Final report - Group 56 - Hans de Nivelle

Viktor Kovalchuk Alikhan Balpykov Alina Kontorbayeva

Yuliya Barko Ilnur Gayetov

April 25, 2025

Project Summary

This project aimed to develop a light, easy-to-use proof-checking tool based on Partial Higher-Order Logic with Interfaces (PHOLI) calculus. The main goal was to create a system that automatically validates mathematical proofs and is not difficult to use like the other existing proof assistants. The project involved the design and development of a compact software prototype written in *C++20*.

The system checks logical proofs by parsing formulas, verifying their validity, and presenting the results in human-readable form. Key accomplishments include the successful implementation of the parser, a pretty printer for terms and belief states, and the validation of several theorems.

This work contributes to making formal proof verification more accessible to students and researchers by providing a simple tool that could be used in teaching and applied logic. The system was tested using various examples and proved to be correct. Future work will focus on developing an easy-to-use UI, type checker, and proof checker.

Introduction

Manually checking whether a mathematical or computer science proof is correct can be a difficult and time-consuming task. Even experienced researchers can spend many hours searching for hidden assumptions and small mistakes. However, existing proof software assistants such as *Coq* and *Isabelle* have shown, however, that it is possible to reliably check mathematical proofs automatically [1, 8]. Nevertheless, these systems are challenging to learn and use. As a consequence, students and applied scientists will avoid these formal tools.

That is why we develop a minimal proof calculus together with a compact software prototype:

- Our logical basis is *Partial Higher-Order Logic with Interfaces* (PHOLI) [2]. PHOLI extends ordinary two-valued semantics, where every formula is

either true or false, by adding a third value that we call *error*. This extra value makes it easy to express partial functions and sub-types inside the logic itself, so we do not need to have separate well-formedness conditions. In addition, the complete set of axioms fits on only a few pages. Thus, we believe that PHOLI is well suited for our goal.

- The software prototype is a single command-line programme written in *C++20*. Apart from the standard *C++* library, our system uses only small tools such as *TreeGen* [4] and *Maphoon* [3], so the code has no external dependencies and therefore builds well on common Linux machines.
- In the long run, such a tool could help lecturers who teach automata theory, and researchers and students who need fast proof checks.

The rest of the report is organized as follows:

1. **Section Background and Related Work** reviews prior works and explains why resolution works well with PHOLI.
2. **Section Project Approach** details the software architecture, main algorithms, and the collaboration process of the project team.
3. **Section Project Execution** describes the design decisions made and challenges faced over two semesters of development.
4. **Section Evaluation** evaluates the prototype in terms of meeting expectations.
5. **Section Conclusion** summarises the findings and contributions and outlines directions for future research.

Background and Related Work

PCL and Higher-Order PHOLI

Partial Classical Logic (PCL) is a three-valued first-order system created by de Nivelle to make type conditions explicit [2]. Besides the usual values T and F it contains a third value e that signifies "error." Unlike Kleene's 3-valued logic, which treats the third value as an unknown that may later turn into T or F, PCL requires a formula to be *well-typed* before it can be assumed: ill-typed statements remain illegitimate premises. This strictness prevents accidental reasoning with malformed formulas and removes the need for an external type checker. Logical connectives come in strict and lazy versions so that e propagates only when required. For instance, the lazy implication $\{\} \rightarrow$ ignores its right side if the left side is F, whereas the strict arrow $\rightarrow$ evaluates both arguments. Because types are encoded as predicates, sub-typing and pre-conditions can be written directly in the logic. To model realistic mathematics PCL must handle higher-order notions. De Nivelle introduces a small family of *structural*

types: the object type O, the truth-value type T, user-defined identifiers, and function spaces $V(U_1, \ldots, U_n)$. Complex data such as sequences or automata are packaged as `structs` whose fields are ordinary predicates; field access is translated into function application at parse time.

Comparison with other solutions

Compared with existing formal assistant systems (*Coq*, *Isabelle*), the PHOLI approach keeps the logic simple, eliminates external type-checking, and retains a very small kernel, yet it is more expressive than classical first-order Kleene logic.

PHOLI in detail

PHOLI (Partial Higher-Order Logic with Interfaces) [2] uses 3-valued logic. In contrast with 2-valued logic which has only **T** (True) and **F** (False) values, 3-valued logic introduces a third value - **e** (Error).

