Friedrich-Alexander-Universität
Erlangen-Nürnberg

T.CS
Chair for Computer Science 8
Theoretical Computer Science

Implementing a Web-Based Proof Assistant for Fitch-Style Natural Deduction Proofs

Master Project in Computer Science

Leon Vatthauer

Advisor:

Thorsten Wißmann

Erlangen, April 1, 2026

Contents

1 Introduction **3**

2 Web Development in Haskell **3**

3 Implementing Finch **4**
3.1 Anatomy of the Website 5
3.2 Anatomy of a Fitch Proof 7
3.3 Specifying Logics in Finch 8
3.4 Verifying Fitch Proofs 9
3.5 Some Notes on Styling and Responsiveness 11
3.6 Testing 12

4 Conclusion and Future Work **12**

A Implementation Details **14**
A.1 Navigation 14
A.2 Sidebar Elements 14
A.3 Tooltips 14
A.4 Drag and Drop 15
A.5 Unicode Input 15
A.6 Sharing Proofs 16
A.7 Parsing Formulae 16

References **17**

1 Introduction

Finch [14] is a proof assistant for Fitch-style natural deduction proofs that has been developed during a Masters project at the Friedrich-Alexander-University of Erlangen-Nuremberg. The goal is to give students a tool which can check correctness of their natural deduction proofs and if incorrect give helpful error messages. Of course, such tools already exist and are available online [5] [10] [9], however none of them fulfill all of the following criteria:

1. Uses Fitch notation, where sub-proofs are denoted by vertical lines on the left, and a horizontal line to separate assumptions from other derivations inside the proof.
2. Gives the user helpful feedback if the proof contains errors.
3. Has an intuitive and accessible user experience.

Finch satisfies all of these criteria, with a special focus on usability. Intuitive operation is achieved in Finch by making proofs modifiable using simple drag and drop. The user can build their proof step-by-step by dragging single lines or new sub-proofs into the existing proof or by (re-)moving existing lines and sub-proofs to correct errors in the proof.

Additionally, this project is an exercise in modern Haskell [2] development. We use the Haskell library Miso [12], which can be considered a shallow embedded domain-specific language for web programming. It further makes use of the WebAssembly backend of the Glasgow Haskell Compiler [3], to produce static artifacts that can be easily hosted on a web server with no backend server required. Lastly, we use the testing framework Tasty [8] to ensure robustness of our codebase.

2 Web Development in Haskell

The goal of this section is to give a short introduction to the Miso library and the way that it is used in this project. For further information, especially concerning the internals of Miso, refer to its documentation [12].

Miso uses the *Model-View-Update* (MVU) pattern, which has been introduced by the Elm [11] programming language, for that reason it is also called *The Elm Architecture* (TEA). Concretely, every app built using Miso consists at its base of these three ingredients:

- The *model*, which is the state of the application;
- The *view function*, which turns the model into HTML;
- The *update function*, which updates the current model based on a given action (for example a browser event, like "button clicked").

Consider for example a very simple website that has one button, and displays a number that can be incremented by clicking the button. In this case the model would just be a number, and the only available action would be "Increment". The update function would also be rather simple, it would take the model (a number) and increment it. The code for such a small example application can be seen in Listing 1

The point of this example is to show how one can write HTML as a domain-specific language (DSL) embedded in Miso, using functions that represent the most common HTML constructors, like `H.div_`, `H.button_`, etc. and also HTML attributes like `HP.class_`.

Note also that in this example (in fact in every Miso application) the update function does not have type `Action -> Model -> Model`, but type `Action -> Effect parent Model Action`, where `Effect` is a special state monad used by Miso, that is abstracted over the `Model` and

```haskell
module Main where

import           Miso
import qualified Miso.Html as H
import qualified Miso.Html.Property as HP

-- | Export required for the WASM backend.
foreign export javascript "hs_start" main :: IO ()

-- Types
data Action = Increment deriving (Show, Eq)
type Model = Int

--  The update function
update :: Action -> Effect parent Model Action
update Increment = modify (+ 1)

-- The view function
view :: Model -> View Int Action
view n =
  H.div_
    [ HP.class "container" ]
    [ H.button_ [ H.onClick Increment ] [ text "+" ]
    , text $ ms n
    ]

--  Entry point of the application
main :: IO ()
main = startApp defaultEvents (component 0 update view)
```

Listing 1: An example Miso website with one button and a counter

`Action` types as well as a type variable `parent`, which is the type of the parent component, which is only relevant if there are more than one components in an app. This is not the case in Finch. Finally, all of this is glued together, by setting:

```haskell
main = startApp defaultEvents (component 0 updateModel viewModel),
```

which runs the Miso monad, with some default browser events enabled (e.g. `onClick`) and the component specified by our update and view functions together with the starting value `0`.

