

The Sequent Calculus Calculator

Bachelor thesis

DEPARTMENT OF COMPUTER SCIENCE
HOCHSCHULE FÜR TECHNIK RAPPERSWIL

Spring term 2019

Authors:	Matteo Kamm, Mike Marti
Advisor:	Prof. Dr. Farhad Mehta
Client:	Institute for Software
External Co-Examiner:	Dr. Hermann Lehner
Internal Co-Examiner:	Prof. Stefan Richter

Contents

1	Setting	5
2	Goal	6
3	Technology decisions	7
3.1	Haskell to JavaScript transpilers	7
3.2	PureScript	7
3.3	Elm	7
3.4	Decision	7
3.5	Renderer	8
3.6	UI Frameworks	8
4	Design decisions	9
4.1	Kernel	9
4.2	Switching between calculi at runtime	9
4.3	Monotonic logics	9
4.4	Nominals and De Bruijn indices	10
4.5	Architecture	10
4.5.1	Frontend	11
4.5.2	Backend	11
5	Make or reuse decision	12
6	Data types	13
7	Parser	15
7.1	Example	15
8	Instantiation algorithm	16
9	Application algorithm	17
9.1	Implicit hypothesis matching	18
10	Substitution operator	19
10.1	Known limitations	19
10.2	Applying a substitution	19
10.3	Alpha conversion	19
11	Side Condition	21
11.1	Instantiation of side conditions	21
11.2	Checking of side conditions	21
11.3	Definition of side conditions	21
11.4	Limitation of the genericity	22
11.5	Supported side condition types	23
12	Proof tree export and import	24
12.1	Implementation	24
12.2	Format of the JSON	24
12.3	Emerging problems	26
12.3.1	Import checks	26

12.3.2 LCF-style circumvention	26
13 User Interface	27
13.1 Special character encoding	27
13.2 Rendering algorithms	27
13.2.1 Proof tree renderer	28
13.2.2 Proof rule schema renderer	28
13.2.3 Term renderer	28
13.3 Webpage header	28
13.4 Proof rule schema view	29
13.5 Instantiation view	30
13.6 Proof view	31
14 Testing	32
14.1 Test dependencies	32
14.2 Fuzzing	32
14.2.1 Shrinker	33
14.3 System tests	33
15 Implementing a new calculus	35
15.1 Name and identifier	35
15.2 Parser	35
15.3 Proof rule schemas	36
15.4 Term renderers	36
15.5 Special character encoding	37
16 Deployment	38
17 Evaluation	39
17.1 Implemented features	39
17.2 Features that were not implemented	40
17.3 Problems	40
17.3.1 Renderer	40
17.3.2 GitLab free plan	41
17.3.3 Styling-problem with Safari	41
17.4 Consequences of the LCF-style	42
17.5 Conclusion	42
Appendix	43
A Task description	43
B Project plan	46
B.1 Milestones	46
C Risk analysis	47
C.1 Occurences	48
D EBNF descriptions	49
D.1 Basic propositional calculus	49
D.2 Propositional calculus	49
D.3 Basic first-order predicate calculus with equality	49

D.4	First-order predicate calculus with equality	50
D.5	Simply typed lambda calculus	50
D.6	Polymorphically typed lambda calculus	50
E	Usability test with paper mockups	51
E.1	Insights	52
E.2	Conclusion	53
E.3	Paper mockups	54
F	Development Roadmap	58
G	Redevelopment Roadmap	60
H	List of figures	62
I	List of tables	62
J	List of listings	62
K	Acronyms	63
L	Bibliography	64

Abstract

The sequent calculus is the most widely used style of formal proof in computer science. Its applications include proving logical statements correct and checking the type correctness of programs. The sequent calculus is taught as part of the computer science bachelor curriculum at the HSR.

Learning the sequent calculus on paper involves copying a lot of large formulae and is therefore tedious. Mechanical checks cannot be performed when proving on paper, which makes the process error prone. A few online web-based interactive proof assistants already exist. Unfortunately, none of them are currently suitable for use as a didactic aid at the HSR. Given the popularity of the sequent calculus, it would be advantageous to have a well engineered, interactive, web-based proof assistant as a learning aid for the sequent calculus.

This project has resulted in a web-based proof assistant, written in the functional programming language Elm, that can be used as a didactic aid to teach several different calculi in the sequent calculus style. Proofs are performed using a drag and drop style. The implemented solution is structured in a way, to allow for easy extensibility without the need of changing the kernel. As a result, new calculi with different proof rule schemas, side conditions and operation symbols can be added without much effort. The implemented website is a single-page application, running without a complicated deployment.

The thesis shows the possibilities and also the corresponding limits of sequent calculus style proofs that can be performed in web applications.

1 Setting

In computer science the most frequently used representation to reason about deduction systems is the sequent calculus. The use of paper-based proofs to learn the application of the sequent calculus is often tedious. A lot of pattern matching, rewriting and copying is involved in the process of proving a sequent on paper. Furthermore, it is an error-prone process because each deduction step cannot be checked by a formal machine.

For those reasons, a number of web-based interactive provers exist. Upon closer inspection it was found that none were ideal to be used as didactic aids at the HSR due to the following reasons:

- Logitext¹ allows the user to prove sequents by clicking directly on them. This means that they do not have to know about the proof rule schemas that get applied. This is different from the process that is involved when proving on paper. Moreover, the user cannot go back in a proof and try to apply different rules. Implementing new theories in this setting seems difficult and impossible.
- The incredible proof machine² supports multiple calculi and also seems to be fairly extensible. However, the proof rule schemas are in the natural deduction style and proofs are performed in a DAG.
- The propositional sequent calculus prover³ works only with propositional logic. Unfortunately, the proofs are displayed in an indented unordered list and not in a tree structure.
- SeqProver⁴ supports multiple output formats for proofs, such as plain text and L^AT_EX. It only works for the predicate calculus and is not an interactive prover.

As the sequent calculus is a powerful and important tool, it is clear that a well designed sequent calculus calculator is highly sought-after.

¹<http://logitext.mit.edu/tutorial>

²<http://incredible.pm/>

³<https://www.nayuki.io/page/propositional-sequent-calculus-prover>

⁴<http://bach.istc.kobe-u.ac.jp/seqprover/>

2 Goal

The main goal of this thesis is to develop an interactive web-based theorem prover to teach the sequent calculus. The software written in this project should outperform existing solutions in the areas of usability, attractiveness, maintainability and extensibility. The idea is that it can be used during the first half of the PrfM course at the HSR.

The formal requirements of the application, as provided by the advisor, are the following:

The application must:

- Support the construction of interactive sequent calculus style proofs in propositional logic, predicate logic, the simply typed lambda calculus and the polymorphically typed lambda calculus.
- Be easy and intuitive to use for teaching and learning.
- Have an attractive user interface that supports a drag and drop style of proof along with explicit instantiation of meta-variables that cannot be determined using unification.
- Require no prior installation as far as possible.
- Be maintainable and easily extensible programmatically to support reasoning about other theories.
- Use a CI/CD pipeline for development and deployment.
- Use a Haskell-based toolchain as far as possible.
- Use a LCF-style proof kernel.
- Take advantage of the good points in existing solutions.
- Be able to be effectively used for teaching and learning at the HSR and internationally.

The advisor describes additional requirements that can be included:

The application should:

- Be able to indicate which rule schemas are applicable to the current goal.
- Be able to save and load proofs.
- Be able to export proofs in a variety of formats.
- Be able to define (programmatically) and use (via the UI) proof tactics.

In general, the kernel has to be of high enough code quality to be used as a case study for functional programming. The results are to be documented in the form of a paper suitable for academic publication.

3 Technology decisions

The software created for this thesis had to be written in a statically typed functional programming language. This means that the functional programming language had to output JavaScript that would then run in the browser of the user. This left multiple choices to be made during the evaluation phase of the project. This section describes the decisions taken and possible alternatives as well as reasons why they were not chosen.

3.1 Haskell to JavaScript transpilers

There exist multiple transpilers that convert Haskell to JavaScript. Each of them has different properties. The ones that were considered during the evaluation phase are listed below.

GHCJS

GHCJS is still under development, but compiles most Haskell based libraries to JavaScript [11]. To interact with JavaScript a so called Foreign Function Interface (FFI) can be used. The installation process of GHCJS is quite tiresome as it can take up to multiple hours depending on the computer hardware and internet connection.

Fay

Fay compiles a proper syntactic and semantic subset of Haskell to JavaScript [10]. It also uses FFIs to call JavaScript code. The opposite, to call Fay from JavaScript, is also possible.

3.2 PureScript

PureScript focuses on the generation of readable, efficient JavaScript [17]. There is an extensive collection of web development libraries available for PureScript. The syntax is similar to the one of Haskell, but the evaluation strategy is quite different. PureScript is strictly evaluated as this matches the evaluation strategy of JavaScript.

3.3 Elm

Elm's syntax is similar to the one of Haskell [4]. The language is very beginner-friendly. There is no support for type classes in Elm [8]. This implies that some parts of the source code have to be duplicated in order to work with different types and refactoring types can become cumbersome. For instance, changing from a *List* to a *Dict* requires one to change all *List.map* calls to *Dict.map* calls. Elm has an enormous library and tooling ecosystem. With so called ports and flags it is possible to interoperate with JavaScript code.

3.4 Decision

Elm was chosen for this project because it seemed to be the most simple language that would solve the problem at hand. The tooling and library support played an important role in this decision. With this choice it was also clear, that some downsides had to be accepted. Elm is still lacking debugging and refactoring features and also requires some initial training.

The number-two choice was GHCJS. However, the tedious installation led us to favour Elm instead. Fay was not chosen because of the limited amount of libraries available at the time of evaluation⁵. The last library that we decided against is PureScript. It still uses **Bower**⁶ as a package manager, which has not been in active development since 2017. Moreover, PureScript seems to be difficult to learn and get started with.

3.5 Renderer

To render proof rule schemas and proof rules we will write our own renderer. We choose not to use an existing rendering library such as **KaTeX**⁷ or **MathJax**⁸. The following (not exhaustive) list presents some of the reasons for that decision:

- Adding event listeners onto rendered \LaTeX is not possible.
- Styling parts of the rendered \LaTeX is not possible.
- The mentioned libraries use JavaScript which we try to avoid as much as possible.
- The instantiation view uses input fields to represent meta variables. Adding those to the rendered output is not trivial.
- Proof trees use a custom \LaTeX style file⁹ which is probably not supported by those libraries.

Yet, implementing our own renderer has several disadvantages. It takes a lot of time and can be error prone. Furthermore, we have to support multiple browsers which can become time consuming. As extensibility and customizability are of utmost importance to this project, we still decided to write a custom solution.

One renderer should support the conversion of proof rule schemas, proof rule instantiations and proof trees to HTML. This implies that the renderer has to be generic and that it has to be configurable. For example, to support the rendering of proof rule instantiations it has to replace meta variables with input fields. The implementation of this can be seen in section 13.2.2.

3.6 UI Frameworks

For the styling of the web application the well-known CSS framework Bootstrap¹⁰ is being used. This framework was chosen because of its use in the Lambda Calculus Calculator project [3]. The decision was made that both, The Lambda Calculus Calculator and The Sequent Calculus Calculator, should have a similar look and feel to increase the familiarity and usability for users who work with both tools.

Special icons are provided by the icon provider Fontawesome¹¹ because of its reputation and wide range of high quality icons.

⁵Fay package list: <https://github.com/faylang/fay/wiki#fay-packages>

⁶<https://bower.io/>

⁷<https://katex.org/>

⁸<https://docs.mathjax.org/>

⁹<http://research.nii.ac.jp/~tatsuta/proof-sty.html>

¹⁰<https://getbootstrap.com/>

¹¹<https://fontawesome.com/>

4 Design decisions

The design of The Sequent Calculus Calculator is shaped by several decisions. These are described and explained in this section.

4.1 Kernel

As the calculator has to be extensible, i.e. adding new calculi has to work seamlessly, there has to be a well-defined set of APIs that are independent of any calculus. This part is called the kernel. Changing the kernel could potentially break other parts of the project. Therefore a stable kernel that still supports extension is required.