Here is the truth table for well-typedness operator #:

# :	
t	
f	
t	

Additionally, with three-valued logic we have two versions of each logical operator. For example, implication has two operators: lazy and strict. Below are the truth tables for them:

{} → :			
t	**t**	**t**
e	**e**	**e**
f	**e**	**t**

→ :		
t	**e**	**t**
e	**e**	**e**
f	**e**	**t**

In the similar way, the conjunction operator is also split to strict and lazy versions:

{} ∧ :			
f	**f**	**f**
e	**e**	**e**
f	**e**	**t**

∧ :		
f	**e**	**f**
e	**e**	**e**
f	**e**	**t**

3-valued logic provides a more flexible way for working with partial functions and dealing with subtypes. 3-valued logic treats types as predicates, which makes it possible to add preconditions to functions, add subtypes, add a single element to a type, etc. However, introduction of 3-valued logic adds complexity to the system.

Project Approach

Project Description

This *C++* project implements parsing and verifies logical formulas and proofs. It includes parser, a pretty printer for human-readable output, and supports

various logical operations and quantifiers.

Use case and Architecture

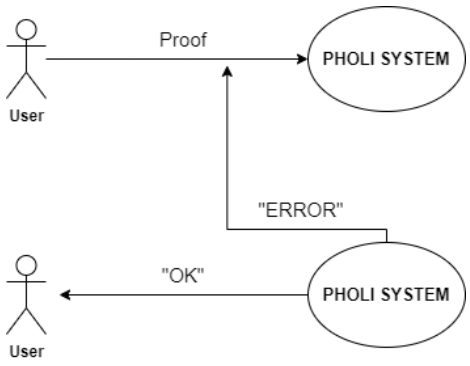

Figure 1: Use case diagram

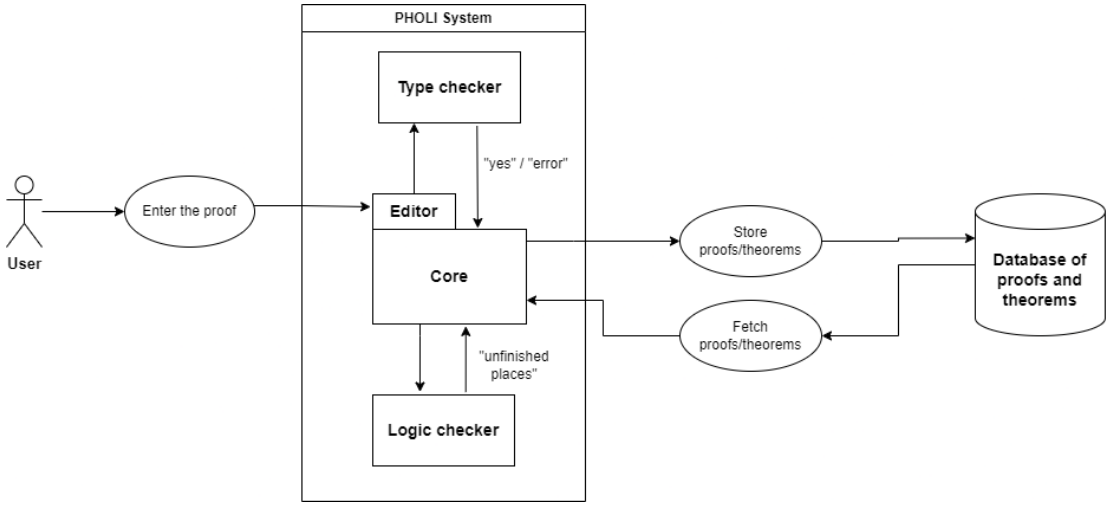

Figure 2: Architecture of the system

External Tools

For effective communication and productive work on our project, we used following tools:

- Piazza [9] - for better communication with our advisor (Prof. Hans de Nivelle).