This is all that is needed to get started, for more complex applications one would just need to modify the view function to give the app more HTML structure and adjust the `Action` and `Model` types. Generally the `Action` type is an enumeration of all the Actions that the app supports (usually related to browser events) and the `Model` type is a record containing all the state data of the application. For complex records one can then use lenses, which is a Haskell design pattern for easily accessing and modifying record fields, that is also implemented in Miso, to circumvent the need for external lens libraries.

3 Implementing Finch

After the quick introduction to web development using Miso in Section 2, this section shall serve as an overview over the Finch web application, explaining roughly how the website is

structured, what state is used, how Fitch proofs are verified and how the application as a whole is tested to ensure correctness.

3.1 Anatomy of the Website

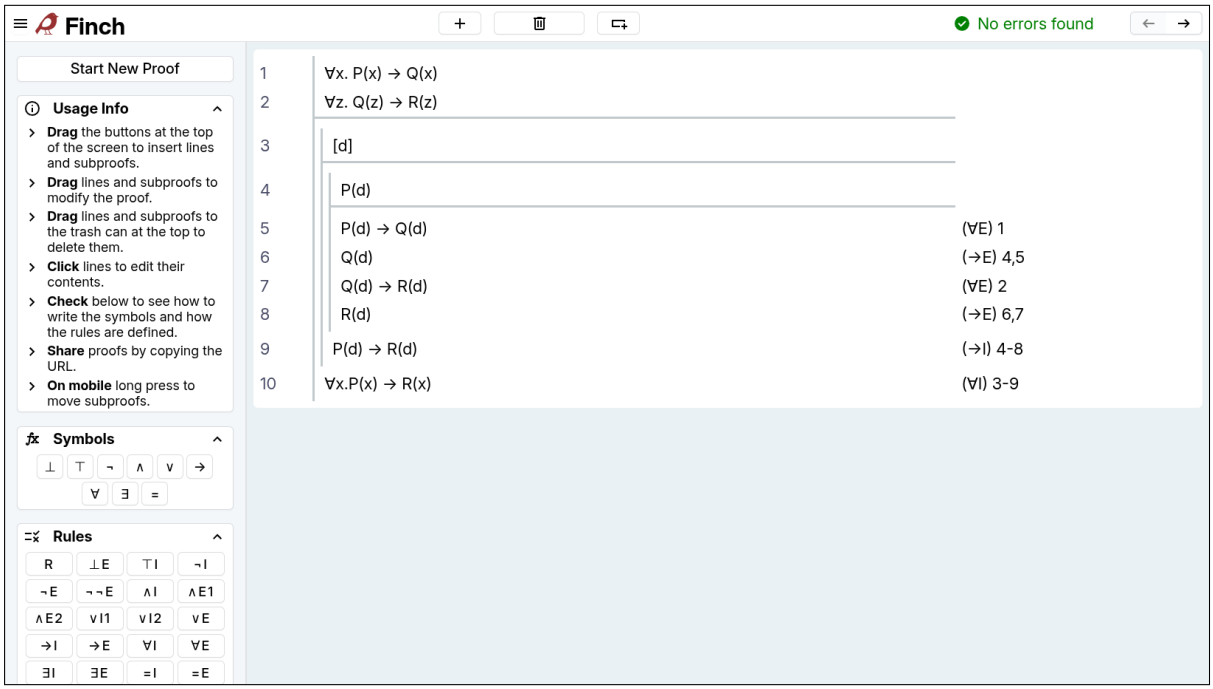

Figure 1: The Finch Proof Assistant on desktop devices.

The Finch website as depicted in Figure 1 can be dissected into three parts which we will now consider separately. Note however, that this is the view on desktop devices, on mobile devices the application looks slightly different to adjust for the smaller screen, this is covered in Section 3.5.

The header is split into three parts:

1. On the left, there is a button for opening/closing the collapsible sidebar, as well as the Finch logo and title;
2. In the center, there are buttons for modifying the proof (more on this in the explanation of the proof workspace);
3. On the right, there is a display for how many errors are left in the proof, as well as arrows for navigating between states of the proof.

There is no state inherent to the header, however the implementation of the navigation arrows is interesting. They utilize the HTML history API, which Finch uses by calling `history.pushState()` every time the proof is updated. Then, on button press `history.back()` or `history.forward()` is called to move along the history. See Appendix A.1 for more details on how this is implemented.