The design of this kernel is influenced by the paper “Strategic Principles in the Design of Isabelle” by Lawrence Paulson [16]. He described that the interactive theorem prover Isabelle [18] achieved its flexibility by three basic features:

- higher-order syntax
- logical variables and unification
- search primitives based on lazy lists

For The Sequent Calculus Calculator the higher order abstract syntax is the lambda calculus syntax. This decision was already part of the problem statement. Unification is currently not supported by the calculator and users have to instantiate proof rule schemas on their own. If this feature should be implemented in the future, a modified version of the unification algorithm presented in the Programming Languages and Formal Methods course could be used. The last feature described in the paper is not part of The Sequent Calculus Calculator as Elm does not support lazy lists by default [4].

Another major design decision is the style of the kernel. The Sequent Calculus Calculator uses an LCF-style kernel. Section 6 documents how this is implemented.

Note that there is exactly one kernel for all calculi and not one per calculus. The kernel is a general-purpose prover. The use of a HOAS makes this design possible. It would also be possible to use one syntax per calculus and then map them to some kind of HOAS when they are passed to the kernel. However, this would add additional logic to the kernel which is not really necessary if you use one HOAS.

4.2 Switching between calculi at runtime

One important point is that The Sequent Calculus Calculator should be able to switch between calculi at runtime. The users should not have to load a new webpage when they want to switch to a different calculus. This has important implications on the structure of the frontend as it also has to be divided into a generic and a calculus-specific part.

4.3 Monotonic logics

The Sequent Calculus Calculator only supports monotonic logics, i.e. logics that have an inference rule called weakening [19]. The use of the implicit hypothesis H , which is inherently part of The Sequent Calculus Calculator, dictates this requirement on the logic.

4.4 Nominals and De Bruijn indices

To represent abstractions in the HOAS, nominals were chosen instead of De Bruijn indices as they are more intuitive to understand. Doing α -conversions and checking α -equivalences becomes more complicated by using nominals. This disadvantage was accepted in favour of simplicity during the design of the kernel.

4.5 Architecture

As shown in figure 1, the application consists of two parts, a frontend and a backend. The backend is again split into two parts, the kernel and multiple modules. Despite the separation, the entire application runs on the clients machine. That is, the backend and frontend are both compiled to JavaScript. As there is no client-server communication apart from the initial request, no long-lived internet connection is required.

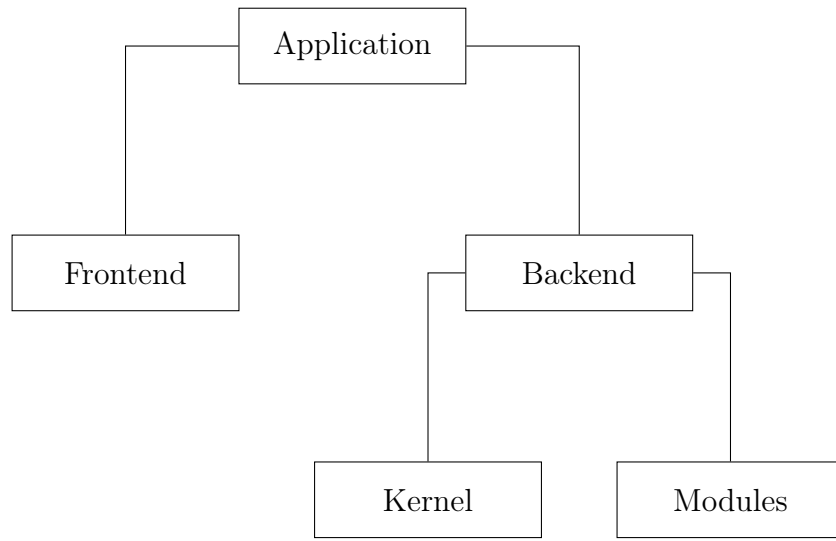


Figure 1: Component overview

How the components interact and what information they share with one another can be seen in figure 2.

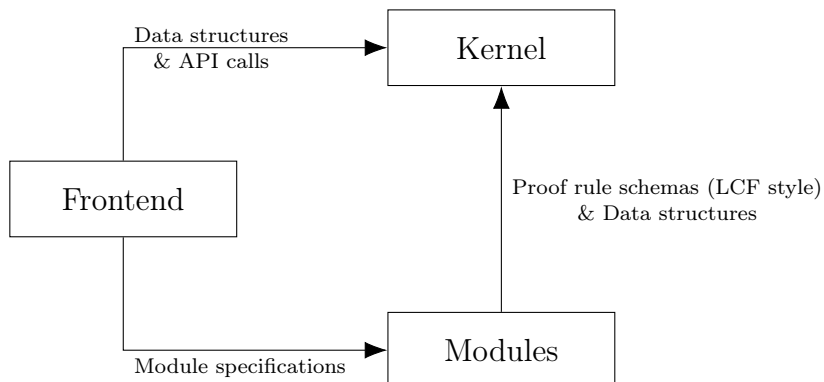


Figure 2: Interaction overview

The kernel does not depend on other components. The modules pass their proof rule schemas to the kernel. They also use the data structures provided by the kernel. The frontend uses the module specifications that are part of the modules. These are described as part of section 15. Finally, the frontend calls the kernel to apply proof rules or instantiate proof rule schemas.

4.5.1 Frontend

The user interface is a web application which is written in Elm. The decision on why this language was chosen is evident from section 3.

Elm modules

According to the Elm architecture guide, a typical Elm project should consist of a model, a view representation and an update function [6]. This architecture resembles the well known Model-View-Controller pattern. The model contains the application state. In our case, it also holds references to calculi specific functions, such as the parser and the supported operators. The view consists of the HTML representation of the application. It is responsible for combining the different DOM elements. The update function reacts to user input and manipulates the DOM accordingly.

This core idea is used as a design guideline for The Sequent Calculus Calculator. However, the different view representations and the kernel logic were again separated into different modules to improve the reusability of generic logic. This allows a separation of concerns and leads to less duplicated code. Simultaneously it increases the complexity of the application.

4.5.2 Backend

As the kernel has to be highly extensible, it has to support some kind of plugin architecture. For The Sequent Calculus Calculator plugins are user defined modules. Those modules allow for extension by new, not yet supported calculi. The backend could theoretically be used on the command line as well. It is not dependent on a particular user interface.

Kernel

The kernel contains generic algorithms that support the instantiation of proof rule schemas and the application of proof rules. The kernel does not depend on calculi modules and can therefore be used without them. Furthermore it is able to load and save previously created proofs stored as JSON files.

Modules

A module consists of components that are specific to a calculus. Each calculus that has to be supported by the web application needs its own Elm modules. A module contains a parser, implemented based on the EBNF of the calculus grammar. Implementing a parser is a complex and time consuming task. That is why a parsing library was used [5]. Section 5 answers why this specific library was chosen. The proof rule schemas of a calculus are also part of the module. They are instantiated during a proof performed by the user.

It is important to note here, that only one such module can be active at once. This is to prevent proof rules of different calculi to interfere with each other.

5 Make or reuse decision

This decision answers the question whether a component should be written from scratch or reused as is from other sources. In this case this question had to be answered for the parser component. We decided to use two parsing libraries written in Elm [5, 13]. This allowed us to do rapid prototyping early on without having to write a complex parser. The chosen parser falls into the category of top-down parsers [2]. One downside of this library is that left-recursive grammars cannot be parsed. In the case of propositional calculus predicates, this problem required some additional effort to solve. For an example on how this problem was solved for the basic predicate calculus syntax, see section 7.1. With this library we lost some degree of flexibility which a hand written parser would have provided.

6 Data types

The most important data types of this project are *Term*, *Sequent*, *TrustedProofRuleSchema*, *TrustedProofRule* and *ProofTree*. They are all part of the kernel and play an important role in most algorithms.

Term

As described in section 4 The Sequent Calculus Calculator uses the lambda calculus syntax.

```
type Term
  = Variable String
  | Metavariable String
  | Application Term Term
  | InParentheses Term
```

Listing 1: Term data type

A meta variable represents a variable that gets replaced by a concrete term during the instantiation process. Meta variables are part of proof rule schemas but are never present in concrete proof rules.

The *InParentheses* constructor should preserve parentheses declared by the user. Without this constructor it would not be possible to retain this information in the AST. In order to not confuse the user, explicit parentheses are never removed and always displayed.

Sequent

A sequent contains hypotheses and goals, both of which can contain multiple elements. For our purpose, the goal side of a sequent contains exactly one goal. However, there are calculi for which the goal side contains multiple *Terms*.

The resulting data type looks like this:

```
type alias Sequent =
  { hypotheses : List Term, goals : List Term }
```

Listing 2: Sequent data type

TrustedProofRuleSchema

Trusted proof rule schemas are part of the kernel and, surprisingly, not of the calculus module. That is because the *TrustedProofRuleSchema* type is implemented in the LCF-style, i.e. the constructor is not exposed. The creation of trusted proof rule schemas is therefore only possible inside this kernel module. Users cannot create arbitrary proof rule schemas on their own. This prevents users and developers from creating arbitrary proofs that might be wrong.

TrustedProofRule

Similar to the *TrustedProofRuleSchema* type, the *TrustedProofRule* type does not expose its constructor. A trusted proof rule can only be created by instantiating a trusted proof rule schema. Users are never able to create proof rules on their own.

ProofTree

A proof tree consists of *Nodes*. A *Node* always contains a path that is represented as a *List* of *Ints* and a *Sequent*. Furthermore, when no *TrustedProofRule* has been applied to a *Node* it contains a *Nothing*. If, on the other hand, a *TrustedProofRule* has been applied it contains a *Tuple* consisting of the proof rule name and a list of child *Nodes*.

```
type ProofTree
  = Node (List Int) Sequent (Maybe (String, List ProofTree ))
```

Listing 3: Proof tree data type

The so called pending subgoals of a proof tree are the leafes that have not been proven. Those are the *Nodes* that contain a *Nothing*. Conversely, a proven leaf contains a *Just* of an empty list. A proof tree is proven if it contains no pending subgoals.

Applying a *TrustedProofRule* to a *Node* of a *ProofTree* converts the tree into a new tree. The concrete algorithm is described in section 9. *ProofTrees* cannot be directly created or manipulated by the user as it is also hiding its constructor.

The previously mentioned path that is part of each *Node* is always unique within the *ProofTree*. This is necessary because the (==) operator of Elm uses structural equivalence. If the path was not present, a proof rule could be applied to multiple (accidentally equivalent) locations. A path is constructed by labeling each child *Node* with an increasing number starting at zero. The path of a particular *Node* could look like this: [0,1,0], i.e. the first child of the second child of the first child.

7 Parser

The parsers are part of the calculus specific modules. For new calculi the parser has to be programmatically provided. It is not possible to generate the parser logic based on the EBNF, as this would require an Elm-based parser generator.

As discussed in section 4, The Sequent Calculus Calculator uses the lambda calculus syntax. This means that every term has to be represented as some kind of function application. The parser has to convert text representations to function applications.

7.1 Example

The following example illustrates how a term is parsed. First, the user provides a string as input. The parser then tries to match it against one of the possible categories. For *basicPC* those are the logical complement, conjunction, variable or contradiction. The most difficult part lies in this step. The *basicPC* grammar is left recursive:

$$P \rightarrow P \wedge P \mid \neg P \mid A \dots Z \mid \perp$$

It contains a production rule of the form $A \rightarrow A\alpha$. A top-down parser is not able to parse such production rule because it would end up in an infinite loop. To solve this problem the grammar has to be slightly modified:

$$\begin{aligned} P &\rightarrow \neg PP' \mid A \dots ZP' \mid \perp P' \\ P' &\rightarrow \wedge P \mid \varepsilon \end{aligned}$$

This solution is able to parse the same strings but removes the left-recursion [1]. Note that a term of the form $A \wedge B \wedge C$ is ambiguous and will not be parsed correctly. To resolve such ambiguities, the user has to use parentheses.

In an ideal scenario the module developer does not have to think about this property of the grammar. Therefore, they can use the Elm parsing library `parser-extras` which is able to parse expressions, operators and resolve their precedence [13]. The usage of this library is highly recommended but not mandatory.

