Prolog reasoning engine. For this project, we are using the free Prolog implementation SWI-Prolog (Wielemaker, Schrijvers, Triska, & Lager, 2012). However, the explanations in this section are not particular to a specific Prolog interpreter. Moreover, the natural natural deduction system should work with any ISO-compliant Prolog interpreter. See Wikipedia contributors (2021) for a thorough overview of Prolog implementations available to this date.

Unlike the Prolog theorem provers by Stickel (1992) and Beckert and Posegga (1995) mentioned in Section 1, this theorem prover will not use Prolog directly as a representation language for formulas: it will merely use Prolog's syntax indirectly to represent the formulas. In other words, the object language and metalanguage of this system are separated. See Section 4.1 for an overview of the object language.

This section shall serve as a short introduction to Prolog, introducing only the concepts and terminology necessary for understanding the natural natural deduction system. It is based on Blackburn, Bos, and Striegnitz (2011) and Wielemaker and contributors (2021).

3.1 Syntax

Terms, variables and functors Prolog syntax has four different kinds of *terms*: atoms, numbers, variables and complex terms. Atoms are atomic terms; they start with a lowercase letter. Variables in Prolog start with an uppercase letter or with an underscore. Variables can take any term as their value. The anonymous variable `_` (underscore) is used when one is not interested in the value that Prolog gives to the variable; important to note here is that each occurrence of the anonymous variable is independent, i.e. they do not (necessarily) refer

Listing 3.1 Two examples of disjunction.

```
% disjunction with semicolons
p(X) :- q(X); r(X).

% disjunction with separate rules
p(X) :- q(X).

p(X) :- r(X).
```

to the same thing. Complex terms are composed of functors and terms. A functor is an atom that has a number of arguments that are placed after the functor between parentheses and separated by commas; for example `f(a, b, c)`. The number of arguments of a functor, the *arity*, is placed after the functor when communicating about them, e.g. `f/3`. This is because Prolog does not consider two functors with the same name but different arity to be the same functor, so conventionally the arity is specified. This `functor/n` notation will be used extensively throughout the report.

Facts, predicates and rules Prolog knowledge bases consist of facts and rules, both of which their ending is indicated by a full stop. Facts state things that are unconditionally true. Rules consist of a *head* and a *body*, separated by `:-` which may be interpreted as 'if'. For instance, the rule `p :- q.` may be translated as 'p if q', where p is the head of the rule and q is the body. The body of a rule contains the conditions under which the head of the rule is true. The body of a rule can be a conjunction of clauses, in which case the clauses are separated by commas: for example `p :- q, r.` may be translated as 'p if both q and r'. The body of a rule may also be a disjunction of clauses, indicated with semicolons. However, the preferred method of expressing disjunction according to Blackburn et al. (2011) is to separate the disjunction into multiple rules with the same head where each body contains one of the disjuncts: see Listing 3.1 for an example. We will follow this convention in the natural natural deduction system. Both of the notations are exactly the same in meaning; the second notation is regarded as more legible and clear.

In the body of a rule, one may express negation with `\+`, for example `isOpen(X) :- \+ isClosed(X).`

Throughout this report, we will distinguish a specific subset of functors, namely predicates. A predicate is a functor which is incorporated into a knowledge base as a fact, head of a rule or part of a body of a rule. The difference in Prolog syntax between functors and predicates may be tentatively compared to the difference between functions and predicates in first-order logic, in the sense that functions and functors are terms, while predicates (with instantiated arguments) evaluate to true or false.

Arithmetic As explained earlier, numbers in Prolog are a type of term. Mathematical operators such as $+$ and $-$ are functors in Prolog. Complex arithmetic expressions like `2 + 2` are then complex terms in Prolog syntax: querying `?- X = 2 + 2.` would therefore simply return `X = 2 + 2.`. Prolog has the special predicate `is/2` for *evaluating* arithmetic expressions. Thus to force Prolog to actually compute the value of the arithmetic expression, one should query `?- X is 2 + 2.`, which would return `X = 4.`.

In the natural natural deduction system, we will use the functors `+/2` and `-/2` for building arithmetic expressions, and we will use the Prolog arithmetic comparison predicates `</2` and `>/2` ($<$ and $>$), and `=</2` and `>=/2` ($\leq$ and $\geq$). We will use Prolog's arithmetic features for the calculation of line numbers (see Section 4.1 and Listing A.1) and for controlling the iterative deepening search (see Section 4.2).

Lists In Prolog, a list is a finite sequence of terms. Syntactically a list is specified by surrounding its elements with square brackets: e.g. `[a, b, c, d]` is a list consisting of four elements. The first element of a list is called the *head*, and the rest is the *tail*. For example: the head of the previously mentioned list is `a` and the tail is `[b, c, d]`.

The operator `|` can be used to split a list into its head and its tail like `[H|T]`. In our system, this operator will be used often to prepend a newly deduced proof line to a list of proof lines.

A built-in predicate that is used throughout the program is the `member(X, L)` predicate, which is `true` iff `X` is a member of list `L`. Another built-in predicate that is used in the program, `reverse(X, Y)`, is `true` iff list `Y` contains the same elements as list `X` but in reverse order.

3.2 Querying and proof search

A Prolog knowledge base can be queried by starting the interpreter, loading the knowledge base and typing a query `q(X)` at the interpreter: `?- q(X).` Prolog will then try to find an `X` for which this query holds, based on the facts and rules in the knowledge base.

Unification An important concept in Prolog's proof search strategy is that of *unification*. Two terms unify if they are equal or if the variables in both terms can be instantiated such that the resulting terms are equal; for a more exact definition, see Blackburn et al. (2011, p. 23). For example, `p(1)` and `q(1)` do not unify; `p(X)` and `p(Y)` do unify;

Listing 3.2 A simple knowledge base.

```
p(1).
q(2).
p(X) :- q(X).
```

and `p(q(1))` and `p(_)` also unify. The unification predicate `=/2` is `true` if its two arguments unify; the predicate `\=/2` is `true` if its arguments do *not* unify.

Occurs check In order to keep its unification algorithm efficient, Prolog does not perform an *occurs check*. That is, when attempting to unify two terms of which one is a variable `X`, Prolog does not check whether `X` itself occurs in the other term. Thus, a 'self-referencing' query like `?- X = p(X)` poses a problem: it should return false (since there can be no `X` for which this statement holds), but SWI-Prolog[3] returns `X = p(X)` as the answer to the query. In other words, Prolog is able to make unsound inferences. In order to avoid self-reference problems in a Prolog program, one may explicitly implement such an occurs check in their rules; see Section 4.4. For more information and background on the occurs check, see Blackburn et al. (2011, pp. 28–30).

Search strategy Prolog uses a depth-first backward-chaining search strategy to solve queries (Russell & Norvig, 2009). When a knowledge base is queried, Prolog searches for facts or heads of rules in the knowledge base that can be unified with the query. It does so from the top of the knowledge base to the bottom. If there are multiple matching facts or rules for a goal (a 'choice point'), Prolog will go depth-first into the first branch of the search tree. If a certain branch of the search tree fails, Prolog will backtrack to the last choice point and continue from the open branch there.

When the knowledge base is queried, Prolog will return the first solution it finds to the query. Pressing `;` in the interpreter forces backtracking of the last choice Prolog has made.