The sidebar is a scrollable and collapsible menu, which contains all supplementary information the user needs to operate the application. It consists of small elements that can be collapsed (see Appendix A.2 for details on how this is implemented), where most of the elements just

contain buttons that open a tooltip with more information on hover. Let us use the "Rules" container as an example, for every rule in the natural deduction calculus there is one button with that rules name. On hover a tooltip (see Appendix A.3 for details on the tooltips) is opened that shows the definition of the rule. In this case the rules definition is printed using MathJax [7], a JavaScript engine for displaying LaTeX in browsers. The LaTeX code for the rule definitions is automatically generated from each rules definition in Haskell.

The proof workspace contains our representation of Fitch proofs. There is one proof displayed at a time which can be edited by the user. Editing the proof is done using the HTML drag and drop API, where every line and sub-proof can be dragged into designated drop zones that are highlighted in Figure 2, where also the margins are visible that are needed to make proofs draggable (see Appendix A.4 for details on how drag and drop is implemented). Additionally, new lines and sub-proofs can be added by dragging the corresponding button in the header into a drop zone. Removing lines or sub-proofs works the other way around, there is a trash bin in the header, where elements can be dragged to for deletion. The contents of single lines in a proof can be edited by clicking on them, since they are simple `<input>` elements, we add additional text processing to these input fields, to enable convenient input for unicode characters (e.g. type `forall` for ∀) and for adding parentheses around selected text, see Appendix A.5. Finally, Finch proofs can easily be shared using the URL of the website, which encodes the current proof, see Appendix A.6 for implementation details.

Figure 2: A proof in Finch where the drop zones have been highlighted.

3.2 Anatomy of a Fitch Proof

Now that we have seen the structure of the website, let us dive deeper into how Fitch proofs are represented in Finch. Consider the following Fitch proof:

1	$\exists x.\forall y.P(x,y)$	
2	$\boxed{c}$	
3	$\forall y.P(c,y)$	
4	$\boxed{d}$	
5	$P(c,d)$	($\forall$E) 2
6	$\exists x.P(x,d)$	($\exists$I) 5
7	$\forall y.\exists x.P(x,y)$	($\forall$I) 3–5
8	$\forall y.\exists x.P(x,y)$	($\exists$E) 1, 2–7

On a general level, such a Fitch proof consists of the following data:

1. A list of assumptions,
2. a list of sub-proofs or derived formulae,
3. and the conclusion.

Thus, we have chosen the following data structure for representing proofs:

```
data Proof = SubProof [Assumption] [Either Derivation Proof] Derivation
```

Where a derivation is a formula together with a rule application (which itself consists of a rule name and a list of references, i.e. a reference to a single line or a reference to a proof specified as a line range) and an assumption is either a formula or a freshness condition like $\boxed{c}$. Lastly, we define formulae generically, i.e. without concrete operators, predicates, functions or quantifiers, instead there are abstract constructors like `Opr : Name -> [Formula] -> Formula`, which takes an operator (e.g. $\wedge$) and the list of formulae it is applied to. Terms are then defined in the standard way. This yields the following additional definitions:

```
1  data Term            = Var Name | Fun Name [Term]
2  data Formula         = Pred Name [Term]
3                       | InfixPred Name Term Term
4                       | Opr Name [Formula]
5                       | Quantifier Name Name Formula
6  data Assumption      = FreshVar Name | Assumption Formula
7  data RuleApplication = RuleApplication Name [Reference]
8  data Reference       = LineReference Int | ProofReference Int Int
```

Where for example `Quantifier` takes the quantifier name and the quantified variable. Additionally `InfixPred` is a special helper for defining infix predicates like '=' in first-order logic.

Note that this does not fully correspond to how these types are used in Finch, instead in Finch, we distinguish between `RawFormula`, which correspond to the type defined here, and `Formula`, which is a `RawFormula` together with parsing information. Refer to Appendix A.7 for more details on how this is implemented.

We have now seen roughly how proofs are stored in Finch, note that we do not need to store line numbers, since they can be derived when iterating over the proof, by just dragging a counter along that is incremented on every line.

Indeed, internally we rarely use line numbers to address specific lines, instead we introduce addresses, that mimic the structure of proofs. We distinguish between addresses for single lines and addresses for sub-proofs. They are defined as follows:

```
data NodeAddr  = NAAssumption Int
               | NAConclusion
               | NALine Int
               | NAProof Int NodeAddr
data ProofAddr = PANested Int ProofAddr
               | PAProof
```

Where every type of address specified what kind of node it addresses (assumption, conclusion, line or proof) and two special constructors **NAProof** and **PANested**, are used to descent into the another sub-proof. Note that some constructors are annotated with a number, recall that proofs are defined in terms of lists: **SubProof** [**Assumption**] [**Either Derivation Proof**] **Derivation**, the number is just the index in the corresponding list.