Parsing the term $\neg(A \wedge B)$ with the above grammar will result in the following data structure:

```
Application (Variable "¬") (
  Application
    (Application (Variable "^") (Variable "A"))
    (Variable "B")
)
```

Listing 4: Parsed term represented in the term data structure

8 Instantiation algorithm

This algorithm takes instantiations provided by the user and creates a *TrustedProofRule* from a *TrustedProofRuleSchema*. This is achieved by replacing meta variables with concrete terms. The implementation is straightforward:

```
instantiate : String -> Term -> Term -> Term
instantiate metavariable assignment term =
  case term of
    Variable _ ->
      term

    Metavariable content ->
      if metavariable == content then
        assignment

      else
        term

    Application left right ->
      Application
        (instantiate metavariable assignment left)
        (instantiate metavariable assignment right)

    Abstraction parameter body ->
      Abstraction
        (instantiate metavariable assignment parameter)
        (instantiate metavariable assignment body)

    InParentheses inParanthesis ->
      InParentheses
        (instantiate metavariable assignment inParanthesis)
```

Listing 5: Instantiation of a term

The if statement checks if the correct meta variable has been reached. If so, it does a simple replace. Otherwise the structure of the term is unchanged. Despite the fact that a term of a *TrustedProofRuleSchema* never contains parentheses this case has to be handled.

The implicit meta variable H is never explicitly instantiated by the user. This meta variable gets its assignment when applying the proof rule to a pending subgoal of the proof tree. With this restriction in mind it's clear that each meta variable gets assigned to exactly one term. How this implicit meta variable is instantiated is described in section 9.1.

9 Application algorithm

When applying a *TrustedProofRule* to a *Node* of the *ProofTree* multiple preconditions have to hold. As seen in listing 6, there are three preconditions:

```
isApplicable : ProofTree -> ProofTree -> TrustedProofRule -> Result String ()
isApplicable tree location trustedProofRule =
  boolToResult (isEmpty location) "Only pending subgoals"
  |> Result.andThen
    (always <|
      boolToResult (isSubtree location tree) "Not part of tree"
    )
  |> Result.andThen
    (always <|
      isApplicableToLocation trustedProofRule location
    )
  |> Result.map (always ())
```

Listing 6: Preconditions of an application

The *isEmpty* function call checks if the location where the proof rule should be applied is a leaf of the *ProofTree*. If not, then no proof rule can be applied.

The second function call checks if the location is part of the tree. It could be that a developer accidentally applied a proof rule to a *Node* that is not even part of the current *ProofTree*.

The last precondition is a bit more interesting. It checks if the consequent of the *TrustedProofRule* is equal to the pending subgoal, i.e. the *Sequent* contained within the location *Node*. It is important to note that this check disregards the order of terms. P, Q can be applied to Q, P . For the hypotheses of the sequents a sublist comparison is performed because the implicit hypothesis H could match multiple terms. Finally, the side conditions are evaluated.

If all preconditions are complied, the application takes place. The following listing 7 shows part of the application algorithm.

```

applyProofRule : ProofTree -> ProofTree -> List Int -> ProofRule -> ProofTree
applyProofRule (Node path sequent possibleChildren) appliedLocation
    currentPath proofRule =

    if Node path sequent possibleChildren == appliedLocation then
        proofRule.antecedents
        |> List.indexedMap Tuple.pair
        |> List.map
            (\( index, element ) ->
                createNode
                    (currentPath ++ [ index ])
                    element
            )
        |> Tuple.pair proofRule.name
        |> Just
        |> Node path sequent

    else
        case possibleChildren of
            Nothing ->
                Node path sequent Nothing

            Just ( name, children ) ->
                children
                |> List.indexedMap Tuple.pair
                |> List.map
                    (\( index, element ) ->
                        applyProofRule
                            element
                            appliedLocation
                            (currentPath ++ [ index ])
                            proofRule
                    )
                |> Tuple.pair name
                |> Just
                |> Node path sequent

```

Listing 7: Application of a proof rule

The if statement checks if the recursion reached the location where the proof rule should be applied. If so the new pending subgoals get created and all of them receive their own paths. If, on the other hand, the base case has not been reached, the function *applyProofRule* is called on all children.

9.1 Implicit hypothesis matching

The matching of the implicit hypothesis H is quite simple. The algorithm uses the proof rule that should be applied as well as the node of the proof tree and modifies the proof rule appropriately. It does this by adding all the hypotheses that are not present in the proof rule but in the node of the proof tree to the proof rule itself.

If, for example, the following proof rule $X \vdash A \wedge B$ gets applied to this pending subgoal $X, A, B \vdash A \wedge B$, then H will be instantiated with $\{A, B\}$. The instantiation of H will also be added to all antecedents of the proof rule. This ensures that the new pending subgoals are consistent.

10 Substitution operator

The substitution operator, which is part of *basicFoPCe* and *FoPCe*, is a fundamental operator that allows the user to apply substitutions. The implementation is largely based on the script of the PrfM course [9, p. 48].

10.1 Known limitations

One restriction that is important to keep in mind is that only one substitution can be applied per step. To completely apply the substitution $[x := E](\top \wedge \perp)$ a total of three substitutions are necessary. The individual steps can be seen in figure 3.

$$\begin{aligned}
 & [x := E](\top \wedge \perp) \\
 \cong & ([x := E]\top) \wedge ([x := E]\perp) \\
 \cong & \top \wedge ([x := E]\perp) \\
 \cong & \top \wedge \perp
 \end{aligned}$$

Figure 3: Steps of a substitution

Allowing multiple substitutions at a step would make the source code much more complicated and could potentially confuse the user. For those reasons it was not implemented.

10.2 Applying a substitution

Applying a substitution, i.e. substituting a bound variable for its replacement, happens when a user clicks on the substitution operator. To uniquely identify the location of the substitution a node and a path are necessary. The node identifies the sequent of the proof tree and the path identifies the term inside the sequent. The node is structured as described in section 6.

10.3 Alpha conversion

Certain substitutions require non-freeness side conditions. More information on side conditions and their implementation can be found in section 11. One such non-freeness side condition is present in the $\widehat{=}_{[:=\forall_2]}$ substitution:

$$\begin{aligned}
 [x := E](\forall y.P) & \cong \forall y.([x := E]P) \\
 & \text{if } x, y \text{ are distinct and } (y \text{ } \underline{\text{nfin}} \text{ } E)
 \end{aligned}$$

The following example illustrates the necessity of the side condition: The substitution $[x := y](\forall y.x)$ should not result in $\forall y.y$ as this would bind the free variable y by mistake. The correct result of the substitution is $\forall y.x$. In general, the bound variable y of the \forall quantification has to be renamed before substituting the term E if the side condition check fails [9, p. 49].

The following code listing 8 shows how such an α -conversion could be implemented.

```

replaceBoundVariable : String -> String -> Term -> Term
replaceBoundVariable current replacement term =
  case term of
    Variable variable ->
      if variable == current then
        Variable replacement

      else
        term

    Application left right ->
      Application
        (replaceBoundVariable current replacement left)
        (replaceBoundVariable current replacement right)

    Abstraction parameter body ->
      Abstraction
        (replaceBoundVariable current replacement parameter)
        (replaceBoundVariable current replacement body)

    InParentheses termInParentheses ->
      InParentheses
        (replaceBoundVariable current replacement termInParentheses)

  _ ->
    term

```

Listing 8: Simple α -conversion

11 Side Condition

Side conditions are a fundamental part of many calculi. For instance, they occur in the $\forall hyp$ proof rule schema, which is part of *basicFoPCe*. New calculi could introduce different kinds of side conditions. Supporting such conditions requires the implementation to be generic.

11.1 Instantiation of side conditions

Side conditions are instantiated as soon as possible. This means that side conditions on meta variables are instantiated whilst instantiating the proof rule schema itself. Yet, side conditions on the implicit hypothesis H can only be instantiated after the proof rule is applied to a pending subgoal. This is because only at that point the instantiation of H is available.

As soon as a side condition is instantiated, the user-supplied instantiations are applied to the function represented by an *UninstantiatedSideCondition* and the result is saved as an *InstantiatedSideCondition*. These type constructors can be seen in listing 9.

```
type SideCondition dictContent
  = InstantiatedSideCondition Bool
  | UninstantiatedSideCondition (Instantiations dictContent -> Bool)
```

Listing 9: Side condition type

11.2 Checking of side conditions

All side conditions are checked at the same time, namely as part of the precondition check in the application algorithm, even though most side condition results are already known earlier. A more detailed description of the application algorithm can be found in section 9. One advantage of evaluating the side conditions simultaneously is that there are not multiple, possibly confusing, error messages.

11.3 Definition of side conditions

Side conditions need to be defined as part of a proof rule schema. The type alias for a side condition can be seen in listing 10.

```
type alias SideConditions =
{ implicitHypothesis : List (SideCondition (List Term))
, normal : List (SideCondition Term)
, name : String
}
```

Listing 10: Side conditions type alias

In addition to the side condition functions, a name can be defined, which will be appended to the proof rule schema name. We decided to omit the side condition name in the proof tree, as it is no longer of any concern to the user at that point. It would not have been trivial to implement this feature. Long proof rule names tend to overlap with other content in the proof tree and replacing meta variables with the string representation of their instantiation is not straightforward either.

11.4 Limitation of the genericity

Listing 9 shows that side conditions can base their truth value only on instantiations. Side conditions that evaluate something else cannot be represented in The Sequent Calculus Calculator. This should not be a problem as the calculi presented as part of PrfM do not have such side conditions.

Usually meta variables represent exactly one term. However, the implicit hypothesis H can match multiple terms. Therefore, these two instances of side conditions need to be handled separately. Listing 10 shows that The Sequent Calculus Calculator differentiates between *normal* and *implicitHypothesis* side conditions. With this restriction in mind, it is clear that an *UninstantiatedSideCondition* can contain two different side condition types. Listing 11 shows the two possible types of side condition functions.

```
sideConditionFunctionImplicitH : Instantiations (List Term) -> Bool
sideConditionFunction : Instantiations Term -> Bool
```

Listing 11: Side condition function type

If a developer of a calculus wants to add new kinds of side conditions checks, they have to implement both side condition functions of listing 11.

11.5 Supported side condition types

From the supported calculi only *basicFoPCe* and *FoPCe* require side conditions. In both calculi there are proof rule schemas that require a *non-freeness* check. Therefore, a *non-freeness* side condition was introduced, which can be seen in listing 12.

```
notFreeIn : String -> Term -> Bool
notFreeIn nonFreeVariable term =
  case term of
    Variable variable ->
      variable /= nonFreeVariable

    Application leftTerm rightTerm ->
      notFreeIn nonFreeVariable leftTerm
      && notFreeIn nonFreeVariable rightTerm

    Abstraction (Variable parameter) body ->
      if parameter == nonFreeVariable then
        True

      else
        notFreeIn nonFreeVariable body

    InParentheses inParenthesis ->
      notFreeIn nonFreeVariable inParenthesis

  - ->
    False
```

Listing 12: Implementation of a non-freeness side condition

The algorithm is a recursive traversal of the term structure. It follows the rules presented in the script of the PrfM course [9, p. 47, 48]. As meta variables are not allowed to be present during a *non-freeness* check, a term containig one will always fail that check.

12 Proof tree export and import

To allow saving and loading proof trees, an export to and an import from JSON is available to the user. The implementation, the format of the JSON, as well as restrictions and problems with this functionality are part of this section.

12.1 Implementation

The encoding of the proof tree data structure to a JSON string and decoding it back to the data structure is implemented using `elm/json`¹².