As an example, consider Listing 3.2. The query `?- p(X).` would first match with the fact in the first line, returning `X = 1`. When pressing `;`, it would match with the head of the rule. The new subgoal `q(X)` would then match with the second fact, returning `X = 2.`.

Prolog's top-to-bottom search through the knowledge base for matching facts or rule heads will be important for the natural natural deduction system, since it can serve as a way to give priority to certain rules over others.

The cut The predicate `!/0`, known as the 'cut', gives one more control over the search process. By using the cut in the body of a rule, one 'cuts' branches from the search tree by committing Prolog to the choices it has made so far. This way, one can eliminate options that are not useful and possibly make the program more efficient. For an extensive explanation of the cut, see Blackburn et al. (2011, p. 168). For an example of the cut in use, see the listings in Appendix D.

[3] Different Prolog implementations have implemented different behavior in regards to the occurs check.

4 The system

The theorem-proving part of the natural natural deduction system is contained in one large Prolog knowledge base, aided by an additional knowledge base which contains a predicate that implements a certain heuristic. A number of Python functions and some Prolog predicates provide extra in- and output functionalities to the main system.

In Section 4.1 we discuss the object language of the system, i.e. a number of functors and atoms that we define to have a certain meaning within the system. Then Section 4.2 discusses the search strategy we have employed for this system. Section 4.3 introduces the `proves/7` predicate, the main predicate used to find the proofs, and explains each of the arguments it takes. Sections 4.4 and 4.5 explain the different rules used in the main knowledge base. Section 4.6 explains some of the design choices that have been made to make the program more efficient. Finally, Section 4.7 lays out the system for parsing output and input.

4.1 System-specific syntax

In the program, a natural deduction proof is represented by a list of proof lines. A subproof is represented by a list which is (as a whole) a member of its parent proof. A Prolog list is a natural representation of a natural deduction proof since a formal proof, like a list, is a finite sequence of elements. Moreover, subproofs naturally fit in this representation as lists that are member of the main proof.

Each line in a Fitch-style formal proof (see Figure 2.2) has a line number n, a formula F, a justification j and a citation c, in that order. In the system, these elements are combined into one functor `line(N, F, J, C)`, such that all of the information from a proof line is contained in one term. See Listing 4.1 for an example of a full proof in Prolog syntax. It is a Prolog version of the Fitch proof in Figure 2.2 and illustrates the use of the `line/4` functor and the representation of each of its arguments, as well as the nested subproof list. We will now discuss how each of the elements of the line is represented as an argument of the `line/4` functor in Prolog.

Listing 4.1 Example proof in Prolog format, which corresponds to the Fitch proof in Figure 2.2. Newlines and indentation were added manually.

```
[
  line(1, if(p, q), premise, 0),
  line(2, if(q, r), premise, 0),
  [
    line(3, p, premise, 0),
    line(4, q, impElim, two(1, 3)),
    line(5, r, impElim, two(2, 4))
  ],
  line(6, if(p, r), impIntro, sub(3, 5))
]
```

Line number The line number is simply an integer representing the line number of this line. Prolog's arithmetic features (see Section 3.1) can be used to calculate subsequent line numbers. The line number is not only needed for numbering each line, but also for generating the correct citations for each justification.

Formula In the system, a propositional atom P is represented by a lowercase letter `p`, i.e. a Prolog atom. The propositional symbol for contradiction $\bot$ is represented by the atom `contra`. The connectives are defined as functors in Prolog, which may be thought of as prefix versions of the connectives (i.e. $P \land Q$ is represented by `and(p, q)`). See Table 2.1 for an overview of the functors corresponding to the connectives. These functors can be nested to create complex formulas, i.e. complex terms in Prolog. As an example, the sentence $(P \land \neg P) \to \bot$ is represented as `if(and(p, neg(p)), contra)` in the program.

In propositional logic, conjunctions and disjunctions with more than two conjuncts or disjuncts may be expressed without disambiguating parentheses, so $A \land B \land C$ is the same as $(A \land B) \land C$ and $A \land (B \land C)$. In the natural natural deduction system however, conjunction and disjunction will be expressed using `and/2` and `or/2` as binary functors only, for simplicity's and generality's sake.

Justification Justifications of inference steps in a Fitch-style proof are either empty (in the case of a premise), introduction rules, elimination rules, or reiterations. Each of these cases is represented by a Prolog atom in the third argument of `line/4`. An empty justification, i.e. a line being a premise, is represented with the atom `premise`. When an operator is introduced or eliminated, the justification is `Intro` or `Elim`, prepended by an abbreviation of the name of the connective as seen in Table 2.1. For example, $\lor$ Intro is represented as `disjIntro` in the system. The Prolog atom `reit` represents the Reit rule.

Citation A proof line can either cite no other proof lines (in case it is a premise), one line, two lines, or a whole subproof. The number `0` as a citation indicates that the line cites no other steps, i.e. it is a premise. A natural number `i` indicates that the line cites one other step i. When a line cites two other elements[4] i and j, the functor `two(i, j)` is used. When a line cites a subproof $i - j$, the functor `sub(i, j)` is used. When a line cites three elements[4] i, j and k, the functor `three(i, j, k)` is used. For example, the citation for a $\lor$ Elim justification may be `three(1, sub(2, 3), sub(4, 5))`, which corresponds to the justification $\lor$ Elim: 1, 2–3, 4–5.

4.2 Iterative deepening

The search strategy Prolog uses to find answers to queries is not suitable for theorem-proving: "many theorem-proving problems cannot be solved using Prolog's unbounded depth-first search strategy" (Stickel, 1992). In order for the natural natural deduction system to succeed in finding a proof, some kind of bound needs to be imposed on the search. This is because the search space for natural deduction proofs is infinite; we could introduce connectives and start subproofs with new assumptions ad infinitum if we would like to do so.

The benefits of both depth-first search and breadth-first search can be combined in iterative deepening search[5] (Russell & Norvig, 2009, p. 88). In iterative deepening, one performs depth-limited search, repeatedly increasing the depth limit of the search until a solution has been found. In terms of our system: it tries to prove the argument in the least amount of lines possible by repeatedly incrementing the maximum allowed proof length until it has found a proof. This ensures that the system will find a shortest proof, and that the program terminates within finite time, either by returning a proof or returning `false` when a proof could not be found within the maximum proof length. For the purposes of this system (see Section 1), the maximum proof length is set at a static upper bound of `30`.

See Listing 4.2 for a simplified Prolog implementation of the iterative deepening rules with a `proves/3` predicate. The first rule tries to find a proof at the current search depth `MaxDepth` by calling the 'regular' `proves/3` predicate. The second rule is executed once the first rule fails at its goal (since it has then backtracked to this choice point, see Section 3.2). The second rule increments the search depth by 1, checks that the proof length does not exceed the static upper bound, and tries

[4] where an element can be a single proof line or a whole subproof.

[5] sometimes referred to as IDS.

Listing 4.2 Simplified iterative deepening rules.

```
provesIDS(Premises, Line, MaxDepth) :-
    proves(Premises, Line, MaxDepth).

provesIDS(Premises, Line, MaxDepth) :-
    NewDepth is MaxDepth + 1,
    NewDepth < 30,
    provesIDS(Premises, Line, NewDepth).
```

to find a proof at the new depth by recursively calling itself with the new depth passed.