Using these addresses, we can now easily define functions, like a lookup function for retrieving a line at a given address:

```
naLookup :: NodeAddr -> Proof -> Maybe (Either Assumption Derivation)
naLookup (NAAssumption n) (SubProof fs _ _) = Left <$> fs !!? n
naLookup (NALine n) (SubProof _ ((!!? n) -> Just (Left d)) _) = Just . Right $ d
naLookup NAConclusion (SubProof _ _ c) = Just . Right $ c
naLookup (NAProof n na) (SubProof _ ((!!? n) -> Just (Right p)) _) = naLookup na p
naLookup _ _ = Nothing
```

where the Haskell extension `ViewPatterns` is used to make the definition even more concise.

3.3 Specifying Logics in Finch

As mentioned in the previous section, Finch is designed in a way to be (somewhat) agnostic of the logic that is used, i.e. we do not force a singular logic in the design of our data types. Instead, different logics can be specified using the given general data types. Recalling the data types we defined in Section 3.2, when defining a logic we need to define the following data for the syntax:

1. A list of operator names (together with their arity),
2. a list of infix predicate names,
3. a list of quantifiers names,
4. and one needs to decide whether terms are allowed (i.e. whether the logic is first-order).

Currently, there are propositional logic and first-order logic implemented, which differ in the obvious ways. Additionally one needs to implement the natural deduction rules of the logic, to then verify the proof as is explained in Section 3.4. Let us consider some exemplary natural deduction rules to get a feeling what such a rule might consist of:

$$(\wedge I)\ \frac{\phi \qquad \psi}{\phi \wedge \psi} \qquad (\forall E)\ \frac{\forall x.\phi}{\phi[E/x]} \qquad (\forall I)\ \frac{\begin{array}{|l} \boxed{c} \\ \vdots \\ \phi[c/x] \end{array}}{\forall x.\phi}$$

As one might expect, such a rule consists of a number of assumptions (either formulae or sub-proofs) and one conclusion. Note however, that these are not regular formulae and sub-proofs, the formulae may contain substitutions and placeholders 'ϕ', and the sub-proof is hollow, i.e. we are not interested in the proofs contents, just its outer structure (the assumptions and the conclusion). Thus the data structure for rule specifications we have chosen in Finch looks as follows:

```
data RuleSpec = RuleSpec [FormulaSpec] [ProofSpec] FormulaSpec
```

Where **FormulaSpec** is built like **Formula** with two additional constructors for placeholders and substitutions, similarly we add **TermSpec** and **AssumptionSpec**. Finally, we also add **ProofSpec**, which consists of just a tuple of the assumptions and the conclusion:

```
data TermSpec       = TVar Name | TFun Name [TermSpec] | TPlaceholder Name
data FormulaSpec    = FSubst Name (Subst Name)
                    | FPlaceholder Name
                    | FPred Name [TermSpec]
                    | FInfixPred Name Term Term
                    | FOpr Name [FormulaSpec]
                    | FQuantifier Name Name FormulaSpec
data AssumptionSpec = FFreshVar Name | AssumptionSpec FormulaSpec
type ProofSpec      = ([AssumptionSpec], FormulaSpec)
```

Note that **RuleSpec** does not contain the rules name, instead we store these specifications in a **Map Name RuleSpec**. For the three exemplary rules that we considered, the Haskell definitions look as follows:

```
exampleRules :: Map Name RuleSpec
exampleRules = fromList
  [ ("∧I", RuleSpec [phi, psi] [] (phi ∧ psi))
  , ("∀E", RuleSpec [FQuantifier "∀" "x" phi] [] (FSubst "φ" ("x" ~> "E")))
  , ("∀I", RuleSpec
              []
              [([FFreshVar "c"], FSubst "φ" ("x" ~> "c"))]
              (FQuantifier "∀" "x" phi))
  ]
  where
    phi = FPlaceholder "φ"
    psi = FPlaceholder "ψ"
```

3.4 Verifying Fitch Proofs

Now that we have investigated how Fitch proofs can be represented in a program, let us discuss an algorithm for verifying Fitch proofs. Note that natural deduction is a syntactical calculus, meaning our verification algorithm can also be purely syntactical.

There are multiple parts in a Fitch proof that need to be checked, on the outside we take the following steps:

(i) Check that all predicate and function symbols appear with only one arity (i.e. if somewhere in the proof $f(x)$ occurs, then an appearance of $f(x, y)$ somewhere else should throw an error).