To download and upload files `elm/file`¹³ is being used. Implementing a file download with this library is straightforward. The file upload, on the other hand, is much more complicated. The upload functionality requires three Elm messages, as each step of the upload process is called individually. One is called when requesting a JSON file to be uploaded, another one after the user selects the file and the final message is sent as soon as the upload completes. The last message contains the content of the selected file as a string. The required messages can be seen in listing 13.

```
ImportProofTreeRequested
ImportProofTreeSelected File
ImportProofTreeLoaded String
```

Listing 13: Import messages

12.2 Format of the JSON

The chosen format is a simple JSON representation of the Higher-order abstract syntax. Because of that, new calculi are automatically able to use this feature without any additional implementation effort. The structure of a proof tree can be seen in listing 14.

```
{
  "modulId": String,
  "proofTree": {
    "path": [ Int ],
    "sequent": {
      "hypotheses": [ Term ],
      "goals": [ Term ]
    },
    "appliedProofRule": {
      "appliedProofRuleName": String,
      "children": [ ProofTree ]
    }
  }
}
```

Listing 14: Proof tree JSON structure

¹²<https://package.elm-lang.org/packages/elm/json/latest/>

¹³<https://package.elm-lang.org/packages/elm/file/latest/>

As there are four different term constructors, each one of them needs its own JSON structure, which are displayed in listings 15, 16, 17 and 18. The type attribute is always the name of the term constructor. Meta variables can never be part of a proof tree and they do not need to be represented as part of the JSON structure.

```
{  
  "type": String,  
  "value": String  
}
```

Listing 15: Variable JSON structure

```
{  
  "type": String,  
  "left": Term,  
  "right": Term  
}
```

Listing 16: Application JSON structure

```
{  
  "type": String,  
  "parameter": Term,  
  "body": Term  
}
```

Listing 17: Abstraction JSON structure

```
{  
  "type": String,  
  "term": Term  
}
```

Listing 18: InParentheses JSON structure

12.3 Emerging problems

12.3.1 Import checks

The current solution does not check if the specified proof tree is valid or not.

It does, however, check if the generated proof tree is of the exact same calculus as the currently active one. If this is not the case, the decoding of the JSON fails. This one-to-one check was chosen in favour of an “inheritance”-type check, even though some calculi are completely contained in other ones (e.g. *basicPC* in *PC*). The reason for this is its simpler implementation and because the initial sequent of a proof determines the rules that can be applied to a proof. If a user begins his proof in *basicPC* they will never switch to *PC* during the course of their proof.

It is also important to mention that a malicious user could circumvent this protection by editing the JSON file directly. However, our concerns lie with the protection of the average user, for whom this approach is sufficient.

12.3.2 LCF-style circumvention

Allowing a user to specify and load his own arbitrary proof tree results in a circumvention of the LCF-style. A user could potentially upload a proof tree that’s not proven up to that point. This circumvention is possible because the import and export is directly part of the kernel. Even though this is a very serious problem and could potentially lead to broken proof trees, it is also a highly sought after feature. It should primarily be used to exchange or save proof trees and not to write them in JSON by hand.

13 User Interface

This section describes all UI components and their interactions with each other in more detail. The architecture of the frontend is part of section 4.5 and is not described here.

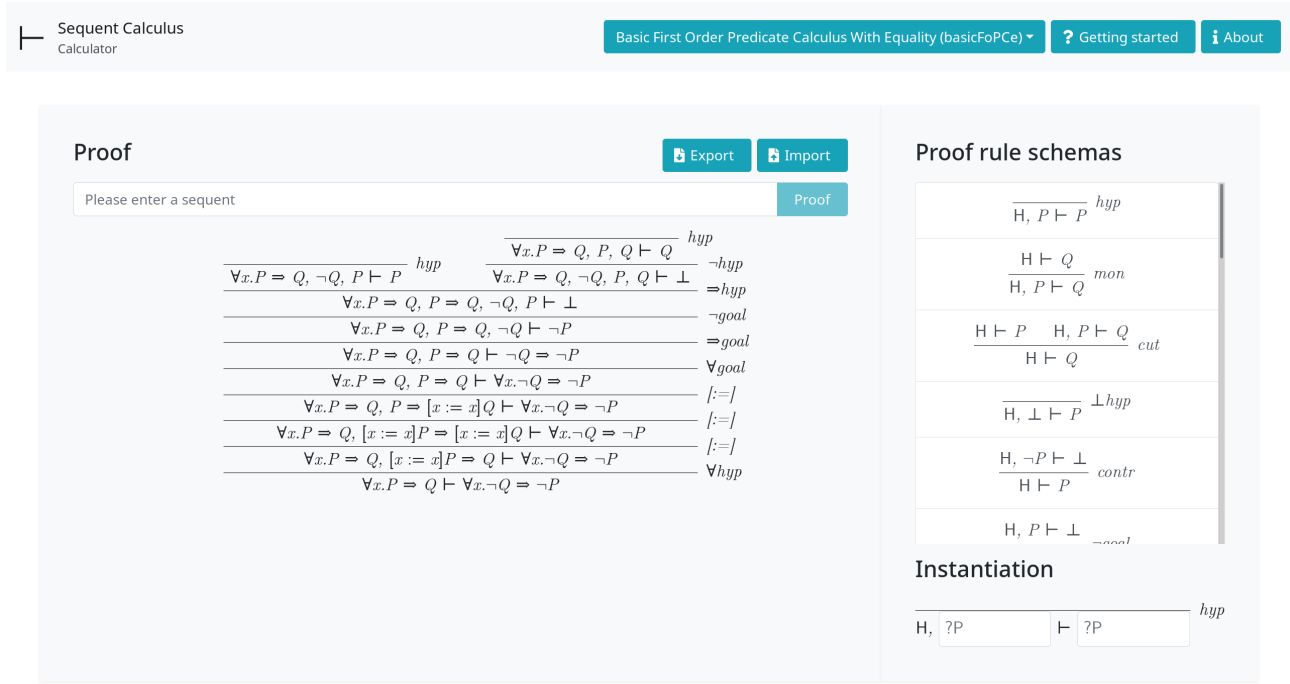


Figure 4: Entire webapplication

13.1 Special character encoding

To make the input fields user friendly, the user does not have to enter Unicode characters of operators. Instead, predefined keywords can be used. These are defined in calculus specific Elm modules, as operators vary between calculi. As soon as a keyword is detected, it gets converted to the corresponding Unicode symbol. For example *basicPC* contains the following special characters:

\neg → !
 \wedge → ^
 \neg → \not
 \wedge → \and
 \perp → \true

13.2 Rendering algorithms

There are three different rendering algorithms in The Sequent Calculus Calculator. One is used to display the proof tree data structure in a tree form to the user. Another one creates the proof rule schema view and the instantiation view. The last one is module dependent and is responsible for constructing an HTML structure from a term.

13.2.1 Proof tree renderer

The proof tree renderer is independent of the module and uses the term renderer to complete its task. It creates a tree structure by nesting HTML `div`s into one another. This structure is later styled using CSS, which makes it look like the trees seen during the PrfM course. The fully proven sequent $A, B \vdash A \wedge B$ can be seen in figure 5.

$$\frac{\frac{}{A, B \vdash A} \text{hyp} \quad \frac{}{A, B \vdash B} \text{hyp}}{A, B \vdash A \wedge B} \wedge \text{goal}$$

Figure 5: Rendered proof tree

13.2.2 Proof rule schema renderer

Just like the proof tree renderer, the proof rule schema renderer is module independent and uses the term renderer to fulfill its task. In contrast to the proof tree HTML structure, the proof rule schema does not need nested HTML elements. A proof rule schema is not a recursive structure after all.

13.2.3 Term renderer

The term renderer is module dependent, as different modules have different conventions and need to call different parsers based on the term structure. Because the term renderer is used in the proof rule schema list view, the instantiation view and the proof tree view, it needs to be highly generic. As a result, it accepts a lot of parameters for configuration and has to fulfill different purposes.

The following list contains the tasks that the term renderer needs to fulfill:

- The renderer has to create the path representation of rendered terms and add them in case they are required to identify terms in the proof tree.
- It has to support the rendering of meta variables in a dynamic way, as they are represented as input fields in the instantiation view and as normal strings in the proof rule schema list view.
- It has to call the correct parser for meta variables. Note that expressions, predicates and variables do not use the same parser.

Term to string renderer

Besides the term to HTML renderer, there is also a term to string version. This functionality can be used on the CLI.

13.3 Webpage header

The webpage header contains a dropdown menu, to allow the user to switch between calculi. The selection is saved in the local storage¹⁴ of the browser and loaded upon startup. The Sequent Calculus Calculator detects which calculus was used during the previous session and loads the current proof rule schemas accordingly.

¹⁴<https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>

There is also a getting started guide for new users where they can learn the usage of the web application. It also contains a list of all special character encodings available as part of the selected calculus. More on special character encodings can be read in section 13.1.

Lastly, there is an about button containing information on the thesis itself and a link to the git repository, to encourage people to add new modules and improve the user interface.

Both, the getting started guide and the about page, are implemented using bootstrap modals¹⁵.

13.4 Proof rule schema view

The proof rule schema view displays all proof rule schemas of the active calculus. The schemas are loaded on startup and can be changed by selecting a different calculus from the dropdown menu. Section 4.5.2 described these calculus modules. Figure 6 shows the proof rule schema list view.

Proof rule schemas	
$\frac{}{H, P \vdash P} \text{hyp}$	
$\frac{H \vdash Q}{H, P \vdash Q} \text{mon}$	
$\frac{H \vdash P \quad H, P \vdash Q}{H \vdash Q} \text{cut}$	
$\frac{}{H, \perp \vdash P} \perp \text{hyp}$	
$\frac{H, \neg P \vdash \perp}{H \vdash P} \text{contr}$	
$\frac{H, P \vdash \perp}{H \vdash \neg P} \neg \text{goal}$	
$\frac{H \vdash P}{H, \neg P \vdash Q} \neg \text{hyp}$	
$\frac{H \vdash P \quad H \vdash Q}{H \vdash P \wedge Q} \wedge \text{goal}$	
$H, P, Q \vdash R$	

Figure 6: Proof rule schema view

Presentation

As previously mentioned, the rendering of proof rule schemas does not use a rendering library that is able to convert L^AT_EX to HTML because of reasons discussed in section 3.5.

¹⁵<https://getbootstrap.com/docs/4.0/components/modal/>

To achieve the desired UI the following CSS components are used:

- The antecedents are spread evenly by the flexible box module (Flexbox) [15].
- The horizontal bar between the consequent and the antecedents is a border on the top side of the consequent.
- The proof rule name is placed to the right of the `div` on the same height as the vertical bar.

Interaction

By clicking on a proof rule schema it gets loaded as an instantiatable version into the instantiation view.

13.5 Instantiation view

The instantiation view displays the selected proof rule schema. The user can instantiate meta variables, i.e. change meta variables to concrete terms. Figure 7 shows the instantiation view with an instantiated $\wedge goal$ proof rule schema.

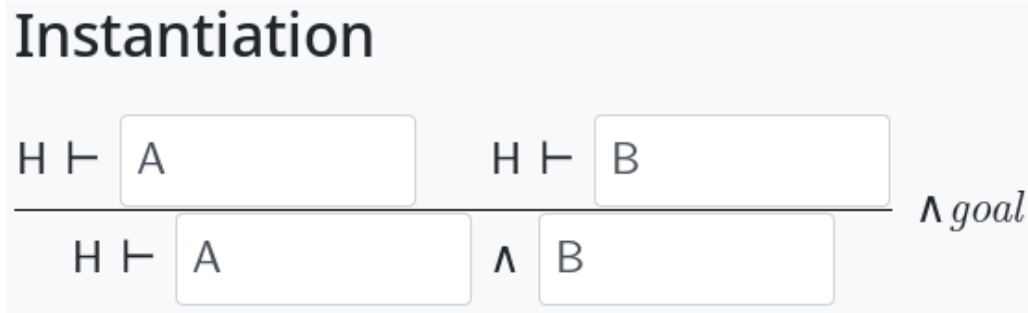


Figure 7: Instantiation view

Presentation

The instantiation view uses the same rendering algorithm as the proof rule schema view. The only difference between them is that the instantiation view renders meta variables as input fields.

Upon each user input, the inserted text gets parsed by the term parser of the loaded module. If this parsing process fails, the user gets a notification in form of a tooltip, indicating that his inserted term is not valid. This tooltip is removed as soon as the term is parsable again or the user changes the input field. In addition, the border of the input field turns red, as long as the term inside the input field is not parsable. Figure 8 shows the instantiation view with an invalid meta variable instantiation.

Interaction

Using input fields, the user can replace meta variables with terms. Input fields representing the same meta variables are coupled to each other, i.e. changing one input field also changes all connected fields.

As soon as all meta variables are instantiated, the user can drag the proof rule over a pending subgoal of the proof tree. Dropping the proof rule results in an application of the proof rule to the pending subgoal.