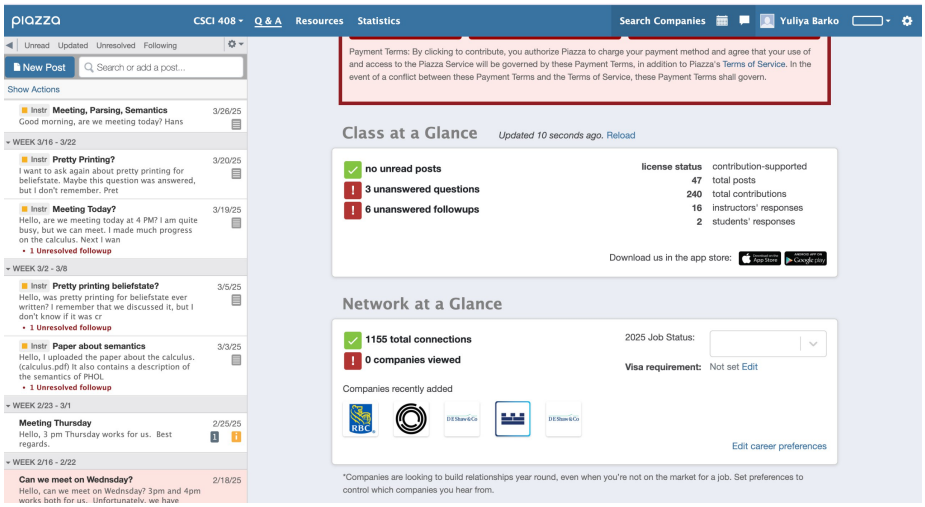

Figure 3: Piazza discussions

- Git [6] and GitHub [7] - for development in general.

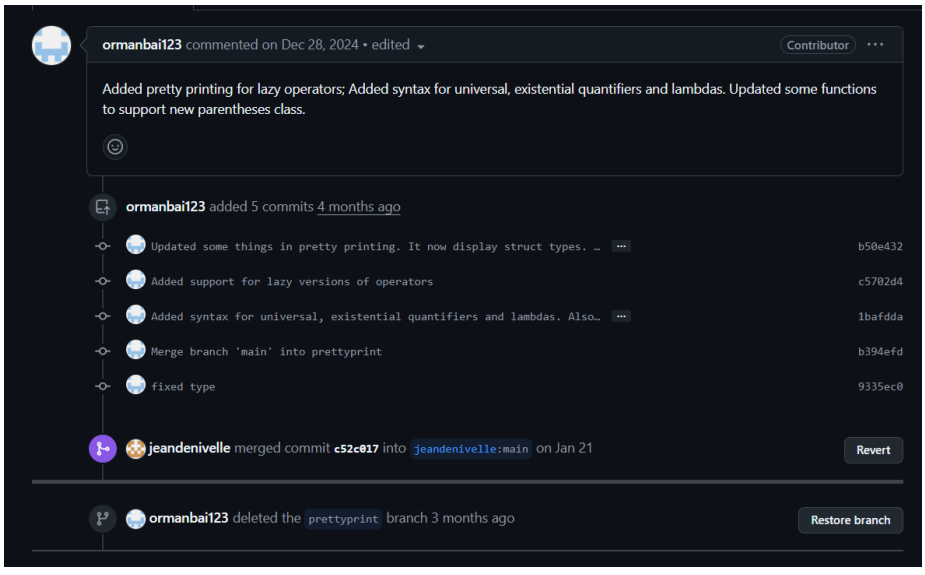

Figure 4: Example of merged pull request for GitHub repo

- TreeGen [4] and Maphoon [3] were used during parser development.

Pretty Printer

The pretty printer is used for printing the types, formulas, definitions and belief-states in human-readable form. All the code related to pretty printing locates inside *logic/pretty.cpp* and *pretty.h*. We provide overloaded functions for printing terms, types, functions, contexts and belief states. Although the task of printing may seem trivial, handling the parentheses and printing them correctly is not that easy. For that, we use *attractions* struct, that assigns "attraction power" or sometimes called "binding power" [10] to each operator to know when to print the parentheses. In particular, we have a *logic::pretty::getattractions* method that returns attraction powers to left and to right of the operator. The code below shows how we define attraction power to some of the operators:

```
logic::pretty::attractions
logic::pretty::getattractions( logic::selector sel )
{
   switch( sel )
   {
   case op_not:
   case op_prop:
      return { 0, 150 };

   case op_and:
   case op_kleene_and:
      return { 140, 141 };

   case op_or:
   case op_kleene_or:
      return { 130, 131 };

   case op_implies:
      return { 121, 120 };
   case op_equiv:
      return { 110, 110 };
   case op_equals:
      return { 160, 160 };
```

Using the function above, we check if parentheses should be printed as follows:

```
void logic::pretty::parentheses::check( attractions attr,
            std::pair< unsigned int, unsigned int > env )
{
   if( env.first && attr.left && env.first >= attr.left )
   {
```

```
        needed = true;
        return;
    }

    if( env.second && attr.right &&
                                    env.second >= attr.right )
    {
        needed = true;
        return;
    }
}
```

We also provide an option to user on how to print the functional types: either in C style, i.e T(O) or in mathematical style O ->T. This is currently controlled by the *csyntax_types* variable at compile-time.