(ii) Check freshness conditions, i.e. if we have an assumption $\boxed{c}$, then before that assumption, c should not be used and in the same proof that the subproof containing $\boxed{c}$ occurs, there

should be no occurrence of c (this is slightly stronger than regular definitions of freshness, but enables a simpler implementation and does not restrict expressiveness of the calculus).

(iii) Check validity of every rule application.

Where the first two points are easily implemented, the third however is more complicated. We implement the checking of rules generically, such that there is one rule checking algorithm, that takes a generic rule (the `RuleSpec` type that is defined in Section 3.3) and checks its validity. Consider for example that we want to check the rule application "`(∃E) 1, 2-4`" and whether it can derive some formula `f`, the algorithm proceeds as follows:

1. Check that `(∃E)` is a valid rule, i.e. check that there exists a corresponding `RuleSpec`, by indexing into the `Map Name RuleSpec`.
2. Check that the conclusion of the rule (in this case just the placeholder ψ) matches the formula f by unification on the placeholders (in this case resulting in a simple unifier that maps ψ to f.
3. Match the references, i.e. `1, 2-4` to the assumptions of the rule, this takes two different approaches, when checking a single line, i.e. `1`, we just unify the reference with the expected assumption. When checking a line range (i.e. a sub-proof) we match the sub-proofs assumptions to the expected assumptions (again by unification) and also unify the sub-proofs conclusion with the expected one. At the same time we also check that references are visible, i.e. not outside of the current scope, and that all referenced lines are fully parsed.

So far these steps just checked that the outer form of the rule is valid, i.e. that references match the expected formulae, next we need to check that all placeholders that occur more than once in a rule, instantiate to the same formula. For example the rule that takes a ϕ and derives $\phi \vee \psi$ needs to make sure that the ϕ in the assumption and in the conclusion are instantiated to the same formula. We do this by step-by-step collecting the instances for placeholders (again using unification) and then comparing them afterwards. The 'hard' part is handling a `RuleSpec` that contains substitutions, here we need some special care. The steps are as follows:

4. Collect all instances of term placeholders, i.e. for a `RuleSpec` like `E = D`, collect the actual terms for `E` and `D` in a map `Map Name [Term]`, where `Name` is for example `E` and `[Term]` is a list of all terms that `E` is instantiated to.
5. Next, we want to reduce this `Map` to a `Map Name Term`, by checking that every list of terms, contains only the same instances, i.e. by folding the list with equality. If this succeeds we now have a mapping from placeholder names to terms.

Using this map, we will now need to collect all the instances of formulae:

6. Collect all mappings from formula placeholders to formulae. When matching a formula to a specification, we need to consider three cases:

   a) "Simple" specifications, i.e. ones that do not contain substitutions

   $\Rightarrow$ collect assignments for placeholders and add these to the list of formulae that need to be equal at the end.

   b) Substitutions $\varphi[E/x]$, where φ is known

   $\Rightarrow$ unify the actual formula with the assignment for the placeholder φ, this (might) result in an assignment for E, which is then compared for equality to the previous assignment of E (if present) in the `Map Name Term`.

   c) Substitutions $\varphi[E/x]$, where E is known

   In this case special care is needed, such a substitution states, that the formula we

are looking for, was yielded by substituting E for x in some formula ϕ, however E might have already been in the formula beforehand. Thus, we cannot simply replace every occurrence of x with E, we have to do a backward substitution that returns a choice of formulae, where in each formulae a different number of occurrences of x have been replaced with E.

$\Rightarrow$ do backward substitution, yielding a list of possible formulae for ϕ.

Note that we do not consider the case for substitutions where both ϕ and E are unknown, such cases are trivial since there are no constraints, thus they should never occur and are a sign for an erroneous rule specification.

These steps result in a `Map Name [Either Formula [Formula]]`, where each placeholder maps to a list of either a single formula or a list of formulae that were obtained by backward substitution.

7. Finally, we need to reduce the aforementioned map, this is somewhat like the case for terms, where we just checked equality, we want to fold over the list and check equality of formulae. However, since the list may contain a list of formulae, for every such list, we need to choose one formula and check that the resulting list of formulae can be folded using equality, if not, we choose different elements in the "inner" formula lists, until we have tried every permutation or found an assignment that is valid.

All these steps may throw error messages and thus abort the checking process along the way, if the algorithm finishes without throwing an error, the proof is valid. If however an error is found, the offending rule application is annotated with a helpful error message using the means explained in A.7.