See Listing D.2 for the version of the iterative deepening rule as it is in the final system. Other than the addition of more arguments to pass on to `proves/7`, the difference between the simplified Listing 4.2 and the full implementation in Listing D.2 is in the cut at the end of the first `provesIDS/7` rule. The cut is used here because once the `provesIDS/7` call succeeds, we are not interested in any other option anymore.

4.3 The `proves/7` predicate

The `proves/7` predicate is the most important predicate of the natural natural deduction system, since it enables the system's main proving capabilities. See Listing 4.3 for a short overview of the arguments it takes. We will now discuss each of these arguments and the role it plays in proving theorems.

Listing 4.3 The `proves/7` predicate and its arguments.

```
proves(ProofLines, Available, Line,
       End, NewAvailable,
       MaxDepth, Connectives)
```

An informal reading of this predicate would be the following: The list of proof lines `ProofLines` and available proof lines `Available` together form a shortest proof for the line `Line`, which results in the proof `End` and the new available list `NewAvailable` within the maximum search depth `MaxDepth`.

`ProofLines` This variable contains the previous proof lines of the current proof, which may be either the main proof or a subproof. Important to note here is that the order of the lines in this list is reversed, since Prolog 'prefers' operations to the head of a list; see Section 4.5.

`Available` This variable contains all lines of the proof that are available at this current moment. This variable is different from `ProofLines` in case the proof is a subproof. As mentioned before, a subproof is represented as a list within the proof list. Consider lines 3–5 of the proof in Figure 2.2. It is a $\rightarrow$ Intro subproof that starts with premise P and needs to reach conclusion R. This subproof can be proven using the same `proves/7` rules as the 'regular' proving system; the premise P will be the initial `ProofLines` and the conclusion R will be the `Line` of this 'inner `proves/7` call'. Since the list of `ProofLines` is necessary for returning the full proof in the end, this list may only consist of the lines of the subproof itself here. However, according to the Fitch system all previous lines[6] of the proof are also available to the subproof to make inferences from. `Available` is thus a superset of `ProofLines`. For example: at Figure 2.2 line 5, the `ProofLines` list consists of only lines 3 and 4, whereas the `Available` list consists not only of lines 3 and 4 but also lines 1 and 2.

`Line` This variable contains the line that has to be proven. When the program enters recursion, this variable contains the conclusion of the proof; once the program goes into deeper levels of recursion, `Line` will contain the current subgoal.

`End` This is the variable that the original query's variable will be unified with; this will contain the full final proof once the program is done. This variable is needed to keep things separated from the variable(s) that are used to 'build up' the proof.

`NewAvailable` Similar to the fact that `ProofLines` and `End` need to be different variables, `Available` and `NewAvailable` also need to be two different variables so as to keep the 'building' list and the final list separate.

`MaximumDepth` This argument contains the current search depth limit of the iterative deepening search (see Section 4.2). It represents the maximum allowed length of the proof at the current iteration.

`Connective` This last argument of `proves/7` contains a list of all connectives that appear in the formulas of the premises and conclusion of the outermost proof. This is part of the implementation of a heuristic: see Section 4.6.

4.4 The base case rules

The `proves/7` predicate has two types of rules: it has a number of base cases, and a recursive case. Each of the base cases corresponds to a justification rule in the Fitch system. See Figure 4.4 for a comparison of a Fitch-system rule and the corresponding (simplified) base case of the Prolog system.

One could read the Prolog rule in Figure 4.4b as "The line $A \wedge B$ can be proven by conjunction introduction from premises Σ if line A is present in

[6] Except for individual lines that are within a separate subproof.

i.	A	
j.	B	
k.	$A \wedge B$	$\wedge$ Intro: i, j

(a) The $\wedge$ Intro rule in Fitch format.

```
proves(Sigma, line(and(A, B), conjIntro)) :-
    member(line(A, _), Sigma),
    member(line(B, _), Sigma).
```

(b) The simplified `conjIntro` rule in Prolog format.

Figure 4.4: The conjunction elimination rule in two formats.

Σ and line B is present in Σ." Note here the usage of the anonymous variable, since we are looking for proof lines containing formulas A and B, but we do not care about their justifications.

This simplified base case rule for $\wedge$ Intro is missing a number of elements when comparing it to the full implementation of this base case, as seen in Appendix B. First of all, it is missing many of the arguments of `proves/7`. In the full rule, the line that is proven by the rule, `Line`, is added to `ProofLines` to form the `End` list and to `Available` to form the `NewAvailable` list. Moreover, the line numbers of the two lines to be cited are combined in the `two/2` functor in the last argument of the `line/4` functor of the `Line` variable. For all Intro rules a heuristic is implemented in the base case (see Section 4.6), and an occurs check is manually performed to avoid self-referencing formulas (see Section 3.2). Lastly, the line number of `Line` itself needs to be calculated. This is achieved using the `nextLineNumber/2` predicate. It would be undesirable to use the built-in predicate `length/2` to calculate the length of the proof list and thus the next line number. This is because a subproof is only one element of the list in Prolog, but the list members of a nested subproof need to be counted individually in order to arrive at a correct line number. See Appendix A for the implementation of the `nextLineNumber/2` predicate.

For symmetric rules, such as $\wedge$ Elim (where both A and B may be derived from $A \wedge B$) and $\vee$ Intro (where $A \vee B$ may be derived from either A or B), each rule needs to be implemented separately.

As mentioned in Section 4.1, in this system conjunction and disjunction are only expressed using binary functors `and/2` and `or/2`. Hence, the $\wedge$ Intro rule can only introduce a conjunction of two conjuncts; the same holds for the $\vee$ Intro rule. This may reduce the 'naturality' of the proofs, since for example Barker-Plummer et al. (2011) teaches students rules for 'general' $\wedge$ Intro and $\vee$ Intro with n conjuncts or disjuncts. Here simplicity is favored above naturality.

4.4.1 Subproofs

Some introduction and elimination rules require a subproof: $\neg$ Intro and $\rightarrow$ Intro require one subproof, $\vee$ Elim and $\leftrightarrow$ Intro require two subproofs. See Figure 2.2: the $\rightarrow$ Intro rule applied in line 6 requires a subproof that starts with the antecedent and ends with the consequent of the implication. See Listing C.1 for the implementation of the $\rightarrow$ Intro base case rule. It states that you may conclude `if(X, Y)` if there is a subproof with premise `X` and conclusion `Y`. The general `subproof/12` predicate then takes care of proving the subproof in question using `proves/7`, generating the relevant lists and instantiating the line numbers.

Proof by contradiction is not an explicit rule allowed in the Fitch system: to prove A by contradiction one should start a subproof that starts with $\neg A$ and ends in $\bot$, conclude $\neg\neg A$ from this subproof using $\neg$ Intro and get rid of the double negation there using $\neg$ Elim. However, in the natural natural deduction system this rule needs to be implemented explicitly as a base case in order for the system to be able to prove something by contradiction. This is because of the forward-searching strategy of the system; the system can only prove A from $\neg\neg A$ using $\neg$ Elim if $\neg\neg A$ is a line that already appears in the previous proof lines. This limitation caused by the search strategy actually poses a more general problem for the system, see Sections 5 and 6. Because of this issue, if the system does not have this explicit proof-by-contradiction base case, it returns `false` when one for example queries the law of excluded middle $A \vee \neg A$, which requires a proof by contradiction. Because of this explicit implementation of the proof by contradiction rule, the natural natural deduction system is not a fully 'direct' theorem prover (see Section 1), since it internally uses a rule that is not part of the Fitch system.