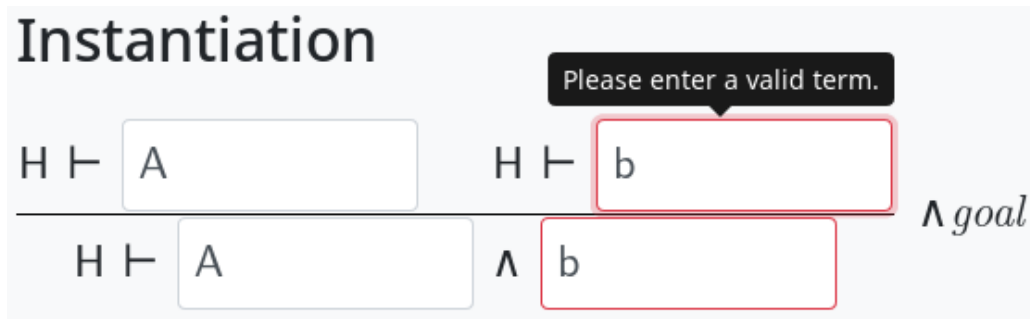


Figure 8: Instantiation view with an error

13.6 Proof view

The proof view consists of four components. An export button, an import button, the sequent input field and the proof tree view. The proof tree view is responsible for displaying the proof tree. If the user has not yet inserted a sequent to be proven, a placeholder text is displayed instead. Figure 9 shows the proof view with a partially proven proof tree.

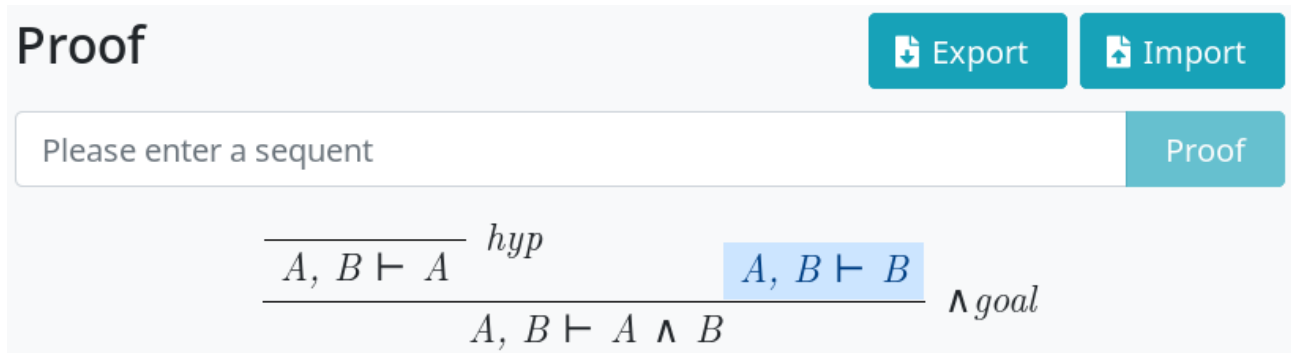


Figure 9: Proof view

Presentation

The import and export buttons are used to save the current proof tree and load an old proof tree respectively. More information about exporting and importing proof trees can be found in section 12. The export button is disabled, as long as the user has not yet started a proof. It would not make sense to export an empty proof.

The component with the sequent input field is made up of the input field itself and a proof button to the right. The input field itself has the same functionality as the ones of the instantiation view of section 13.5. The only difference is the used parser, as in this case the user input needs to be an entire sequent and not just a single term. The proof button is disabled, as long as the sequent is not parsable.

Pending subgoals are treated differently, as they are highlighted through a blue background to give the users a visual aid of what they can proof next. On dragging a proof rule over a pending subgoal, its background color changes to green to indicate the drop location.

Interaction

After the users specify a sequent and click on the proof button, the sequent gets displayed in the poof tree view as the only pending subgoal and the placeholder text disappears.

14 Testing

The application logic is tested with the help of `elm-explorations/test`¹⁶. The frontend is not automatically tested because we know from experience that this requires a lot of effort and has little benefits. The folder structure of our tests mirrors the one of the system under test.

14.1 Test dependencies

The term renderer tests and parser tests are based on each other, i.e. the *PC* tests are built on top of the *basicPC* tests and so forth. This avoids unnecessary code duplication of the test modules.

14.2 Fuzzing

All modules use a fuzzer to test the term to string rendering. The following property has to hold for any term t : $t = \text{parse}(\text{termToString}(t))$. This is important because it allows the user to copy displayed text and paste it into input fields. The function *parse* removes superfluous whitespaces so it is not the exact inverse function of *termToString*.

Generating random terms is not a simple task. The following example shows how unary operations are generated for *basicPC*:

```
unaryOperationGenerator : Generator Term
unaryOperationGenerator =
  Random.map2
    (\operator operand ->
      createUnaryOperation operator operand
        |> InParentheses
    )
    (Random.constant "¬")
    (Random.lazy (\_ -> termGenerator))
```

Listing 19: Unary operation generator

¹⁶<https://github.com/elm-explorations/test>

The same process is implemented for all term structures currently available. For instance, to generate an expression, the following generator is used:

```
expressionGenerator : Generator Term
expressionGenerator =
    Random.andThen
        (\b ->
            if b < 9 then
                Random.lazy (\_ -> variableExpressionGenerator)

            else
                Random.lazy (\_ -> functionExpressionGenerator)
        )
    (Random.int 0 10)
```

Listing 20: Expression generator

Note that it uses *Random.lazy* to force a lazy evaluation of the expressions. In nine out of ten cases the first and simpler case is chosen. This is because Elm runs out of stack space when using too many recursive function calls.

14.2.1 Shrinker

A fuzzer is usually combined with a shrinker which simplifies the random data in case a test fails. Its purpose is to make it easier for the developer to identify the failing component. Let us assume the following term is generated by the fuzzer:

$$\text{Application}(\text{..})(\text{Application}(\text{Variable}''?'')(\text{Variable}''P''))$$

It might not be parsed correctly, because of the question mark used as a variable as part of the application. The left side of the outer application might consist of multiple nested terms.

The shrinker would now take this whole term and find simpler terms that also fail the test. As a developer I would then not have to search for the failing input, because I would receive $\text{Application}(\text{Variable}''?'')(\text{Variable}''P'')$ directly. A shrinker is therefore nothing but a simplifier.

Unfortunately, there is an unresolved issue¹⁷ that prevents developers from writing their own shrinkers. At the moment a term is not simplified and identifying the failing component can be difficult.

14.3 System tests

To verify that the application works as a whole, we performed system tests at the end of the implementation phase. This section describes the tests and their results. All tests described in this section are performed manually on each of the implemented calculi. Of course, some tests require a specific calculus and will therefore not be performed on the other ones.

¹⁷<https://github.com/elm-explorations/test/issues/66>

Nr.	Description	Expected result	Applicable calculi
1	Enter the sequent $P \wedge Q \wedge O \vdash P$ into the sequent input field.	The sequent should not parse and an error message should be displayed.	Any
2	Enter the sequent $P \wedge (Q \vee O) \vdash P$ into the sequent input field.	The sequent should parse and clicking the Proof button should display the new subgoal.	$PC, basicFoPCe, FoPCe$
3	Show that conjunctions are associative by proving $P \wedge (Q \wedge O) \vdash (P \wedge Q) \wedge O$.	It should be possible to prove the sequent.	Any
4	Apply <i>contr</i> , <i>¬hyp</i> and <i>hyp</i> (in this order) to the sequent $\neg\neg\perp \vdash \top$.	The sequent should be proven.	Any
5	Enter and proof $\forall x.H(x) \Rightarrow M(x), H(s) \vdash M(s)$.	It should be possible to proof the sequent.	$basicFoPCe, FoPCe$
6	Enter the sequent $x \vdash \forall x.P$ and apply <i>∀goal</i> .	The side condition should fail and an error message should be displayed.	$basicFoPCe, FoPCe$
7	Enter the sequent $\vdash [x := y]\forall y.x$ and perform a substitution on the goal.	The new pending subgoal should be $\vdash \forall y_1.[x := y]x$.	$basicFoPCe, FoPCe$

Table 1: System tests

The cases presented in table 1 verify that the core behaviour of the application is correct. Although the aforementioned test cases test the system thoroughly, several exercises that were set during the PrfM course were also solved using the calculator. Using the calculator it should be possible to proof all exercises of *basicPC*, *PC*, *basicFoPCe* and *FoPCe*. Proving the validity of proof rule schemas is currently not possible with the application, because syntactic rewriting rules are not part of the application.

15 Implementing a new calculus

How a new calculus can be implemented and integrated in the current application is described in this section. Each calculus is a separate module which is represented using the following data type:

```
type alias Module =
  { name : String
  , id : String
  , proofRuleSchemas : List TrustedProofRuleSchema
  , termToHtml :
      Maybe (Term -> ParseFunction Term -> Html Message)
      -> (Html Message -> List Int -> Html Message)
      -> List Int
      -> Term
      -> Html Message
  , termToString : Term -> String
  , parse : ParseFunction Term
  , parseTerm : Parser Term
  , specialCharacters : SpecialCharacters
  , operators : Operators
  }
```

Listing 21: Primary module data structure

This data type can be used as a checklist when implementing a new calculus. Providing all functions in their correct type is already a good indicator as to whether the module is correctly implemented.

15.1 Name and identifier

Each calculus module has to contain a name and an identifier. The former is a text representation that does not have to be unique, whereas the latter has to be unique as it is used internally to match the calculus.

15.2 Parser

The first thing one should write when adding support for a new calculus is the parser. This step is arguably the most tedious one. The parser has to expose a function of the following type, where element is of type *Term*:

```
type alias ParseFunction element =
  String -> Result (List DeadEnd) element
```

Listing 22: ParseFunction type

The *parse* function takes the user input and converts it into a *Result* that either contains a *List* of *DeadEnds* or the parsed *Term*.

To make the parsing of operators a bit easier, the Elm package `Punie/elm-parser-extras` can be used [13]. It handles the precedence of operators and supports both binary and unary operators. To use it, simply add an operator table and then use the function `buildExpressionParser`. The operators that appear before others in the list have higher precedence.

```
operators : OperatorTable Term
operators =
  [ [ prefixOperator "¬" ]
    , [ infixOperator "^" ]
  ]

term : Parser Term
term =
  buildExpressionParser operators (lazy <| \_ -> variable)
```

Listing 23: BasicPC operators

15.3 Proof rule schemas

The next step is to add the proof rule schemas of the calculus. They have to be added to the file `Kernel/TrustedProofRuleSchema.elm` as that is the only place where the corresponding constructor is visible. More information on why this restriction exists can be found in section 6.

15.4 Term renderers

Now the term renderers can be written. There has to be a renderer that converts *Terms* to HTML and one that converts them to text. The type of the former is slightly different:

```
termToHtml :
  Maybe (Term -> ParseFunction Term -> Html Message)
  -> (Html Message -> List Int -> Html Message)
  -> List Int
  -> Term
  -> Html Message
```

Listing 24: Type of termToHtml

The first three parameters require some explanation:

- The first parameter defines how meta variables are rendered. This has to be configured outside of the renderer and that is why this function is passed as a parameter. If it is a *Nothing* then the meta variable will simply be displayed as text. If the term is rendered as part of an instantiation, the meta variables are represented as input fields. The second part of the first parameter is the parser that should be called when rendering the meta variable. This part is defined by the current renderer. For instance, in *basicFoPCe* a meta variable *E* represents an expression, whereas *P* represents a predicate.
- The second parameter is a decorator that is called on substitution operators. *basicFoPCe* uses it to add *onClick* listeners for substitutions.

- The third parameter is the path of the proof tree that was taken before this *Term* was reached. This path is there to uniquely identify the *Term* inside the *ProofTree*. It is constructed similarly to the one of the *Node* which is described in section 6.

15.5 Special character encoding

The final step is to define which special characters are part of the calculus. These definitions will be used to replace occurrences of special characters in user input.

```
specialCharacters : SpecialCharacters
specialCharacters =
    Dict.fromList
        [ ( "\\and", "^" )
          , ( "&", "^" )
          , ( "\\not", "¬" )
          , ( "!", "¬" )
          , ( "\\false", "⊥" )
        ]
```

Listing 25: Special characters of BasicPC

The above listing defines that `&` should be replaced with \wedge when encountered in user input.