3.5 Some Notes on Styling and Responsiveness

We have now covered all important bits of the implementation of Finch, here are some final notes on how the application is styled, before investigating how Finch has been tested.

In Finch we use the Sass [13] CSS extension to style the application, which is a layer on top of CSS, which supports additional features like superclasses, variables and even functions and is compiled down to regular CSS. We split our styling into three modules, corresponding to the three parts of the website (header, sidebar and proof workspace) as well as one shared module which defines variables used throughout the styling like theme colors or margins. All these modules are imported into a singular main module that contains some global styling for the whole website.

Since CSS is not that complicated we won't cover too much here, besides some notes on responsiveness. In modern web development, every website should work and look good on mobile browsers. To guarantee that, we use media queries in Sass (also available in regular CSS) to render the website contents differently depending on the size of the screen. For example, on desktop the header contains the Finch logo as well the title text, whereas on mobile only the logo is displayed. Furthermore, on mobile the sidebar is by default collapsed, whereas on desktop it is by default shown. And, since the sidebar as it is on desktop screens, would be too narrow for mobile screens, it is enlarged and drawn over the rest of the application when open.

Besides media queries in CSS, these adjustments require something akin in Javascript/Miso, because the opening and collapsing of the sidebar is handled in the Miso state. We facilitate this, by listening to the `resize` browser event, which fires when the browser window is resized. On such a resize (and on page load) we use the `matchMedia()` function in Javascript, which takes a CSS media query and returns, whether the current document matches that query, this is stored in our state as a boolean flag `onMobile`. The concrete query used is:

```
window.matchMedia("screen and (max-width: 1200px)").matches
```

which returns true, on devices that have a screen (the other option being print, for printed documents) that is less than 1200px wide, so it should select mobile devices and small desktops/laptops.

3.6 Testing

Finally, a proof assistant (actually any software!) is no good without proper testing to ensure functional correctness. We use two major techniques for testing:

1. Property based testing, where random values for data types are generated using generator functions that have to be defined for each data type. Using these random values, abstractly specified properties are checked multiple times using different values.
2. Unit testing, where concrete values are given, together with expected results, the test then only consists of comparing the expected with the actual result for these values.

We use the Haskell testing library Tasty [8] to facilitate these tests and split it into two modules:

1. Testing the UI: Where we test that proof modifications work as expected, this is done by randomly generating proofs and then checking properties (property based testing) like "after removing a line, the proofs length is decremented by one" or "after moving a formula to a different line, the formula is actually at this line".
2. Testing the verification algorithm: Where we check that the proof verification algorithm as specified in Section 3.4 is correct, this is implemented using unit tests, where for every rule in first-order logic, we give example proofs, where the given rule is applied correctly (and check that the algorithm does not falsely detect an error) and examples, where the rule is applied incorrectly (and check that the algorithm detects that).

These tests serve to prevent bugs in the application, however as is the nature of these kinds of tests, they do not cover every possible case and thus cannot guarantee correctness of the application. They do however, serve as a good estimate.

4 Conclusion and Future Work

The result of this project is a tool that can be used in teaching Fitch-style natural deduction calculi to students. Furthermore, it was a study in practical modern Haskell and web development, using modern Haskell extensions and libraries as well as web development tools like Sass. Finally, it also resulted in open source contributions, since Finch is one of the few bigger web applications using the Miso library, resulting in multiple smaller contributions to the library, mostly consisting of bug reports, but also small improvements to the way DOM elements can be modified in Miso.

Future Work

The application is in a state where it can be used to teach the natural deduction calculi for first-order logic and propositional logic, future work could extend this to additionally cover some modal logic systems like **K**, **T**, **S4** and **S5**. Since the proof verification in Finch is written very generally, one would not have to do that much to get this functionality, the steps would be as follows:

1. Add 'strict' sub-proofs to Finch, which are roughly sub-proofs that force most rule applications to reference only the inner contents of the sub-proof (for more details, see [4]).

2. Adjust rule specifications to add a special kind of rule that can always refer to lines outside of strict sub-proofs (also explained in [4]).
3. Adjust the checker such that it verifies strictness of strict sub-proofs.

With these changes one would then only need to specify the natural deduction rule systems for these modal logics in Finch. Note also that no changes to the parser are required, since modalities can be treated as unary operators.