4.5 The 'transitivity' rule

In order to arrive at a proof, these base cases need to be accompanied by a recursive rule. The recursive rule implements a property of provability which we will, for a lack of a better word, call the 'transitivity' of provability: Consider a set of formulas Σ and two formulas A and B. Then we have:

$$\text{If } \Sigma \vdash B \text{ and } \Sigma \cup \{B\} \vdash A, \text{ then } \Sigma \vdash A.$$

Or in more Prolog-oriented notation:

$$\Sigma \vdash A \text{ if both } \Sigma \vdash B \text{ and } \Sigma \cup \{B\} \vdash A.$$

Note that this property is strictly speaking not the same as transitivity, since formula B *added* to the set of premises entails A, not by itself.

Listing 4.5 Simplified 'transitivity' rule.

```
proves(Sigma, A) :-
    proves(Sigma, B),
    proves([B|Sigma], A).
```

See Listing 4.5 for a very naive implementation of this rule in Prolog syntax. It closely resembles the second 'formulation' of the transitivity property.

See Appendix D for the final implementation of the transitivity rule. There are a number of additional elements that the naive transitivity rule lacks. First of all, before any recursive calls are made, the transitivity rule checks that the current search depth of iterative deepening has not been exceeded. It does so by calculating the current line number and checking it does not exceed `MaxDepth`. Moreover, as we have seen in Section 4.4 and Appendix B, the proven line `LineY` in this rule will have been added to what are referred to as `NewP` and `NewA` in the transitivity rule. These two variables contain lists of proof lines that will serve as the first and second argument (so `ProofLines` and `Available`) of the second recursive call. In the final transitivity rule there is twice a check that ensures that `LineX` and `LineY` are both fully instantiated, i.e., they contain no variables. There is also a heuristic applied inbetween the recursive calls, see Section 4.6. Lastly, after the second recursive call, there is a cut (see Section 3.2). This is because we are only interested in one proof, so once the second `proves/7` call has succeeded, any other options (i.e. branches of the search tree) are irrelevant for our purpose.

Prolog 'prefers' list operations that modify the first element of a list rather than the last (see Section 3.1, 'Lists'): the list operator `|` operates on the head i.e. the first element of the list. To perform modifications to the last element of a list, such as adding an element to the back of the list, one would need to recurse down the full list. Therefore, it is most computationally efficient and convenient to add new proof lines to the front of the list rather than to the back. Thus, the desired proof lines $(\ell_1, \ell_2, \ldots, \ell_n)$ will result in reversed order when calling `provesIDS/6`: $(\ell_n, \ell_{n-1}, \ldots, \ell_1)$. The wrapper predicate `provesWrap/4` takes care of this issue. It first reverses the premises $(p_1, p_2, \ldots, p_n)$ such that they also appear in 'descending order' $(p_n, p_{n-1}, \ldots, p_1)$ in the list. Then it calls `provesIDS/6` with an initial search depth of the number of premises, and finally it reverses the final proof that results from the `provesIDS/6` call, such that it is in the correct order again once it is returned to the user.

4.6 Heuristics and design choices

To optimize the system, we implemented some heuristics. These heuristics are motivated not only by a desire to decrease the running time of the system, but also by a desire for the system to produce proofs that are similar to the proofs a human would produce. These heuristics are natural in the sense that humans consciously or unconsciously apply these as well when making natural deduction proofs: some of these heuristics are even explicitly taught in the *Introduction to Logic* course discussed in Section 1.

Do not 'reprove' lines In the transitivity rule (see Appendix D), we added a check that ensures that the system does not prove lines that have already been proven. This is realized by checking that the formula-to-be-proven `Y` is not a member of the list of available lines: `\+ member(line(_, Y, _, _), Available)`. This condition is not strictly necessary for the program to produce an efficient proof: the system uses iterative deepening, so it will only produce the shortest proofs, and a proof with repeating lines will never be a shortest proof. However, the addition of this condition drastically reduces the number of inferences made and thus the running time of the program with 60-80%, depending on the query. Moreover, it is a very natural heuristic; a human efficiently solving a formal proof would never prove a line they had already proven.

We should note here that this excludes the case where a line that has already been proven is used as a premise of a subproof. This is not necessarily inefficient proof-solving and it is sometimes required, even, to arrive at a proof: for example, one can only prove $A \to (A \to A)$ by starting a (nested) subproof with a premise that already appeared as a line in the proof. The heuristic is not applied to premises of subproofs, since these are decided based on which rule will eventually use the subproof; see Section 4.4.1. Another case which is excluded from this heuristic is reiteration as conclusion of a subproof. For example, a proof for $A \to A$ needs a subproof with premise A and as conclusion a reiteration of A in order to use the $\to$ Intro rule to arrive at the final conclusion. The system is still able to find a proof in such situations despite the implemented heuristic. This is because reiteration is only strictly necessary in the conclusion of a subproof which has no other proof steps; in any other use case, the 'original' line can just be used. In the case of such a subproof that starts with a line A and ends with the same line A, the system will not enter recursion using the transitivity rule, since it can be proven with the Reit base case. Therefore it will not 'encounter' the aforementioned heuristic.

Restrict introduction of connectives In each

of the Intro base case rules, we added a condition that the system only tries to introduce connectives that appear either in the premises or in the conclusion. This is also a natural and human-like heuristic, since a shortest proof never contains any connectives that do not appear in the premises or conclusion. An exception would be negation, since one may prove theorems using proof by contradiction. In that case, negation might not appear in either the premises or conclusion, but still ¬ Intro should be used in the proof. However, proof by contradiction is implemented as an explicit rule (see Section 4.4.1) in the system, so this heuristic does not apply here.

To only introduce connectives that appear in the premises or conclusion, the `provesWrap/3` predicate calls on a `connectives/3` predicate that returns a list of the connectives that appear in either the premises or the conclusion of the query. This list is then passed on to the `proves/7` predicate to be used in each introduction rule base case[7]: see Listing B.1 for the implementation. We consider it to be suboptimal that this heuristic requires an extra argument to be added to the `proves/7` predicate. This choice is practically motivated however, since calculating the list of 'relevant' connectives at the start and then passing it along is considerably more efficient computation-wise than recomputing this list every time; the latter would require the system to make two to four times as many inferences as the former.

Order of base case rules Another important heuristic is the order of the base cases (i.e. the rules of inference) in the knowledge base. As explained in Section 3.2, Prolog tries to unify clauses from top to bottom. Therefore, the rules that are higher up in the knowledge base will get 'priority'. The ⊥ Intro and ⊥ Elim rules are moved to the top of the knowledge base, because contradictions have a high priority when humans prove theorems. A contradiction should be introduced once it is possible to do so. Moreover, one may prove any sentence using contradiction elimination (*ex falso quodlibet*), so this powerful rule takes high priority as well.