16 Deployment

`elm-live`¹⁸ is used to deploy the application locally. It compiles the Elm code and serves static files through a web server. Furthermore, it reloads the website automatically as soon as an Elm module changes. This can be quite helpful during live testing. Elm provides a similar utility called `elm-reactor`¹⁹ which does not support auto-reloads.

In general, an Elm application is built using `elm-make`²⁰. It compiles Elm modules to JavaScript but does not publish them to a web server.

Deploying the application is fairly simple. After compiling the application to a single JavaScript file, it has to be served with all static files (HTML, JS, acsrshortcss) using a web server of your choice. The current solution uses GitLab pages for this task. The configuration that describes how the code is built looks like this:

```
build-website:
  stage: build
  image: trenneman/elm-ci
  script:
    - cd code
    - elm make src/Main.elm --output=build/src/out/elm.js
    - cp src/index.html build/
    - cp -r src/stylesheets/ build/src/
    - cp -r src/images/ build/src/
  artifacts:
    paths:
      - code/build
```

Listing 26: Website build plan

¹⁸<https://github.com/wking-io/elm-live>

¹⁹<https://guide.elm-lang.org/install.html#elm-reactor>

²⁰<https://guide.elm-lang.org/install.html#elm-make>

17 Evaluation

The results of the thesis are described in this section. We will begin by describing what we were able to implement as part of the project and what not. Furthermore, we will discuss the problems that we came across during the development of The Sequent Calculus Calculator and what the consequences of the LCF-style are. Finally we will draw a conclusion.

17.1 Implemented features

In this thesis we implemented a fully functional web application that is able to proof sequents of multiple calculi in a drag and drop proof style. The application can and hopefully will be used to teach *basicPC*, *PC*, *basicFoPCe* and *FoPCe* as part of the PrfM course. The Sequent Calculus Calculator is able to support side conditions as well as substitutions, which are both part of *basicFoPCe* as well as *FoPCe*.

The developed web application does not need a web server at all. It is a simple single-page application that works on the client-side. The deployment is also straightforward as can be seen from section 16.

The kernel uses an LCF style and is extensible. Through strict code reviews we tried to achieve a very high code quality. This project can be used to introduce students to functional programming. Elm is very beginner friendly and can easily be understood. It is clear that due to its genericity, the kernel might not be as simple as it could have been if we used one kernel per calculus. This was a decision that made the project more extensible from a long-term perspective but also less suitable for didactic use (to teach functional programming).

To make sure that the application works as intended, we performed system tests, which are described in section 14.3. The kernel and the parser were tested through unit tests. To assure that the application is intuitive to use, a usability test was conducted. From there we gained valuable feedback which we described in section E.2. The fuzzer tests that are described in section 14.2 helped us identify failing cases that we would have never found through manual or simple unit testing.

The Sequent Calculus Calculator allows one to switch between calculi at runtime. The users do not have to visit a new webpage to proof a sequent in a different calculus. All available calculi are listed in a dropdown menu.

It is possible to export the current proof state to JSON. This format was chosen because Elm already supports this encoding through `elm/json`²¹. The implementation was straightforward and worked without any problems. Other formats are currently not supported. \LaTeX , for example, could be added to the supported export formats in the future. However, this would require a new kind of renderer, i.e. one that converts terms to \LaTeX , as well as a new parser.

²¹<https://package.elm-lang.org/packages/elm/json/latest/>

17.2 Features that were not implemented

We did not implement other calculi than the ones mentioned above. For instance the lambda calculus was not implemented in The Sequent Calculus Calculator. Implementing *basicFoPCe* took longer than we expected as we ran into multiple problems which will be described in the next section.

During the implementation of The Sequent Calculus Calculator we decided to not support syntactic rewrite rules as they would introduce a handful of new challenges. Supporting them in our current kernel would probably not be too difficult. But they need to be added to the current user interface as a new concept. Due to our limited time we chose to first implement everything else and then focus on this feature. A consequence of the missing syntactic rewrite rules is that a type of exercise of the PrfM course cannot be solved. Exercises that ask the reader to prove proof rules based on other proof rules can only be partially solved with The Sequent Calculus Calculator.

Substitutions have been implemented as part of proof rule schemas. Users cannot enter them during instantiations though. This limitation is not too problematic as it does not prevent the user from proving vital sequents. Because of this, two exercises of the PrfM course that cannot be solved using The Sequent Calculus Calculator. We did not implement this because it would make the parsing step more complicated. In section 4.4 we described that we use nominals instead of de brujin indices. These could also become problematic when allowing the user to enter substitutions of the form $[x := E]P$.

Automatic unification is not part of The Sequent Calculus Calculator. We decided to focus on the important features to make the web application usable for teaching at the HSR. We accepted the downside of having to enter sequents manually during instantiation.

Finally, we did not write a technical paper that is suitable for academic publication. However, the technical documentation tries to be a replacement for the missing paper.

17.3 Problems

17.3.1 Renderer

The renderers of *basicFoPCe* and *FoPCe* are currently the most complicated pieces of code of our project. Refactoring them is, from our point of view, not possible at the moment as it would require a lot of effort. Unfortunately, those renderers play an important role and perform multiple tasks at the same time:

- They render terms to HTML and thereby follow certain calculus specific conventions.
- They call the correct parsers when used in the instantiation view.
- They calculate paths that are needed to match *Terms* in *ProofTrees*.

Ultimately, this lead to some code duplication as all renderers convert similar structures to HTML. Furthermore, the *basicPC* and *PC* renderers do not use all the parameters that are passed to them. Elm requires us to use one and the same signature for all of them though. As this is an important problem, we decided to add it to the development roadmap that is described in section F.

17.3.2 GitLab free plan

The GitLab free plan offers only 2,000 CI pipeline minutes per group and month [12]. As we did not know about this restriction, we ran into the problem of using up all of our free hours. We were no longer able to deploy our application and had to work with a local version for about a week. This risk occurrence and how we solved the problem for the future is described in section C.1.

17.3.3 Styling-problem with Safari

During our tests with different browsers, we discovered a bug in the proof view when using Safari. As it turns out, Safari does not support the CSS value `relative` for the property `position` on an HTML table row element in the same way as Firefox does. Therefore, the name of the applied proof rule was positioned relative to the first `div` element with the previously described `position` property. In our case, this was the `div` containing the entire proof tree view.

Solution

The solution to this bug was quite simple but it took us a while to find it. It turns out that setting the property `display` to the value `block` prevented this bug from happening in Safari. The following listing shows the fix:

```
.proof-tree .subgoal:last-child .proven-sequent {  
  position: relative;  
  /* Needs to be set to fix a display bug in Safari */  
  display: block;  
}
```

Listing 27: CSS bug fix for Safari

This example shows that supporting multiple browsers can become very challenging. Different behaviour in different rendering engines is not an uncommon occurrence.

17.4 Consequences of the LCF-style

The LCF-style allows us, as developers, to be sure that no invalid proofs can be constructed. For instance, a user with malicious intent is not able to create proofs that are not valid with respect to the sequent calculus or the current active calculus. Note that this is not entirely true due to the import function that has been added to the kernel directly. It allows a user to upload unverified proofs. In the future, some kind of check could be implemented that prevents users from doing that though.

The LCF-style forces the developer to not expose type constructors of their protected types. This leads to very large modules, as code that has to use such a type constructor suddenly becomes part of that module. In our case the *ProofTree* and the *TrustedProofRuleSchema* modules are prime examples of that problem.

Another consequence of the LCF-style is that proof rule schemas have to be defined in the kernel itself. The constructor is only available as part of the kernel. This makes the design somewhat awkward as one would think that they should be part of their individual module.

Overall, the LCF-style is very helpful as it gives the project stability and allows us to argue about types with a certain confidence. The current import undermines this and should therefore be fixed as soon as possible.

17.5 Conclusion

This thesis shows that it is possible to implement an extensible web based sequent calculus calculator. The developed application can be used as a didactic aid and supports almost all of the requested features. Thanks to the generic kernel, it is possible to switch between calculi at runtime without any problems. The convenient export and import feature allows users to share proofs. Moreover, supporting a new calculus is also possible without changing the existing kernel.

The genericity of the kernel will definitely pay off in the future when new calculi are added. Of course, it took us a lot of time to get to this point but the reuse factor that we get now is much greater compared to an implementation that uses one kernel per calculus.

It is clear that there are some limitations to the application that can make the process of proving a sequent cumbersome. For instance, the missing unification can be a bit frustrating. As this tool is intended to be used as a didactic aid to teach students the basics of the sequent calculus, we think that the current solution is satisfactory.

Overall, we would say that the outcome of the project is a well-engineered application that shows great potential for future development. We cannot wait to see how this tool is used in the upcoming semester and how it will improve over time.

Appendix

A Task description



Task Description – Bachelor Project „The Sequent Calculus Calculator“ FS 2019

1. Client and Supervisor

- *Client:* Institute for Software, HSR
- *Supervisor:* Prof. Dr. Farhad Mehta

2. Students

- Matteo Kamm
- Mike Marti

3. Setting

The sequent calculus is the most frequently used representation to specify and reason about deduction systems (such as those used for proof and type checking) in computer science. Although it is a simple, formal, syntax-based technique, the use of paper-based proofs to learn the sequent calculus is often tedious (since there is a lot of copying involved) and error prone (since paper-based proofs cannot be mechanically checked).

A number of online web-based interactive provers currently exist. Although well made, upon closer inspection it was found that none were ideal to be used as a didactic aid at the HSR due to the following reasons:

1. Logitext (<http://logitext.mit.edu/tutorial>): The application of proof rule schemas is linked to the syntax of the formula. This makes the process of proving a sequent different from how it is done on paper. The user can no longer try to apply different rules and see why they are not applicable, nor is possible to have a UI action for rule application in general, making implementing new theories in this setting difficult.
2. The incredible proof machine (<http://incredible.pm>): Supports multiple theories, but proof rule schemas are in the style of natural deduction, and proofs in the form of DAGs.
3. Propositional sequent calculus prover (<https://www.nayuki.io/page/propositional-sequent-calculus-prover>): Only for propositional logic. Proofs are listed as indented sequences and not trees.
4. SeqProver (<http://bach.istc.kobe-u.ac.jp/seqprover/>): Supports multiple proof output formats, but only works for predicate calculus. Unclear if it can be extended. Is not an interactive prover.

Given the possible widespread use of the sequent calculus, there is great potential for a well designed and engineered sequent calculus based interactive prover to be used for teaching and learning worldwide.

4. Goals

The main aim of this project is to design and develop an interactive web-based graphical online theorem prover to learn and play with the sequent calculus that outperforms existing solutions in the areas of usability, attractiveness, maintainability and extensibility. The application must:

1. Be able to support the construction of interactive sequent calculus style proofs in propositional logic, predicate logic, the simply typed lambda calculus, and the polymorphically typed lambda calculus.
2. Be easy and intuitive to use for teaching and learning.
3. Have an attractive user interface that supports a drag and drop style of proof along with explicit instantiation of meta-variables that cannot be determined using unification.
4. Require no prior installation (e.g. deployment as a single-page web-application) as far as possible.
5. Require no application server as far as possible.
6. Be maintainable and easily extensible programmatically to support reasoning other theories (e.g. other forms of type checking, term rewrite systems, ...).
7. Use a CI/CD pipeline for development and deployment.
8. Use a Haskell-based toolchain as far as possible.
9. Use an LCF-style proof kernel.
10. Take advantage of the good points in existing solutions.
11. Be able to be effectively used for teaching and learning at the HSR and internationally.

The students are expected to refine and modify these initial requirements, as well as specify additional requirements, during the course of the project in order to achieve its main aim. Possibilities of such additional requirements could be, that the application must:

1. Be able to indicate to the user, which rule schemas are applicable to the current goal.
2. Be able to save and load proofs.
3. Be able to export proofs in a variety of formats (LaTeX, ASCII, ...)
4. Be able to define (programmatically) and use (via the UI) proof tactics.

The logical kernel must be of high enough code quality to be used as a case study for functional programming in Haskell. To this end, the project should, as a warm-up, first start with refactoring the kernel (business logic, i.e. syntax and rewrite engine) of the existing lamdaCalc.io code, and then develop the logical kernel for this application.

The results should also be documented in the form of a paper suitable for academic publication.

5. Licence

The results of this project may be used and developed further without restriction by the students, the client, and the supervisor.

The effective regulations of the HSR and Department of Computer Science apply.