A Implementation Details

A.1 Navigation

When using the approach explained in Section 3.1, one issue appears: the browser history goes beyond just the current website, thus the user could leave the website by pressing the back-button. To alleviate this problem, the application needs to keep track of its own history, which is facilitated in the javascript module `Navigation.js` that serves as a drop-in replacement for history handling. It wraps the `history.pushState()` function, to annotate each state that is pushed with its index, thus the app knows that as soon as the index 0 has been pushed, the back button needs to be disabled. Furthermore, the biggest index is stored in the browsers sessionStorage, thus the forward button can be dynamically disabled, when the most recent history entry has been reached, and since the index is stored in sessionStorage, this information survives page reloads. Lastly, the module adds an event listener for the `popstate` event, which fires when the navigation functions are called, and then updates the indices (and button states) accordingly.

A.2 Sidebar Elements

To implement the collapsible sidebar elements, we use the HTML `<details>`-element (available across all major browsers since 2020), which has support for collapsing builtin, thus we do not need to add this to our application state. In our case it simply consists of a heading that is displayed when closed, and content that is unfolded when opened, together with some styling and CSS animations.

A.3 Tooltips

The tooltips that are used in the sidebar (as well as the header when hovering over buttons and the proof workspace for displaying errors) are implemented using the relatively new `popover` attribute in HTML (available since April 2024). HTML elements with this attribute are hidden until they are opened by calling `showPopover()` on them and then closed again when calling `hidePopover()` or clicking somewhere else in the page. When shown, these elements are drawn in a separate global scope, overlapping any other content on the website, thus they have one crucial benefit over normal elements with **`position`**: **`absolute`** styling: they can "leave" their parent container, overlapping content that is unrelated to themselves.

Ideally, we want these tooltips to be anchored to the element that triggers the tooltip, this is something that can easily be achieved using the very new CSS feature (available since early 2026) *anchor positioning*. Anchor positioning allows us to define one element as the anchor (in our case the button that triggers tooltip) and attach another element to this anchor. Together with a container of both that limits the scope of the anchor variable. In our tooltip example this looks as follows:

```
.tooltip-container {
  anchor-scope: --tooltip;
}
.tooltip-anchor {
  anchor: --tooltip;
}
```

```
.tooltip {
  position-anchor: --tooltip;
  left: anchor(right);
  align-self: anchor-center
}
```

Where for the tooltip, we specify **`left`**: **`anchor`**`(`**`right`**`)` to position the element to the right of the anchor.

A.4 Drag and Drop

We use the HTML drag and drop API, which functions by attaching fitting browser events to certain elements. Drag functionality is attached to an element by giving it the HTML attribute `draggable`. A drag operation throws the following events:

- `dragstart`: fired when the user starts dragging.
- `drag`: fired regularly while dragging.
- `dragend`: fired when the user lets go of the element.

We can also listen for events on other elements, that get dragged over, these are:

- `dragenter`: fired when a dragged element enters the bounds of another element.
- `dragover`: fired regularly while a dragged element is inside another element.
- `dragleave`: fired when a dragged element leaves the bounds of another element.
- `drop`: fired when a drag operation ends inside of another elements bounds (Note: `drop` only fires, if the elements dragover event is cancelled with `event.preventDefault()`).

One can furthermore transfer data in drag operations, this is however not used in finch and not easy to access in Miso. Thus we will stick to the case where no real data is transferred inside of a drag operation, instead the global state gets adjusted by the drag events. Let us consider one exemplary drag and drop operation in Finch:

1. The users starts dragging line 2 of the proof in Figure 2, thus `dragstart` is fired, which updates the global state with the address of the currently dragged line.
2. Next, the user drags this line over the first drop zone in the proof, i.e. before the formula $A \to B \wedge C$. Now `dragenter` fires on this drop zone, which adds a CSS class that increases the drop zones height, indicate that the line can be dropped here.
3. Finally, the user lets go of the dragged element, thus `drop` is fired on the drop zone, which retrieves the currently dragged line from the global state and moves it to the address of the drop zone (each drop zone is labelled with the address of the line that appears before it). Afterwards, `dragleave` is fired, which shrinks the drop zone once again and finally `dragend` fires, removing the reference to the currently dragged line from the state.

Note, that we do not listen to the `drag` or `dragover` events, since they are not needed for our functionality. Furthermore, as explained before, we cancel the `dragover` event on all drop zones to enable drop functionality.

A.5 Unicode Input

When specifying a logics syntax in Finch as is explained in Section 3.3, each operator can be annotated with an additional alias, enabling the user to write the alias instead of the unicode operator. For example the user can write `forall`, which automatically gets replaced by `∀`.

This is facilitated by listening to the `input` browser event, which is fired every time an input field is written to. When it fires, we replace every occurrence of an alias with the corresponding symbol. Since usually we have a longer alias for a simple unicode symbol, this would make the cursor jump to the end of the text field, to mitigate that, we store the `selectionStart` and `selectionEnd` of the input field (which are both just the cursor position if no text is selected) and update it according to how much shorter the symbol is than the alias.