4.7 Input and output

Once the system has derived a proof for some query, it should display this proof to the user in the Fitch format as in Figure 2.2. In order to do that, the Prolog output as detailed in Section 4.1 (see also Listing 4.1 for an example) is converted to LaTeX code. Optionally, the program combines this code with a preamble and compiles it into a PDF.

See Figure 4.6 for a general overview of how the parsing and file-building by the system works. Parsing the Prolog output is handled by the Prolog

[7] Except for ⊥ Intro.

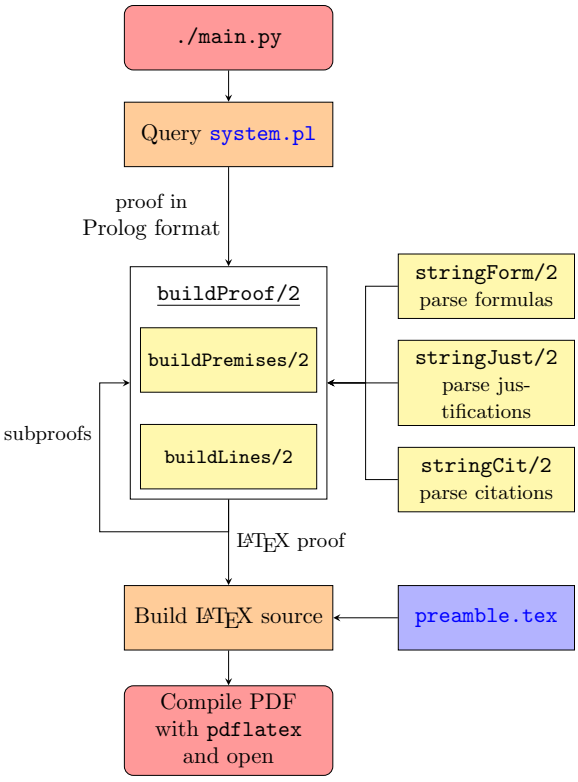

Figure 4.6: A schematic representation of the parsing and LaTeX output system.

predicate `buildProof/2`, which draws on a number of auxiliary predicates that handle the parsing of individual elements of the proof lines. `stringForm/2` parses the formulas from Prolog's prefix notation to the regular infix notation using recursion.

In order to easily handle input arguments and perform file operations, a Python front-end is employed. The PySwip library (Tekol & contributors, 2020) is used to query Prolog knowledge bases from within the Python program.

The Python front-end has a number of command-line options with which the user may specify of which query they would like to see the proof. Moreover, there are a number of options for the format of the output. See Appendix E for an overview of all command-line options.

See Listing F.2 for a full example of the usage of the front-end with a pre-existing example query, and the proof it produces.

It is also possible to manually input queries (i.e. premises and a conclusion) into the system. Since formulas in the query need to be in the Prolog format, a system is put into place to parse formulas in regular logical notation, e.g. $(P \wedge \neg P) \rightarrow \bot$, to formulas in the Prolog notation, i.e. `if(and(p, neg(p)), contra)`. See Figure 4.7 for a general overview of how the input system works. We use the PLY module by Beazly and contributors (2020) as the basis for lexing and parsing. The formula in regular logical notation is

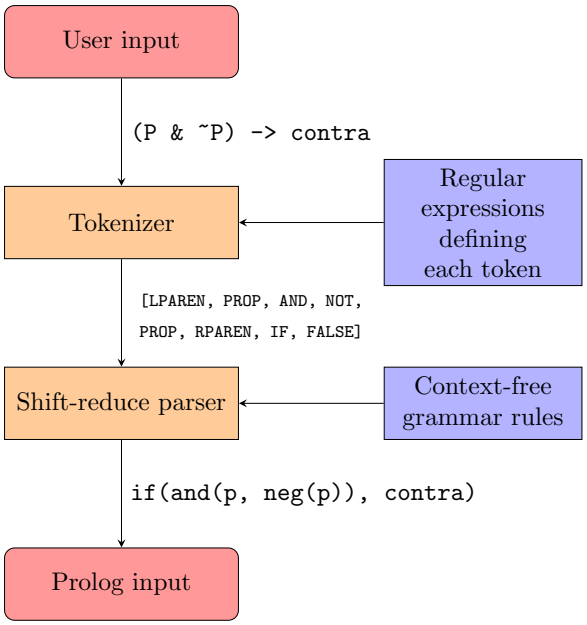

Figure 4.7: A schematic representation of the input system.

tokenized using regular expressions. See Listing E.2 for a complete overview of the options for each token. This list of tokens is then parsed into Prolog code using a shift-reduce parser which recursively converts the regular infix notation to Prolog's prefix notation using context-free grammar rules that define the syntax of the language. Higher precedence is given to the negation operator, following the notational conventions.

Each formula (premise or conclusion) that the user puts in is parsed into the Prolog version of that formula as described above. Then each formula is put into the `line/4` functor, and the premises and conclusion are queried as in Figure 4.6.

See Listing F.3 for a full example of the usage of the input functionality and the proof it produces.

5 Evaluation

The theorem prover was tested with 9 exercises in natural deduction from the homework, midterm, exam and resit of *Introduction to Logic* of the academic year 2020–2021. The amount of proof steps required (obtained by counting the amount of justifications in the proof) for a shortest proof was between 5 and 9. A time limit for the proof search of one query was set at 10 minutes. See Table 5.1 for an overview of the performance of the prover for these 9 queries. The running time of the system is merely given as a rough indication, and does not take into account fluctuations in hardware performance. As can be seen in Table 5.1, the prover was able to solve 6 of the 9 proofs within the set time limit. In the table the number of lines, proof steps, the number of subproofs and the level of nested subproofs is given for every proof. Each of the proofs that resulted from the system was exactly the same as the proof supplied by the model answers for the exercise, except for query 5. The system proved this query in one line less than the model answers did, which still resulted in a natural-looking proof. See Listing F.3 for the output the system gives when queried for this proof.

Important to note is that the system failed to find the proof for query 9 within the needed search depth of 8 and tried going deeper into iterative deepening, reaching search depth 11 before it timed out at 10 minutes. We hypothesize that queries 3, 8 and 9 will never be able to be proven by the system as it is at this moment. This is because of the problems with the search strategy briefly highlighted in Section 4.4.1. The system is only able to either eliminate connectives from *existing* proof lines, or introduce connectives in the conclusion or current (sub)goal. This is because the base cases (including the ones that use subproofs) only work with forward search. Therefore, the system will never start subproofs that do not directly result in proving the conclusion of a (sub)proof. This is exactly what sets apart the proofs for queries 3, 8 and 9 from the other ones: these three queries require subproofs that do not immediately precede the conclusion of a (sub)proof.