Parser Implementation

The bottom up parser is implemented using the `maphoon` tool. The parser should be in agreement with pretty printer. There are three main objects that must be parsed: **structure**, **definition**, and **term**. Below is the detailed explanation of each:

Structure

Structures have the following form:

```
struct <Name> := <field>:<Type>, ...;
```

Example:

```
struct Seq :=
    0 : O,
    succ : O( O );
```

Definition

Definitions have the following form:

```
def <Name>(<arg:  Type>, ...)  ...  := <Term>;
```

Example:

```
def minhomrel( s1 : Seq, s2: Seq )( x1:O, x2:O ) :=
    [ P: T(O,O) ]{ stricton( prod( s1.gen, s2.gen ), P ) }
      -> homrel( s1, s2, P ) -> P(x1,x2);
```

Term

A **term** is a combination of quantifiers, operators, and objects. Terms are parsed using a set of operators with defined associativity and precedence.

Operators and Associativity

The operators in terms are categorized based on their associativity:

- **Left-associative operators:**
  - Logical AND: `&`
  - Logical OR: `|`
  - Implication: `->`
  - If and only if: `<->`
- **Right-associative operators:**
  - Quantifiers: `FORALL []`, `EXISTS <>`
  - Logical NOT: `!`
  - Prop: `#`
- **Lazy operators:**
  - Lazy AND: {} `&`
  - Lazy OR: {} `|`
  - Lazy Implication: {} `->`

Operator Priorities

The priorities of operators (from highest to lowest) are as follows:

Operator	Priority
`!, #`	Highest
{} `&`	
{} `|`	
{} `->`	
`&`	
`|`	
`->`	
`<->`	Lowest

Lazy Operators Restrictions

Lazy operators have specific restrictions on their usage:

- **Lazy AND** ({} `&`) and **Lazy OR** ({} `|`) can be used only with the `EXISTS` quantifier.
- **Lazy Implication** ({} `->`) can be used only with the `FORALL` quantifier.

When operators have the same precedence, their associativity determines how the expression is evaluated.

Weight Computation for Logical Terms

We have changed an algorithm which was computing a numerical weight for a logical term based on its structure. The purpose of this function is to assign a measure of complexity to each term, which can be used to limit processing time or detect overly large terms during logical operations such as resolution.

To ensure efficiency and avoid unnecessary computation, we have extended the function with a weight limit mechanism. If the cumulative weight of a term exceeds a given threshold during computation, the function stops recursion early and returns the limit, indicating that the term is too large to be processed.

Semantic Checks, Enumeration of Interpretations.

When we talk about truth values(which can be true, false, or error) and objects, we refer to the types defined in Definition 1 [5]. The system also supports the following list of operators:

- unary: not, isdefined
- binary: and, lazy and, or, lazy or, implies, lazy implies, equivalence, equals
- kleene: kleene and, kleene or
- quantifiers: exists, forall, kleene exists, and kleene forall

The system needs to verify whether two logical formulas/functions are equivalent or whether correctness of the first formula implies equivalence with the second formula (i.e., whether $A \preceq B$ holds under all interpretations). This can be seen in Definition 11 and Theorem 12 of [5]. To verify such logical relations, we must evaluate the formulas under all possible interpretations. We consider a logic operator as a function that acts over n arguments, which can be truth values as well as objects (these two are called primtypes in the code). And the result of an interpretation also can be any of these primtypes, which makes this approach generalizable. Hence, we are not limited to the number of arguments of the operator and types of the arguments.

The algorithm of the approach will be summed up in following two paragraphs. We have implemented a data type functions that is parametrized by

argtypes (a vector of pairs, consisting of primitive types and their sizes) and restype (a pair of the primitive type that the function returns and its size). The function data type is able to enumerate all possible functions. Initially, the values vector is initialized all zeroes, after operator ++ can be used to obtain the next function. It increases the last function value, where each entry is bounded above by the size of the result type (restype). If the bound is reached for a given entry, the entry before it will be increased, until the result is again a function that maps all values to zeroes.