A.6 Sharing Proofs

We use URL query fragments to store two kinds of information:

1. `?logic=` for specifying either `fol` or `prop`, which decides which underlying logic is used for the program, and
2. `?proof=` which contains the current proof encoded in base64 to enable easy sharing of proofs, just by sharing the current URL. Concretely, the proof gets printed in a format that can easily be reconstructed, and this printed string gets converted to base64 which can later be decoded again to reconstruct the proof.

The query parameters get updated every time the proof changes and are fetched on page load to initialize the proof and decide which logic is used, when no parameters are specified, default values are used (in our case first-order logic with an example proof).

A.7 Parsing Formulae

We use the Haskell library Megaparsec [6] for parsing formulae, rules and also full proofs (to reconstruct the URL parameters). There are good introductions to Megaparsec available online for example this function pearl on monadic parsing [1] (which is what Megaparsec implements), and implementing a parser for first-order logic (and similar logics) is a standard exercise. Thus, we will not cover in detail how parsing is implemented in Finch, instead let us consider how parsing information is used in the application:

As mentioned in Section 3.2, we annotate each formula, assumption and also rule application with parsing information, to this end we introduce the following new data type:

```
data Wrapper a = Unparsed Text Text
               | ParsedInvalid Text Text a
               | ParsedValid Text a
```

and define our elements of proofs using this wrapper type:

```
data Formula = Wrapper RawFormula
data Assumption = Wrapper RawAssumption
data Derivation = Derivation Formula (Wrapper RuleApplication)
```

where the types that are prefixed with `Raw` are defined as in Section 3.2. Let us take a deeper look at the wrapper type, its constructors are:

- **`Unparsed Text Text`**: the text could not be parsed. Consists of the input text that could not be parsed together with an error message (generated by Megaparsec).
- **`ParsedInvalid Text Text a`**: the text could be parsed, but contains semantic errors (errors detected in the verification process explained in Section 3.4). Consists of the input text together with an error message and the element that was successfully parsed.
- **`ParsedValid Text a`**: the text was successfully parsed and contains no semantic errors. Contains the input text and the successfully parsed element.

Note that for any wrapped data we always carry the user input, this is how we connect the value of an input field to the state.

References

[1] G. Hutton and E. Meijer, 'Monadic parsing in Haskell,' *Journal of Functional Programming*, vol. 8, no. 4, pp. 437–444, 1998. DOI: 10.1017/S0956796898003050

[2] S. Marlow et al., *Haskell 2010 language report*, 2010. [Online]. Available: https://www.haskell.org/definition/haskell2010.pdf

[3] S. Marlow and S. Peyton Jones, 'The glasgow haskell compiler,' in *The Architecture of Open Source Applications, Volume 2*, The Architecture of Open Source Applications, Volume 2. Lulu, Jan. 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler/

[4] P. Magnus, T. Button, J. Loftis, R. Trueman, A. Thomas-Bolduc, and R. Zach, *forall x: Calgary*. University of Calgary, 2020, ch. 43. [Online]. Available: https://forallx.openlogicproject.org/html/Ch43.html

[5] M. Rieppel, *Fitch Proof Editor*, 2023. [Online]. Available: https://github.com/mrieppel/fitchjs

[6] M. Karpov, *Megaparsec*, version 9.7.0, 2024. [Online]. Available: https://github.com/mrkkrp/megaparsec

[7] D. Cervone and V. Sorge, *MathJax*, version 4, 2025. [Online]. Available: https://www.mathjax.org/

[8] R. Cheplyaka and O. Charles, *Tasty*, 2025. [Online]. Available: https://github.com/UnkindPartition/tasty

[9] *Stanford Fitch*, 2025. [Online]. Available: http://intrologic.stanford.edu/logica/documentation/herbert.html

[10] R. Zach, K. Klement, and O. H. Ruiz, *fitch-checker*, 2025. [Online]. Available: https://github.com/OpenLogicProject/fitch-checker

[11] E. Czaplicki, *Elm*, 2026. [Online]. Available: https://elm-lang.org/

[12] D. M. Johnson, *Miso*, version 1.9, Mar. 2026. [Online]. Available: https://github.com/dmjio/miso

[13] The Sass Team, *Sass*, version 1.98.0, 2026. [Online]. Available: https://sass-lang.com/

[14] L. Vatthauer, *Finch*, version 1, Mar. 2026. [Online]. Available: https://github.com/Reijix/finch