Listing 5.2 Example query to check a valid proof.

```
?- provesWrap(
 [line(1, if(p, q), premise, 0),
  line(2, if(q, r), premise, 0)],

  line(_, if(p, r), _, _),

 [line(1, if(p, q), premise, 0),
  line(2, if(q, r), premise, 0),
   [line(3, p, premise, 0),
    line(4, q, impElim, two(1, 3)),
    line(5, r, impElim, two(2, 4))
   ],
  line(6, if(p, r), impIntro, sub(3, 5))
 ]
).
Trying search depth 3...
true.
```

5.1 Checking existing proofs

This system can not only be used for automated theorem proving, it can also be used to check the validity of an existing natural deduction proof. See Listing 5.2 for an example of a valid proof: when presented with a list of premises in the first argu-

	Query	Lines	Rules	Subproofs	Subproof level	Time
1	$(\neg P \lor Q) \to P \vdash P$	7	5	1	0	0.03 sec
2	$(A \land B) \lor (\neg B \land C) \vdash A \lor C$	8	5	2	0	2.2 sec
3	$A \to \neg B, \neg A \to \neg C \vdash \neg(B \land C)$	12	8	2	1	$>$ 10 min
4	$\vdash (A \to \bot) \leftrightarrow (A \to \neg A)$	11	7	4	1	0.05 sec
5	$A \lor B, A \to \neg C \vdash \neg B \to \neg C$	10	5	3	1	0.2 sec
6	$\vdash (A \land B) \to ((A \to \neg B) \to \bot)$	8	6	2	1	0.08 sec
7	$\neg(P \lor Q) \to P \vdash P \lor Q$	7	5	1	0	0.03 sec
8	$\vdash ((P \lor \neg P) \to Q) \to Q$	12	9	3	2	$>$ 10 min
9	$A \to \neg A, (\neg A \lor B) \to C \vdash C$	8	5	1	0	N/A

Table 5.1: Performance of the natural natural deduction system for subexercises of question 3 of homework 1, the midterm, the exam and the resit of *Introduction to Logic* in 2020–2021. Metrics given are the total number of lines needed for a shortest proof, the number of rule applications needed for this proof, the amount of subproofs the proof contains and the deepest level of nested subproof (e.g. a subproof would have level 0, a subproof within a subproof would have level 1, etc.). For each proof, the time needed to arrive at a full proof is given, except for query 9. This query went too deep into iterative deepening, indicating that it was not able to find the proof.

ment of `provesWrap/3`, a conclusion in the second argument, and a full valid *shortest* proof for these premises and conclusion in the third argument, the system returns `true`. See Listing 5.3 for an example of a non-shortest proof which return `false`. See Listing 5.4 for an example of an invalid proof which returns `false`. To check any (non-shortest) proof, one could define a new wrapper predicate for `proves/7` that does not use iterative deepening; that way, the system would not return `false` if the proof was not the shortest proof that exists.

This way, the system could also be used to fill in any information in the proof: for instance, one could replace each of the justifications in the third argument of `provesWrap/3` (i.e. the list of proof lines) in Listing 5.2 with variables, and Prolog would return the unifications of the variables, thus "filling in" the justifications.

Listing 5.3 Example query to check a valid proof which is not a shortest proof. Line 6 of the proof is not needed.

```
?- provesWrap(
 [line(1, if(p, q), premise, 0),
  line(2, if(q, r), premise, 0)],

  line(_, if(p, r), _, _),

 [line(1, if(p, q), premise, 0),
  line(2, if(q, r), premise, 0),
   [line(3, p, premise, 0),
    line(4, q, impElim, two(1, 3)),
    line(5, r, impElim, two(2, 4)),
    line(6, r, reit, 5)
   ],
  line(7, if(p, r), impIntro, sub(3, 6))
 ]
).
Trying search depth 3...
false.
```

5.2 Checking false queries

In theory, this natural deduction theorem prover could be used to check the validity of an argument. If the prover is not able to find a proof within a certain maximum search depth, then within the practical constraints this project poses on proof length it would be safe to assume that the argument is invalid. Concluding from the performance of the system, it is not practically possible to use this theorem prover for that end. After about 6 rule applications, the prover is in practice not able to go deeper into the search tree, which makes it essentially impossible to reach the "end" (the maximum search depth) of the tree.

Listing 5.4 Example query to check an invalid proof. Line 4 of the proof is an invalid inference step.

```
?- provesWrap(
 [line(1, if(p, q), premise, 0),
  line(2, if(q, r), premise, 0)],

  line(_, if(p, r), _, _),

 [line(1, if(p, q), premise, 0),
  line(2, if(q, r), premise, 0),
   [line(3, p, premise, 0),
    line(4, r, impElim, three(1, 2, 3))
   ],
  line(5, if(p, r), impIntro, sub(3, 4))
 ]
).
Trying search depth 3...
false.
```

6 Conclusion

The aim of this research was to develop a natural deduction theorem prover for propositional logic, which would output formal proofs similar to how humans would make them. We chose Prolog as the programming language for this system. We are content with this choice, as the logical representation was very natural for a theorem prover. As expected, the infinite search space of natural deduction proofs did prove to be an obstacle for our theorem prover; fortunately, iterative deepening turned out to be a simple and practical solution to this problem. The system was able to prove 6 out of 9 testing queries, which we are pleased with; for the practical purpose partly motivating this research, the system is moderately effective. As for the human-likeness of the proofs, the proofs the system produced were identical to those in the model answers written by a human, or different but shorter in which case they were still natural-looking. Unfortunately, the system is not complete, as it was unable to find the proof for three queries for a reason other than computational limitations, most probably rooted in a general flaw in the system's search strategy. Moreover, the system was not efficient enough to find proofs for longer queries (requiring 7 or more rule applications) within a practically feasible amount of time.

Let us reconsider the quote by Fitting (1996) from Section 1: "[natural deduction is] inappropriate for automated theorem proving. (...) Heuristics are a necessity, not a nicety". The experience of developing this theorem prover has confirmed this statement for us. The employed heuristics were in theory not necessary to produce a proof in finite time, but in practice they turned out to play a large role in the system being able to producing proofs of meaningful length within a practically feasible time.

6.1 Future work

To further improve upon this system, one could solve the problems highlighted in Section 5. This may be realized by employing a different search strategy, like for example combining forward and backward search. Moreover, one could add more heuristics to improve the running time of the system.

To further improve the naturalness of the proofs (but reduce the simplicity of the system), one could adapt the introduction and elimination rules for conjunction and disjunction to be more general. This would allow one to for example introduce the conjunction $A \wedge B \wedge C \wedge D$ in one step from previous steps A, B, C and D rather than introducing the conjunction $((A \wedge B) \wedge C) \wedge D$ from A, B, C and D in three steps. This would require representing conjunctions and disjunctions as lists, and changing the base case rules (and citations) accordingly.

Moreover, one could survey the ability of the system to produce human-like proofs by asking participants to rate the proofs the system produces on their naturalness or similarity to the proofs they would produce themselves. It may also be interesting to see whether the theorem prover has more 'trouble' with the proofs that humans (say, students taking *Introduction to Logic*) find more difficult and vice versa. Judging from the prover's performance seen in Table 5.1, there seems to be some connection between the time it takes the system to arrive at a proof and the metrics displayed in the table. However, it does seem that the proof time depends on other properties of the proof as well. For example, the proof for query 2 (see Table 5.1) (which takes 5 'steps' and has 2 subproofs) is proven in about 2 seconds, whereas the proof for query 6 (which needs 1 step more, and needs two nested subproofs instead of two consecutive ones) is proven in less than one twenty-fifth of that time. It may be interesting to see whether these differences in the time needed to solve certains proofs would be similar in humans.