We then define interpretation class that has a map of identifiers to the possible functions and vector of functions named valuation. We have then main evaluation function that will switch through the cases of different operators. To optimize evaluation. we introduce the concept of a bottom value. For instance, in two-valued logic with true and false, evaluating a conjunction (AND) does not require evaluating all operands if one is already false, as the entire expression will necessarily evaluate to false. We define a bottom function that, given an operator, returns its worst-case value. In contrast to bottom value, we define top value, that can be as initial starting value for kleene operators. The update to the value is done through a specifically defined merge function that depends on the operator at hand. In case of usual operators, everything is straightforward: the terms breaks into the sub-terms which are themselves evaluated recursively until base cases. If a quantifier is encountered, we evaluate the subformula for all possible interpretations of the quantified variable, so start with function with all values zeros. The function is then incrementally updated using operator++, which systematically enumerates all possible assignments through backtracking. These assignments are managed using an interpretation stack: each new assignment is pushed onto the stack, and evaluation is performed using the interpretation at the top.

Project Execution

Understanding Theory

A considerable amount of time was spent on literature review and understanding of the theory because the project is complex and requires a strong background in logic.

In order to understand the logic, we formalized natural numbers, and proved existence of a recursion operator:

A *Seq* consists of a start element and a successor function:

$$\textbf{struct}\ \text{Seq} := (\ 0{:}\,\mathbf{O};\ \text{succ}{:}\,\mathbf{O}(\mathbf{O})\ )$$

A binary relation P between two Seqs is homomorphic if:

$$\text{homrel}(\ s_1, s_2{:}\text{Seq}\ )(\ P{:}\,\mathbf{T}(\mathbf{O},\mathbf{O})\ ) := \begin{cases} P(\ s_1.0,\ s_2.0\ ) \\ \forall x_1, x_2{:}\,\mathbf{O}\ \{\ \text{reachable}(\ s_1, x_1\ ) \wedge \text{reachable}(\ s_2, x_2\ )\ \} \\ \qquad \rightarrow P(x_1, x_2) \rightarrow P(\ s_1.\text{succ}(x_1),\ s_2.\text{succ}(x_2)\ ) \end{cases}$$

The homomorphic relation is total and functional if:

$$\begin{aligned}
&\text{total}(\, P{:}\,\mathbf{T}(\mathbf{O}),\, Q{:}\,\mathbf{T}(\mathbf{O}),\, R{:}\,\mathbf{T}(\mathbf{O},\mathbf{O})\,) :=\\
&\quad \forall x{:}\,\mathbf{O}\ \{\, P(x)\,\} \to \exists y{:}\,\mathbf{O}\ \{Q(y)\,\} \wedge R(x,y)\\
&\text{functional}(\, P{:}\,\mathbf{T}(\mathbf{O}),\, Q{:}\,\mathbf{T}(\mathbf{O}),\, R{:}\,\mathbf{T}(\mathbf{O},\mathbf{O})\,) :=\\
&\quad \forall x{:}\,\mathbf{O}\ \{\, P(x)\,\} \to \forall y_1, y_2{:}\,\mathbf{O}\ \{\, Q(y_1) \wedge Q(y_2)\,\} \to R(x,y_1) \wedge R(x,y_2) \to y_1 = y_2
\end{aligned}$$

Intersection of all homomorphic relations:

$$\begin{aligned}
&\mathbf{minhomrel}(\, s_1{:}\,\text{Seq}, s_2{:}\,\text{Seq}\,)(\, x_1{:}\,\mathbf{O},\, x_2{:}\,\mathbf{O}\,) :=\\
&\qquad \forall P{:}\,\mathbf{T}(\mathbf{O},\mathbf{O})\ (\{\,\mathbf{stricton}(\mathbf{prod}(\, s_1.\text{reachable},\, s_2.\text{reachable}\,),\, P)\,\}\\
&\qquad\quad \to \text{homrel}(s_1, s_2, P))\\
&\qquad\qquad \to P(x_1, x_2).
\end{aligned}$$

The intersection is by itself homomorphic (and well-typed):

$$\forall s_1, s_2{:}\,\text{Seq}\ \mathbf{stricton}(\mathbf{prod}(s_1.\text{reachable}, s_2.\text{reachable}), \text{minhomrel}(s_1, s_2)).$$