Rapperswil, 19.02.2019
Prof. Dr. Farhad Mehta

B Project plan

The project The Sequent Calculus Calculator will be realised iteratively with a sprint length of two weeks. Based on the 12 ECTS credits for the bachelor thesis, each project member should achieve roughly 42 hours of work per sprint.

B.1 Milestones

Milestone	Period	Project phase	Tasks
M1 Initialisation finished	18.02 - 04.03	Elaboration	Write project plan, setup project management tools, study documents, setup project homepage, sketch kernel API, analyse lambdacalc.io code
M2 Research conducted	05.03 - 18.03	End of Elaboration	Finalize kernel API description, implement kernel prototype, specify used technologies, define project architecture, conduct usability tests, setup CI/CD, create UI mockups, implement simple parts of the UI
M3 Kernel construction started	19.03 - 01.04	Construction Alpha	Implement stable kernel, add calculi modules
M4 Kernel construction finished	02.04 - 15.04	Construction Beta	Refactor kernel, add more calculi modules, implement UI, prepare intermediate presentation
M5 UI construction started	16.04 - 29.04	Construction Gamma	Hold intermediate presentation, refactor and improve kernel, finalize UI, prepare and hold first presentation
M6 UI construction finished	30.04 - 13.05	Construction Delta	Conduct usability tests with live system, improve UI based on usability test results, implement export logic, implement save/load logic
M7 Construction finished	14.05 - 27.05	Construction Release	Improve technical documentation, prepare presentation
M8 Project completed	28.05 - 14.06	Transition	Finalize documentation and scientific paper, hold presentation, create poster

C Risk analysis

Nr	Description	Max. amount of damage in hours	Probability of occurrence	Prevention measures
R1	Temporary downtime of GitLab	4h	20%	No opportunities of influence. Work with local a version.
R2	Temporary downtime of YouTrack	4h	20%	No opportunities of influence. Track time on a piece of paper.
R3	The project management and development tools are not powerful enough	5h	10%	Use them early on and discuss alternatives if problems occur.
R4	Scope of the project is too large	20h	40%	Work iteratively and dynamically change the specified scope based on the current state of the project.
R5	Used technologies need more read up time than expected	20h	40%	Reserve enough study time for the used technologies during the elaboration phase. Reserve extra hours for tickets that involve working with complex technologies.
R6	The kernel implementation takes up more time than expected due to its extension capabilities and its high complexity	30h	60%	Reserve enough time for the kernel API and project architecture definition. Keep the scope dynamic.
R7	The usability and intuitiveness of the UI are not as good as expected	30h	60%	Create UI mockups and conduct usability tests early on.
R8	The used libraries are impractical and hinder the project	10h	10%	Conduct research during the elaboration phase and use libraries in prototype projects.

C.1 Occurences

Occurences of risks will be tracked in this subsection.

Risk Nr	Timecost	Reason	Solution
R3	55m	The GitLab time tracking tool lacked some important features such as multiple assignees, time spending comments and a strong time evaluation tool.	The time and issue tracking was moved to YouTrack. All previous tickets and booked time had to be moved as well.
R1	-	Our free GitLab Runner time was exhausted during May. It was no longer possible to deploy / test the application.	We improved the build script to only build based on changes. Furthermore, the docker container is now only downloaded once per build and not once per L ^A T _E X build.
R4	-	The scope of the project was too large for a single bachelor thesis.	We chose to not implement all requested features. The reasoning behind our choice has been documented accordingly. A roadmap for follow-up projects has also been written.
R8	10h	Elm compares instances based on their structure and not based on identity. For comparisons of <i>Nodes</i> in the <i>ProofTree</i> this is not sufficient.	A path was added to each <i>Node</i> that makes the instance unique.

D EBNF descriptions

This section contains the EBNF descriptions of all calculi which are part of this bachelor thesis. Note that parentheses are not part of these descriptions. However, the parser handles them.

D.1 Basic propositional calculus

Predicate	=	UnaryOperator Predicate Predicate' Variable Predicate' Symbol Predicate'.
Predicate'	=	[BinaryOperator Predicate].
Variable	=	"A" ... "Z".
Symbol	=	" \perp ".
UnaryOperator	=	" \neg ".
BinaryOperator	=	" \wedge ".

D.2 Propositional calculus

Predicate	=	UnaryOperator Predicate Predicate' Variable Predicate' Symbol Predicate'.
Predicate'	=	[BinaryOperator Predicate].
Variable	=	"A" ... "Z".
Symbol	=	" \perp " " \top ".
UnaryOperator	=	" \neg ".
BinaryOperator	=	" \wedge " " \vee " " \Rightarrow " " \Leftrightarrow ".

D.3 Basic first-order predicate calculus with equality

Predicate	=	UnaryOperator Predicate Predicate' PredicateVariable Predicate' PredicateSymbol Predicate' Quantification Expression "=" Expression Relation.
Predicate'	=	[BinaryOperator Predicate].
Quantification	=	QuantificationSymbol ExpressionVariable "." Predicate.
Expression	=	ExpressionVariable Function.
Function	=	ExpressionVariable "(" { Expression } ")".
Relation	=	PredicateVariable "(" { Expression } ")".
PredicateVariable	=	"A" ... "Z".
ExpressionVariable	=	"a" ... "z".
PredicateSymbol	=	" \perp " " \top ".
QuantificationSymbol	=	" \forall ".
UnaryOperator	=	" \neg ".
BinaryOperator	=	" \wedge " " \vee " " \Rightarrow " " \Leftrightarrow ".

D.4 First-order predicate calculus with equality

Predicate	=	UnaryOperator Predicate Predicate' PredicateVariable Predicate' PredicateSymbol Predicate' Quantification Expression "=" Expression Relation.
Predicate'	=	[BinaryOperator Predicate].
Quantification	=	QuantificationSymbol ExpressionVariable "." Predicate.
Expression	=	ExpressionVariable Function.
Function	=	ExpressionVariable "(" { Expression } ")".
Relation	=	PredicateVariable "(" { Expression } ")".
PredicateVariable	=	"A" ... "Z".
ExpressionVariable	=	"a" ... "z".
PredicateSymbol	=	"⊥" "⊤".
QuantificationSymbol	=	"∀" "∃".
UnaryOperator	=	"¬".
BinaryOperator	=	"∧" "∨" "⇒" "⇔".

D.5 Simply typed lambda calculus

TypedLambdaTerm	=	LambdaTerm [":" SimpleType].
LambdaTerm	=	TermVariable Application Abstraction.
Application	=	LambdaTerm LambdaTerm.
Abstraction	=	"λ" Variable "." LambdaTerm.
SimpleType	=	TypeVariable TypeVariable SimpleType'.
SimpleType'	=	["→" SimpleType].
TermVariable	=	"a" ... "z".
TypeVariable	=	"α" ... "ω".

D.6 Polymorphically typed lambda calculus

TypedLambdaTerm	=	LambdaTerm [":" PolymorphicType].
LambdaTerm	=	TermVariable Application Abstraction.
Application	=	LambdaTerm LambdaTerm.
Abstraction	=	"λ" Variable "." LambdaTerm.
PolymorphicType	=	TypeVariable PolymorphicType' "∀" TypeVariable "." PolymorphicType PolymorphicType'.
PolymorphicType'	=	["→" PolymorphicType].
TermVariable	=	"a" ... "z".
TypeVariable	=	"α" ... "ω".

E Usability test with paper mockups

To improve the user interface, a usability test was carried out at the beginning of the project. The usability test was performed using paper mockups of the website. The goal of this test was to locate UI shortcomings and unintuitive behaviour of certain UI components and actions.

Setup

- Prepare all UI elements.
- Present the main window.
- Present all proof rule schemas.
- Present the sequent that needs to be proven.

Story

You are a student and are currently taking the PrfM course. The course is extra demanding and you need to put more time into it. Because you do not like wasting paper you try The Sequent Calculus Calculator, which was recommended by your teacher. Therefore you go to the website and try to prove a sequent you had a lot of trouble with before.

As the test subject, it is not your task to know which proof rule schema you should apply next. It is also not important that you know about all of them. However, you still need to know about the calculus to understand the result of the behaviour of the application. Therefore you, as the test subject, should use the following list from top to bottom to solve the sequent:

1. \forall_{goal}
2. \implies_{goal}
3. *hyp* (produces an error)
4. \implies_{hyp}
5. *hyp*
6. \exists_{goal}
7. $\hat{=}_{[:=]}$
8. *hyp*

E.1 Insights

This section contains notes taken by the examiner during the usability test. It also contains feedback from the test subject after the test was conducted, which influences the conclusion.

Observations:

- Test subject understands how and where to insert the provided sequent.
- Specify proof rule field is confusing.
- Test subject thinks he can already drag and drop a proof rule schema.
- Test subject tries clicking after failed drag and drop attempt.
- Test comes to a halt. Test subject doesn't know what to do with the specified proof rule.
- Test subject clicks around and tries different things to get on.
- The input field was not directly visible or understandable (could be because the test is done with paper mockups).
- After the specify proof rule is explained, the test subject knows how to proceed.
- Drag and drop of the specific proof rule was not intuitive.
- Drop space of the drag and drop functionality was intuitive and understood immediately.
- Error message was immediately visible and understood by the test subject.
- After the initial difficulties the test subject could solve the proof without any further help.

Discussion and feedback by the test subject:

- The test subject asked how he can write the special characters of the input sequent.
- The **Run** button was confusing. The test subject thought he has to run a program.
- It was not clear that he had to click on the proof rule schemas at first, but it would have been, if the test was conducted in the real browser application.
- The text **Specify proof rule** was very confusing. The test subject didn't know what to do at that step.
- The popup of the rule violation was comprehensible.
- The test subject asked how to start a new proof (their first guess would have been right) and he also asked what would happen with the current proof.
- The test subject asked if there is a history of old proofs.
- The test subject thinks the drop mechanic would be even more intuitive if the possible drop locations were highlighted.
- The test subject suggested that the cursor should change if it hovers over a proof rule that can be dragged.
- Proof rule schemas should not span multiple lines.
- The test subject thinks it is worse if the proof rule schemas wrap than if the current proof wraps to multiple lines.
- The title **Derivation** is not fitting.

E.2 Conclusion

This section contains all important recommendations and insights, that will be taken into consideration during the development of the frontend.

- Rename the **Run** button to **Proof**.
- Either add a tutorial, a good getting started guide or describe the different windows with a small text.
- Add a small popup when the user starts a new proof and tell him that his previous proof will be deleted (include an option to hide this popup the next time).
- Add a history of old proofs.
- Highlight the input fields in the specify proof rule box.
- Highlight the drop-off points to improve usability.
- Use the cursor to indicate the possible actions on the component.
- Change the title **Derivation** to something more fitting.

E.3 Paper mockups

Main View

Sequent Calculus Calculator		About
Derivation	<input type="text"/> Run	Proof rule schemas
		Specify proof rule

Error Page

Sequen Calculus Calculator

[About](#)

Derivation

[Run](#)

Proof rule schemas

Proof rule could not be applied ☒

Specify proof rule

Snippets for Usability Tests

Proof:

$$\frac{\text{hyp}}{A \vdash A} (\triangleq_{\text{ref}})$$

$$\frac{A \vdash [x := x] A}{\Rightarrow \text{goal}}$$

$$\frac{A \vdash \exists x. A \quad \frac{\text{hyp}}{A, B \vdash B}}{\Rightarrow \text{hyp}}$$

$$\frac{(\exists x. A) \Rightarrow B, A \vdash B}{\Rightarrow \text{goal}}$$

$$\frac{(\exists x. A) \Rightarrow B \vdash A \Rightarrow B}{\forall \text{goal} (x \text{ fin} \wedge (\exists x. A) \Rightarrow B)}$$

$$(\exists x. A) \Rightarrow B \vdash \forall x. (A \Rightarrow B)$$

Proof rule schemas

$$\frac{}{H, P \vdash P} \text{hyp}$$

$$\frac{H \vdash P}{H \vdash [E := E] P} (\triangleq [E := E])$$

$$\frac{H \vdash [x := E] P}{H \vdash \exists x. P} \exists_{\text{goal}}$$

$$\frac{H \vdash P \quad H, Q \vdash R}{H, P \Rightarrow Q \vdash R} \Rightarrow_{\text{hyp}}$$

$$\frac{H, P \Rightarrow Q}{H \vdash P \Rightarrow Q} \Rightarrow_{\text{goal}}$$

$$\frac{H \vdash P}{H \vdash \forall x. P} \forall_{\text{goal}} (x \hat{n}_{\text{fin}} H)$$

Proof rule instantiations

$$\frac{}{H, \boxed{P} \vdash \boxed{P}} \text{hyp}$$

$$\frac{H \vdash P}{H \vdash [\boxed{E} := \boxed{E}] \boxed{P}} (\triangleq [E := E])$$

$$\frac{H \vdash [x := \boxed{E}] P}{H \vdash \exists x. P} \exists_{\text{goal}}$$

$$\frac{H \vdash P \quad H, Q \vdash R}{H, \boxed{P} \Rightarrow \boxed{Q} \vdash R} \Rightarrow_{\text{hyp}}$$

$$\frac{H, P \Rightarrow Q}{H \vdash \boxed{P} \Rightarrow \boxed{Q}} \Rightarrow_{\text{goal}}$$

$$\frac{H \vdash P}{H \vdash \forall x. \boxed{P}} \forall_{\text{goal}} (x \hat{n}_{\text{fin}} H)$$

F Development Roadmap

In this section we describe how we would continue the development of The Sequent Calculus Calculator. This roadmap should guide follow-up projects.