To expand this system for solving first-order theorems, additional functionalities could readily be added. This would require additional system-specific syntax for representing identity, universal quantification and existential quantification. The representation of predicates and constants already fits naturally into Prolog's syntax. Moreover, introduction and elimination rules for each of these operators would have to be implemented. This includes the implementation of a representation

of subproofs with a boxed constant, and a way to keep an overview of which constants are being used already. Naturally, the input/output system would also have to be adapted to accustom the different formulas, rules and boxed subproofs.

It could be proven that this system is sound, such that we are sure that the system will not prove invalid arguments. A soundness proof could be carried out by checking every one of the `proves/7` rules (base cases and transitivity rule) on their soundness.

To generalize the system to a wider context, the static strict bound on the iterative deepening search depth of `30` (see Section 4.2) could be changed to a more dynamic one. In any proof system, the length of the shortest proof of a propositional tautology is exponentially related to the length of the tautology (Cook & Reckhow, 1979). Based on this fact, one may define the bound for iterative deepening search based on the complexity of the premises and conclusion, and the number of premises. This would enable the system to work in a broader context than just the *Introduction to Logic* exams, where the proofs consist of 30 to 35 lines maximum and such a static bound on the search poses no problem.

More research could be done into whether this system, when the problems described in Section 5 are solved, is complete for propositional logic, and if not, what other changes would be necessary to make it complete. A theorem prover that uses iterative deepening search can be complete, see the Prolog theorem provers by Stickel (1992) and Beckert and Posegga (1995) that both use iterative deepening and are proven to be complete.

Acknowledgements

I would like to thank my supervisor Malvin Gattinger for all his help with debugging the code, coming up with ideas, giving feedback on my writings and for always being available for any questions I had.

I would like to thank Joost Doornkamp for his guidance and his help in debugging an early version of the program. I would like to thank Ronald de Haan for his contributions to exploring the possibilities of breadth-first search in Prolog. I would like to thank Kajetan Dvoracek for sharing the Haskell code of his natural deduction theorem prover with me.

References

Barker-Plummer, D., Barwise, J., & Etchemendy, J. (2011). *Language, proof and logic* (2nd ed.). Stanford, CA: CSLI Publications.

Beazly, D. M., & contributors. (2020). *PLY (Python Lex-Yacc) v4.0.* Retrieved from `https://www.github.com/dabeaz/ply`

Beckert, B., & Posegga, J. (1995). lean*TAP*: Lean tableau-based deduction. *Journal of Automated Reasoning*, *15*(3), 339–358. doi: 10.1007/BF00881804

Blackburn, P., Bos, J., & Striegnitz, K. (2011). *Learn Prolog now!* (Vol. 7). Rickmansworth, England: College Publications.

Bolotov, A., Bocharov, V., Gorchakov, A., & Shangin, V. (2005). Automated first order natural deduction. In B. Prasad (Ed.), *Proceedings of the 2nd Indian International Conference on Artificial Intelligence* (pp. 1292–1311). ISCAI. Retrieved from `https://core.ac.uk/display/161114791`

Cook, S. A., & Reckhow, R. A. (1979). The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, *44*(1), 36–50. doi: 10.2307/2273702

Fitch, F. B. (1952). *Symbolic logic: An introduction.* New York, NY: Ronald Press Company.

Fitting, M. (1996). *First-order logic and automated theorem proving* (2nd ed.; D. Gries & O. Hazzan, Eds.). New York, NY: Springer Verlag. doi: 10.1007/978-1-4612-2360-3

Kalish, D., Montague, R., & Mar, G. (1980). *Logic: Techniques of formal reasoning* (2nd ed.). Oxford, England: Oxford University Press.

Pelletier, F. J. (1998). Automated natural deduction in THINKER. *Studia Logica*, *60*(1), 3–43. doi: 10.1023/a:1005035316026

Pelletier, F. J. (1999). A brief history of natural deduction. *History and Philosophy of Logic*, *20*(1), 1–31. doi: 10.1080/014453499298165

Russell, S. J., & Norvig, P. (2009). *Artificial intelligence: A modern approach* (3rd ed.). Prentice Hall.

Stickel, M. E. (1992). A Prolog technology theorem prover: a new exposition and implementation in Prolog. *Theoretical Computer Science*, *104*(1), 109–128. doi: 10.1016/0304-3975(92)90168-F

Tekol, Y., & contributors. (2020). *PySwip v0.2.10.* Retrieved from `https://www.github.com/yuce/pyswip`

Wielemaker, J., & contributors. (2021). SWI-Prolog reference manual [Computer software manual]. Retrieved from `https://www.swi-prolog.org/pldoc/doc_for?object=manual` (version 8.3.23)

Wielemaker, J., Schrijvers, T., Triska, M., & Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, *12*(1-2), 67–96. doi: 10.1017/S1471068411000494

Wikipedia contributors. (2021, April 15th). *Comparison of Prolog implementations —*

Wikipedia, the free encyclopedia. Retrieved from https://en.wikipedia.org/w/index.php?title=Comparison_of_Prolog_implementations&oldid=1017949259

A Appendix: Line number calculations

Listing A.1 Predicates for the calculation of line numbers.

```
% base case: empty list
currentLineNumber([], 0).

% base case: single line
currentLineNumber(line(_, _, _, _), 1).

% recurse through the list
currentLineNumber([H|T], N) :-
    currentLineNumber(H, A),
    currentLineNumber(T, B),
    % add the line numbers of head and tail together
    N is A + B.

% calculate the next line number of the list of proof lines
nextLineNumber(L, N) :-
    currentLineNumber(L, Current),
    N is Current + 1.
```

B Appendix: Example of a 'simple' base case rule

Listing B.1 Base case rule for $\wedge$ Intro.

```
% conjunction introduction:
proves(ProofLines, Available, Line, [Line|ProofLines], [Line|Available], _, C) :-
    % connective heuristic
    member(and, C),
    % the line to prove
    Line = line(Next, and(X, Y), conjIntro, two(N1, N2)),
    member(line(N1, X, _, _), Available),
    member(line(N2, Y, _, _), Available),
    % occurs check: no self-referencing formulas
    X \= and(X, _),
    Y \= and(_, Y),
    % determine the line number of the line we're proving
    nextLineNumber(Available, Next).
```

C Appendix: Example of subproof base case rule

Listing C.1 Subproof base case rule for → Intro.

```
% implication introduction:
proves(ProofLines, Available, Line, End, NewA, D, C) :-
    % connective heuristic
    member(if, C),
    % the line to prove
    Line = line(Next, if(X, Y), impIntro, sub(N1, N2)),
    % occurs check: no self-referencing formulas
    X \= if(X, _),
    % the premise and conclusion of the subproof
    Premise = line(N1, X, premise, 0),
    Concl   = line(N2, Y, _, _),
    % prove the subproof
    subproof(ProofLines, Available, Line, End, NewA,
             Premise, Concl, Next, N1, N2, D, C).
```