A Seq is freely generated if there exists a functional homomorphism into every Seq:

$$\begin{aligned}
&\text{freegen}(s_1{:}\,\text{Seq}\,) :=\ \forall s_2{:}\,\text{Seq}\ \exists P{:}\,\mathbf{T}(\mathbf{O},\mathbf{O})\\
&\quad \{\,\mathbf{strict}(\text{prod}(s_1.\text{reachable}, s_2.\text{reachable}), P)\,\} \wedge\\
&\qquad \text{homrel}(s_1, s_2, P) \wedge \text{functional}(s_1.\text{reachable}, s_2.\text{reachable}, P)
\end{aligned}$$

A Seq is freely generated iff it never returns to a point that it visited before. This can be expressed by the Peano axioms:

$$\begin{aligned}
&\text{peano}(\, s{:}\,\text{Seq}\,) :=\\
&\quad \begin{cases}
\forall x{:}\,\mathbf{O}\ s.\text{reachable}(x) \to s.\text{succ}(x) \neq s.0\\
\forall x, y{:}\,\mathbf{O}\ s.\text{reachable}(x) \wedge s.\text{reachable}(y)\\
\qquad\qquad \to s.\text{succ}(x) = s.\text{succ}(y) \to x = y
\end{cases}
\end{aligned}$$

We wrote different proofs in order to find a user-friendly calculus for PHOLI. By trying different proofs, we found that resolution can be viewed as a basis for a user-friendly proof calculus. Our proofs also confirmed the observations by our supervisor that sequent calculus is not suitable for the poof verification.

First proof is the totality of the homomorphism relation. Recall the totality definition:

$$\begin{aligned}
&\text{total}(\, P{:}\,\mathbf{T}(\mathbf{O}),\, Q{:}\,\mathbf{T}(\mathbf{O}),\, R{:}\,\mathbf{T}(\mathbf{O},\mathbf{O})\,) :=\\
&\quad \forall x{:}\,\mathbf{O}\ \{\, P(x)\,\} \to \exists y{:}\,\mathbf{O}\ \{Q(y)\,\} \wedge R(x,y)
\end{aligned}$$

Proof:

1 | | $s_1.reachable(x)$

2 | | $\forall P\ P(s_1.0) \wedge (\forall x\ P(x) \rightarrow P(s_1.succ(x))) \rightarrow P(x)$

3 | | $Q := \lambda x\ \exists y\ s_2.reachable(y) \wedge R(x, y)$

4 | | $Q(s_1.0) \wedge (\forall x\ Q(x) \rightarrow Q(s_1.succ(x))) \rightarrow Q(x)$

5 | | $s_2.reachable(s_2.0) \wedge R(s_1.0, s_2.0)$

6 | | $\exists y\ s_2.reachable(y) \wedge R(s_1.0, y)$

7 | | $Q(s_1.0)$

8 | | $(\forall x\ Q(x) \rightarrow Q(s_1.succ(x))) \rightarrow Q(x)$

9 | | $Q(x)$

10 | | $\exists y\ \ s_2.reachable(y) \wedge R(x, y)$

11 | | | $s_2.reachable(y) \wedge R(x, y)$

12 | | | $s_2.reachable(s_2.succ(y))$

13 | | | $R(s_1.succ(x), s_2.succ(y))$

14 | | | $s_2.reachable(s_2.succ(y)) \wedge R(s_1.succ(x), s_2.succ(y))$

15 | | | $\exists y\ s_2.reachable(y) \wedge R(s_1.succ(x), y)$

16 | | | $Q(s_1.succ(x))$

17 | | $Q(s_1.succ(x))$

18 | $\forall x\ Q(x) \rightarrow Q(s_1.succ(x))$

19 | $Q(x)$

20 | $\exists y\ s_2.reachable(y) \wedge R(x_1, y)$

21 $\forall x\ s_1.reachable(x) \rightarrow \exists y\ s_2.reachable(y) \wedge R(x, y)$

22 $\text{total}(s_1.reachable, s_2.reachable, R)$

In order to prove $\text{homrel}(s_1, s_2, \text{minhomrel}(s_1, s_2))$, we need to prove that $\text{minhomrel}(s_1, s_2)$ holds between $s_1.0$ and $s_2.0$, and that $\text{minhomrel}(s_1, s_2)$ is preserved by taking $s_1.succ$ to the left, and $s_2.succ$ to the right. We start with the first condition:

1 | | stricton(prod($s_1.reachable, s_2.reachable$), P)

2 | | homrel(s_1, s_2, P)

3 | | $P(s_1.0, s_2.0)$

4 | $\forall P$ stricton(prod($s_1.reachable, s_2.reachable$), P) $\rightarrow$ homrel(s_1, s_2, P) $\rightarrow P(s_1.0, s_2.0)$

5 | minhomrel($s_1.0, s_2.0$)

Next we show the second condition: Assume that minhomrel(s_1, s_2, x_1, x_2) is true. We need to show that minhomrel($s_1, s_2, s_1.succ(x_1), s_2.succ(x_2)$) is also true.