Refactor import logic

Currently, the import feature does not verify whether the imported proof tree is proven or not. This circumvents the LCF-style and could lead to problems. This is described in more detail in section 12.3.2.

Because of the above-mentioned reasons we would focus on refactoring the import. In an ideal scenario, it would verify that the user-supplied proof tree is correct and proven up until that point. Only then would it return the proof tree data structure to the frontend.

Refactor term renders

The rendering logic is of high importance to this application. It is quite complicated and could hinder the development of new calculi modules. To prevent this we would refactor the term rendering accordingly. Some important points to keep in mind whilst doing so are:

- There is a single renderer for proof rule schemas, proof rules and proof trees.
 - Configuring the renderer happens through parameters.
 - In some cases the renderer has to use input fields to represent meta variables.
 - Some renderers use different parsers depending on the type of meta variable that is displayed.
- Writing a renderer should not be too complicated.

Improve parsing library

The current parsing library has one major shortcoming, namely the parsing of unary operations requires explicit parentheses. The term $\neg\neg P$ cannot be parsed without explicit parentheses as in $\neg(\neg P)$. This problem has to be solved at the library level. Submitting a pull request to `Punie/elm-parser-extras`²² would be able to change this behaviour.

Unification

This is a non-trivial task that has to be tackled in the future. Unification happens only for the implicit hypothesis H , which is a trivial case because all other instantiations are known at that time. Supporting a complete unification is not that simple because Elm, unlike Prolog, does not support this out of the box.

Once unification is implemented, the following features can be added to the calculator:

- Highlighting of applicable proof rules, i.e. the ones that unify with any of the pending subgoals.
- Support automatic proving by applying applicable proof rules at each step.
 - This feature should only be enabled in “Expert”-Mode and certainly not by default.

²²<https://github.com/Punie/elm-parser-extras>

Implement Lambda Calculus modules

This should not be too challenging, as all primitive operations are already implemented. Substitution, for example, is part of the current implementation and can be used to support β -reductions of the Lambda Calculus.

The implementations of the Simply Typed Lambda Calculus and the Polymorphically Typed Lambda Calculus can be based on the Lambda Calculus module.

Support exporting of proofs

In the future an export to a different format, such as \LaTeX , next to the current supported format JSON, would be useful. The exported \LaTeX could be used to create the PDF solution of the exercises.

A new kind of renderer is necessary to convert proof trees to \LaTeX . With the help of `elm/file`²³ it is possible to allow a user to download the generated \LaTeX directly. An import does the reverse of an export, i.e. it parses the \LaTeX code and reconstructs the proof tree. This is not trivial as \LaTeX is a complicated format. A proper subset of \LaTeX would first have to be defined as parsing any valid \LaTeX would probably be out of scope for this project.

Support syntactic rewrite rules

As described in section 17.2 syntactic rewrite rules are currently not supported. To support them, a generic way of detecting and replacing such structures would be required.

For example the term $P \vee Q$ would have to be detected in a proof tree. Clicking on this term would then apply the rewrite rule and display $\neg(\neg P \wedge \neg Q)$. This features requires several changes in the term renderer logic and could potentially make them even more complicated.

Support substitutions in the user input

To be able to solve all exercises presented in PrfM, the user has to be able to insert substitutions in the user input. This poses some new questions: How should they be represented as text? How can the parser handle them in case there are conflicting names? Will the application algorithm ignore the fact that $[x := E]P$ and $[y := E]P$ might represent the same term?

After answering those questions, this feature could probably be implemented fairly quickly as it does not require changing the kernel a lot.

²³<https://package.elm-lang.org/packages/elm/file/latest/>

G Redevelopment Roadmap

In this section we describe how we would develop the technical side of the application if we could start from scratch.

Reconsider programming language decision

First of all we would reconsider the language decision. Elm is a simple functional programming language that works well with single-page web application. Unfortunately, it has some shortcomings, which we discovered whilst developing The Sequent Calculus Calculator.

- Elm does not support type classes and makes it hard to write generic code. For example it's not possible to compare user-defined types. This restriction prevents the developer from creating *Dicts* that have a custom type as their key.
- Equality is based on the structure of records and types in Elm. Comparing nodes of a tree data structure, for example, is not as trivial as it would be in other programming languages. A possible solution for this problem has been described in section 6.
- Tools that support the development of Elm applications don't provide enough refactoring support. There are some tools which allow simple refactorings but they seem to be still in development [7, 14].
- A previous version of Elm supported a Time Traveling Debugger²⁴. Due to internal changes of the implementation, they decided to cease further development. Printing values to the console is currently the only feasible way to debug. This was quite challenging when developing the tree data structure that has to manage internal paths to support comparisons.

Renderer

Reuse and refactor the current rendering infrastructure if possible. This includes an important design decision, namely should there be one renderer for each occurrence of proof rules and proof rule schemas or not. We decided to write exactly one renderer that can be used in different views. With this we were able to maximize code reuse. Simultaneously we accepted the downside of some rather complex code.

Rethink the HOAS

The HOAS has a lot of advantages and makes the kernel extensible. On the other hand it makes the encoding of some terms awkward. For instance, the encoding of a universal quantification such as $\forall x.P$ cannot be represented in the lambda calculus syntax. Adding a new data structure per calculus is definitely not the way to go and we would strongly advise against that, as it would make the kernel less extensible.

Adding a quantification type constructor to the *Term* type would probably be a good compromise that does not pose too many restrictions on the kernel. This new type constructor would also solve the previously mentioned encoding problem.

Obviously there is no clear answer to the question “Should a HOAS be used or not?”. We think that the HOAS has many advantages. For instance, the export / import only works because we had this syntax available. It allows us to reuse a lot of code. In general, we would say that the HOAS is the way to go as we did with The Sequent Calculus Calculator.

²⁴<https://github.com/elm-lang/elm-reactor>

Consider new calculi early on

Adding support for *basicFoPCE* and *FoPCE* introduced a lot of new functionality which in turn forced us to rewrite and refactor a big chunk of our code. We think that this is normal as it is almost impossible to plan this far ahead. However, next time we would definitely plan the support of the *basicFoPCE* from the start. This requires some initial effort as it is difficult to see the bigger picture during the inception phase of a project.

H List of figures

1	Component overview	10
2	Interaction overview	10
3	Steps of a substitution	19
4	Entire webapplication	27
5	Rendered proof tree	28
6	Proof rule schema view	29
7	Instantiation view	30
8	Instantiation view with an error	31
9	Proof view	31

I List of tables

1	System tests	34
---	------------------------	----

J List of listings

1	Term data type	13
2	Sequent data type	13
3	Proof tree data type	14
4	Parsed term represented in the term data structure	15
5	Instantiation of a term	16
6	Preconditions of an application	17
7	Application of a proof rule	18
8	Simple α -conversion	20
9	Side condition type	21
10	Side conditions type alias	21
11	Side condition function type	22
12	Implementation of a non-freeness side condition	23
13	Import messages	24
14	Proof tree JSON structure	24
15	Variable JSON structure	25
16	Application JSON structure	25
17	Abstraction JSON structure	25
18	InParentheses JSON structure	25
19	Unary operation generator	32
20	Expression generator	33
21	Primary module data structure	35
22	ParseFunction type	35
23	BasicPC operators	36
24	Type of termToHtml	36
25	Special characters of BasicPC	37
26	Website build plan	38
27	CSS bug fix for Safari	41

K Acronyms

AST Abstract Syntax Tree. 13

CD Continuous Deployment. 6

CI Continuous Integration. 6

CLI Command Line Interface. 28

CSS Cascading Style Sheet. 8, 28, 30, 41

DAG Directed Acyclic Graph. 5

DOM Document Object Model. 11

EBNF Extended Backus-Naur form. 11, 15, 49

FFI Foreign Function Interface. 7

HOAS Higher-order abstract syntax. 9, 10, 24, 60

HTML Hypertext Markup Language. 8, 11, 27–29, 38, 40, 41

JS JavaScript. 7, 8, 10, 38

JSON JavaScript Object Notation. 11, 24–26, 39, 59

LCF Logic for Computable Functions. 6, 9, 13, 26, 39, 42, 58

MVC Model-View-Controller. 11

PrfM Programming Languages and Formal Methods. 6, 9, 19, 22, 23, 28, 34, 39, 40, 51, 59

UI User Interface. 6, 27, 30, 47, 51

L Bibliography

- [1] Wikipedia contributors. Left recursion, Removing left recursion. https://en.wikipedia.org/wiki/Left_recursion#Removing_left_recursion. Accessed: 20.03.2019.
- [2] Wikipedia contributors. Top-down parsing. https://en.wikipedia.org/wiki/Top-down_parsing. Accessed: 18.03.2019.
- [3] Dominik Kessler and Cyrill Hänni. Lambda Calculus Calculator. <https://lambdacalc.io>. Accessed: 31.03.2019.
- [4] Elm. Elm. <https://elm-lang.org/>. Accessed: 16.03.2019.
- [5] Elm. Parser. <https://github.com/elm/parser>. Accessed: 17.03.2019.
- [6] Elm. The Elm Architecture. <https://guide.elm-lang.org/architecture/>. Accessed: 24.03.2019.
- [7] ElmCast. elm-vim. <https://github.com/ElmCast/elm-vim>. Accessed: 04.06.2019.
- [8] Evan Czaplicki. Type system extensions. <https://github.com/elm/compiler/issues/1039>. Accessed: 10.06.2019.
- [9] Farhad Mehta. Introduction to Formal Proof and the Lambda Calculus, 2019.
- [10] Fay. Fay programming language. <https://github.com/faylang/fay>. Accessed: 16.03.2019.
- [11] ghcjs. Haskell to JavaScript compiler, based on GHC. <https://github.com/ghcjs/ghcjs>. Accessed: 16.03.2019.
- [12] GitLab. Gitlab pricing. <https://about.gitlab.com/pricing/>. Accessed: 06.06.2019.
- [13] Hugo Saracino. Parser Extra. <https://github.com/Punie/elm-parser-extras>. Accessed: 12.05.2019.
- [14] Keith Lazuka. intellij-elm. <https://klazuka.github.io/intellij-elm/>. Accessed: 04.06.2019.
- [15] Mozilla and individual contributors. Basic concepts of flexbox. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Basic_Concepts_of_Flexbox. Accessed: 09.06.2019.
- [16] Paulson, Lawrence. Strategic Principles in the Design of Isabelle. 08 1998.
- [17] PureScript. PureScript. <http://www.purescript.org/>. Accessed: 16.03.2019.
- [18] Tobias Nipkow and Lawrence Paulson and Makarius Wenzel and Gerwin Klein and Florian Haftmann and Tjark Weber and Johannes Hölzl. Isabelle. <https://isabelle.in.tum.de>. Accessed: 24.03.2019.
- [19] Wikipedia contributors. Monotonicity of entailment. https://en.wikipedia.org/wiki/Monotonicity_of_entailment. Accessed: 08.06.2019.