Listing C.2 The 'general' `subproof/12` predicate, which solves a subproof, adds this subproof and the line-to-be-proven to the relevant lists and computes the line numbers.

```
% prove a subproof
subproof(ProofLines, Available, Line, End, NewA, Premise, Concl,
         Next, N1, N2, D, C) :-
    % check that search depth is not exceeded
    nextLineNumber(Available, Dnew),
    Dnew =< D,
    % prove the subproof, the full proof is unified with S
    proves([Premise], [Premise|Available], Concl, S, _, D, C),
    % the subproof needs to be in correct 'descending' order
    reverse(S, Subproof),
    % compute the relevant lists of proof lines
    End  = [Line|[Subproof|ProofLines]],
    NewA = [Line|[Subproof|Available]],
    % calculate the line numbers
    nextLineNumber(Available, N1),
    currentLineNumber(NewA, Next),
    N2 is Next - 1.
```

D Appendix: The transitivity rule and iterative deepening

Listing D.1 The transitivity rule.

```
% transitivity:
proves(ProofLines, Available, LineX, End, NewAvailable, MaxDepth) :-
    % the line to prove
    LineX = line(_, Formula, _, _),
    % the Formula to prove should be instantiated
    ground(Formula),
    % don't exceed the current search depth D of *all available lines*
    currentLineNumber(Available, D),
    D =< MaxDepth,
    % derive 1 line (Y) from the premises
    proves(ProofLines, Available, line(_, Y, _, _), NewP, NewA, MaxDepth, C),
    % don't prove a line you already have (heuristic)
    \+ member(line(_, Y, _, _), Available),
    % with LineY added to the premises, derive line X
    proves(NewP, NewA, LineX, End, NewAvailable, MaxDepth), !.
```

Listing D.2 The iterative deepening rules.

```
% try to prove Line at the current search depth
provesIDS(Premises, Available, Line, NewP, NewA, D) :-
    proves(Premises, Available, Line, NewP, NewA, D), !.

% else do iterative deepening
provesIDS(Premises, Available, Line, NewP, NewA, D) :-
    % increment the search depth
    Dnew is D + 1,
    % don't exceed the maximum proof length
    Dnew < 30,
    % print progress statement to the interpreter
    write('Trying search depth '), write(Dnew), writeln('...'),
    % try with the new search depth
    provesIDS(Premises, Available, Line, NewP, NewA, Dnew).
```

E Appendix: Interface usage information

Listing E.1 Usage information of the interface.

```
Usage:
    1) ./main.py               Solve the last example query; produce a tex
                               file including preamble and label the query;
                               compile the tex file and open the pdf
    2) ./main.py [options]     Solve queries and produce/open files according
                               to the options
    3) ./main.py --help        Display this message
    4) ./main.py --in-help     Display the input help
    5) ./main.py --version     Display the system's version
    6) ./main.py --clean       Remove all proof tex and pdf files

Options:
    -a              Solve all example queries in queries.pl
    -q i            Solve only query number i
    -r i,j          Solve queries i until and including j
    -i              Manually input premises and conclusion; see --in-help
                    for instructions and more information
    --tex           Only produce a tex file without preamble or labeled
                    queries, don't compile or open pdf
    --clip          Same as --tex, but copy the LaTeX code to the clipboard
                    instead of producing a tex file
    --cliptex       Same as --clip, but also produce the tex file
    --nolabel       Don't add labels to the queries
```

Listing E.2 Input instructions for the interface.

```
* You have to input formulas in the 'regular' infix notation: for example,
    (P and not P) implies contra
* Use disambiguating parentheses where needed. They are not needed for
  conjunctions or disjunctions of more than 2 arguments.
* You can use the following options for the different tokens:
  - propositional parameters:
      a single uppercase or lowercase letter
  - conjunction:
      and, \land, \wedge, ^, &, &&, ∧
  - disjunction:
      or, \lor, \vee, v, |, ||, ∨
  - negation:
      not, \lnot, \neg, ~, !, ¬
  - implication:
      if, implies, \lif, \to, \rightarrow, ->, >, →
  - bi-implication:
      iff, \liff, \leftrightarrow, \equiv, <->, <=>, ↔
  - contradiction:
      contra, false, \lfalse, \bot, ⊥
  - parentheses:
      ( ) and [ ]
* You can put in the premises one by one at the prompt. When you are done with
  putting in the premises, type a ';' to continue to putting in the conclusion.
```

F Appendix: Example proofs

Listing F.1 Example query 19, which proves the transitivity of implication.

```
q19(X) :-
    Premises = [line(1, if(p, q), premise, 0),
                line(2, if(q, r), premise, 0)],
    Concl    =  line(_, if(p, r), _, _),
    provesWrap(Premises, Concl, X).
```

Listing F.2 Command line in- and output for producing a proof and only its `.tex` source file (`--tex`) of example query 19 (`-q 19`).

```
$ ./main.py -q 19 --tex
Trying search depth 3...
% 128,298 inferences, 0.026 CPU in 0.026 seconds (100% CPU, 4995231 Lips)
Solved!
Succesfully solved the proof(s)
Succesfully built a proof tex file for q19, to be found in q19.tex
$ cat q19.tex
\fitch{
    & 1. $( P \lif Q )$ &  \\
    & 2. $( Q \lif R )$ &
}{
\fitch{
    & 3. $P$ &
}{
    & 4. $Q$ & $\lif$~Elim: 1, 3 \\
    & 5. $R$ & $\lif$~Elim: 2, 4
}
 \\
    & 6. $( P \lif R )$ & $\lif$~Intro: 3--5
}
```

Note here that conventional LaTeX indentation style requires one to indent the subproof from line 10 to 16 for increased legibility. Moreover, the outer parentheses in proof line 1, 2 and 6 are commonly left out, as in Figure 2.2. They are produced by the parsing system to keep said system as simple and general as possible.

Listing F.3 Command line in- and output for producing a proof and only its `.tex` source file (`--tex`) of the manually input (`-i`) exercise 3(c) of the midterm of *Introduction to Logic* in 2020–2021 (query 5 of Table 5.1). Note that there are a number of other options that could be used for the connectives; see Listing E.2 for an overview.

```
$ ./main.py -i --tex
Premise 1: A v B
Premise 2: A -> ~C
Premise 3: ;
Conclusion: ~B -> ~C
Querying the system with this argument:
    Premises:  ['or(a, b)', 'if(a, neg(c))']
    Conclusion:  if(neg(b), neg(c))
Trying search depth 3...
Trying search depth 4...
Trying search depth 5...
Trying search depth 6...
% 1,667,919 inferences, 0.197 CPU in 0.197 seconds (100% CPU, 8484831 Lips)
Solved!
Succesfully solved the proof(s)
Succesfully built a proof tex file for the manually input proof, to be found
in proof_20210623172056.tex
$ cat proof_20210623172056.tex

\fitch{
    & 1. $( A \lor B )$ &  \\
    & 2. $( A \lif \lnot C )$ &
}{
\fitch{
    & 3. $\lnot B$ &
}{
\fitch{
    & 4. $A$ &
}{
    & 5. $\lnot C$ & $\lif$~Elim: 2, 4
}
 \\

\fitch{
    & 6. $B$ &
}{
    & 7. $\lfalse$ & $\lfalse$~Intro: 6, 3 \\
    & 8. $\lnot C$ & $\lfalse$~Elim: 7
}
 \\
    & 9. $\lnot C$ & $\lor$~Elim: 1, 4--5, 6--8
}
 \\
    & 10. $( \lnot B \lif \lnot C )$ & $\lif$~Intro: 3--9
}
```