1 | $minhomrel(s_1, s_2, x_1, x_2)$

2 | $\forall P$ stricton(prod($s_1.reachable, s_2.reachable$), P) $\rightarrow$ homrel(s_1, s_2, P)

3 | $\rightarrow P(x_1, x_2)$)

4 | | stricton(prod($s_1.reachable, s_2.reachable$), P)

5 | | homrel(s_1, s_2, P))

6 | | $P(x_1, x_2)$

7 | | $P(s_1.succ(x_1), s_2.succ(x_2)$)

8 | $\forall P$ stricton(prod($s_1.reachable, s_2.reachable$), P) $\rightarrow$ homrel(s_1, s_2, P)

9 | $\rightarrow P(s_1.succ(x_1), s_2.succ(x_2)$))

10 | minhomrel($s_1, s_2, s_1.succ(x_1), s_2.succ(x_2)$)

Another proof that we have done, was to prove that when a sequence fulfills the Peano axioms, then the intersection of all homomorphic relations from this sequence into any other sequence, is always functional.

$\forall s_1 : Seq\ peano(s_1)$

$\forall s_2 : Seq$

$\forall x, y_1, y_2\ s1.reachable(x) \wedge s2.reachable(y_1) \wedge s2.reachable(y_2) \rightarrow$

$minhomrel(s_1, s_2, x, y_1) \wedge minhomrel(s_1, s_2, x, y_2)$

$x = s_1.0$

$s2.reachable(y_1) \wedge minhomrel(s_1, s_2, s_1.0, y_1) \wedge$

$s2.reachable(y_2) \wedge minhomrel(s_1, s_2, s_1.0, y_2)$

$((x = s_1.0 \wedge y_1 = s_2.0) \vee$

$(\exists z_1, z_2\ s_1.reachable(z_1) \wedge s_2.reachable(z_2) \wedge minhomrel(s_1, s_2, z_1, z_2) \wedge$

$s_1.succ(z_1) = x \wedge s_2.succ(z_2) = y_1)) \wedge$

$((x = s_1.0 \wedge y_2 = s_2.0) \vee$

$(\exists z_1, z_2\ s_1.reachable(z_1) \wedge s_2.reachable(z_2) \wedge minhomrel(s_1, s_2, z_1, z_2) \wedge$

$s_1.succ(z_1) = x \wedge s_2.succ(z_2) = y_2))$

$(y_1 = s_2.0 \wedge y_2 = s_2.0)$

$y_1 = y_2$

$minhomrel(s_1, s_2, s_1.succ(x), y_1) \wedge minhomrel(s_1, s_2, s_1.succ(x), y_2)$

$s2.reachable(y_1) \wedge minhomrel(s_1, s_2, s_1.succ(x), y_1) \wedge$

$s2.reachable(y_2) \wedge minhomrel(s_1, s_2, s_1.succ(x), y_2)$

$((s_1.succ(x) = s_1.0 \wedge y_1 = s_2.0) \vee$

$(\exists z_1, z_2\ s_1.reachable(z_1) \wedge s_2.reachable(z_2) \wedge minhomrel(s_1, s_2, z_1, z_2) \wedge$

$s_1.succ(z_1) = s_1.succ(x) \wedge s_2.succ(z_2) = y_1)) \wedge$

$((s_1.succ(x) = s_1.0 \wedge y_2 = s_2.0) \vee$

$(\exists z_1, z_2\ s_1.reachable(z_1) \wedge s_2.reachable(z_2) \wedge minhomrel(s_1, s_2, z_1, z_2) \wedge$

$s_1.succ(z_1) = s_1.succ(x) \wedge s_2.succ(z_2) = y_2))$

$s_2.succ(z_2) = y_1 \wedge s_2.succ(z_2) = y_2$

$y_1 = y_2$

$\forall x, y_1, y_2\ minhomrel(s_1, s_2, x, y_1) \wedge minhomrel(s_1, s_2, x, y_2) \rightarrow y_1 = y_2$

Decisions Made

Divided team into 2 subgroups

We have split our team: one (Alina, Ilnur, Yuliya) worked on weight calculation algorithm and verification of logical formulae, while the other (Alikhan, Viktor) worked on pretty printing and parser. Each subgroup was working on its part, uploading the changes to GitHub repository, and was having regular weekly meetings, most of them with advisor as well.

Interpretations initially defined as static tables

As of our first approach, an interpretation was just a mapping that assigns a truth value to the identifier. The overall purpose of interpretation is to provide a concrete context in which a formula can be evaluated. The logic operators were also initially implemented as static tables, and given an interpretation of the identifiers and logic formula, we could evaluate the formula and output the corresponding truth values.

However, we encountered the problem that this approach does not work with quantifiers and objects can not be represented as arguments (their size is not fixed, in the case of truth values, it is 3), so we proceeded to make it stronger.

Challenges and Solutions

Parser

Over the last two semesters, we faced several challenges while implementing the parser. The first issue was ambiguity caused by quantifiers. Behavior of quantifiers does not fit the concept of operator priorities. We solved this by redesigning the grammar so that quantifiers' priority did not interfere with other operators. The same issue was with lazy operators. This was resolved by increasing their priorities.

And in general, we had problems every time a new element was added to the grammar. So we had to revisit some parts of the grammar several times, changing attributes, action code and used data structures. So the design was done in such iterations. Also, we had some logical mistakes with associativity in simple operators, which were found by our adviser.

Lastly, we faced the issue of adding the de Bruijn indices to the grammar. In particular, the problem was that the store the de Bruijn indices we need a special data structure - similar to stack, but also that elements could be quickly accesses by the key. However, we realized that it is inconvenient to support de Bruijn indices during the parser stage, so we decided that it will be better to implement it in further steps.

Evaluation

The project turned out to be more complex than anticipated by both the team and the advisor, which is why we could not complete all initially planned tasks. However, we made a significant progress and core components—such as the parser, pretty printer and logic formula interpretation—were successfully implemented. Both the team and the advisor are satisfied with the results, and the project shows potential for future development.

Conclusion

We began with a literature review to understand the theoretical foundation and explored existing tools provided by our supervisor, including a tree generator, a parser generator, and a structural type checker. Building on this, we defined a concrete syntax for logical formulas and implemented a pretty printer to display them in a readable format. We then constructed a parser that integrates our syntax with the existing parser generator. Using these tools, we tested various logical examples to verify correctness and compatibility. Additionally, we developed a structural weight calculation algorithm for logical terms and implemented functionality to check whether two formulas are logically equivalent, or whether the correctness of one formula implies the equivalence with another. Unfortunately, we were not able to complete all the tasks we initially planned due to time constraints. In particular, the user interface, type checker, and full proof checker remain as future work. These components are crucial for improving usability, ensuring well-formedness of formulas, and supporting complete automated or semi-automated proof verification. We see these as natural next steps to turn our project into a more complete and accessible tool for students and researchers alike.

References

[1] Thierry Coquand, Gérard Huet, Christine Paulin-Mohring, Bruno Barras, Jean-Christophe Filliâtre, Hugo Herbelin, Chetan Murthy, Yves Bertot, and Pierre Castéran. Coq. Can be obtained from the Coq Homepage.

[2] Hans de Nivelle. Theorem proving for logic with partial functions by reduction to Kleene logic. In Christoph Benzmüller and Jens Otten, editors, *Automated Reasoning in Quantified Non-Classical Logics (ARQNL) 2014*, pages 71–85. VSL Workshop Proceedings, 2014.

[3] Hans de Nivelle. Maphoon: A c++ based parser generator, 2022. Can be obtained from here.

[4] Hans de Nivelle. TreeGen: A code generator for recursively defined data types, 2023. Can be obtained from here.

[5] Hans de Nivelle. An interactive proof calculus for strict, three-valued, higher-order logic. In *2012 ACM Subject Classification*, 2025.

[6] Git. Git - fast version control system, 2025. Accessed: 2025-04-25.

[7] Inc. GitHub. Github: Where the world builds software, 2025. Accessed: 2025-04-25.

[8] Larry Paulson, Tobias Nipkow, and Makarius Wenzel. Isabelle. Can be obtained from the Isabelle Homepage.

[9] Piazza. Piazza: The online collaboration platform, 2025. Accessed: 2025-04-25.

[10] Vaughan R. Pratt. Top down operator precedence. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, page 41–51, New York, NY, USA, 1973. Association for Computing Machinery.