Friedrich-Alexander-Universität
Technische Fakultät

FAU

Finch: A Browser-Based Proof Assistant for Fitch-Style Natural Deduction

Masterproject in Computer Science

Leon Vatthauer

April 21, 2026

1. Demo
2. Web Development in Haskell
3. Representing Fitch in Haskell
4. Verifying Natural Deduction Proofs
5. Conclusion and Future Work

1. Demo
2. **Web Development in Haskell**
3. Representing Fitch in Haskell
4. Verifying Natural Deduction Proofs
5. Conclusion and Future Work

Web Development in Haskell

The Miso Framework

Miso is a Haskell framework for front-end web development:

Web Development in Haskell

The Miso Framework

Miso is a Haskell framework for front-end web development:

- Embeds HTML in Haskell

Web Development in Haskell

The Miso Framework

Miso is a Haskell framework for front-end web development:

- Embeds HTML in Haskell
- Compiles to WebAssembly or JavaScript (depending on the GHC backend used)

Web Development in Haskell

The Miso Framework

Miso is a Haskell framework for front-end web development:

- Embeds HTML in Haskell
- Compiles to WebAssembly or JavaScript (depending on the GHC backend used)
- Uses the Elm architecture where every application consists of:

Web Development in Haskell

The Miso Framework

Miso is a Haskell framework for front-end web development:

- Embeds HTML in Haskell
- Compiles to WebAssembly or JavaScript (depending on the GHC backend used)
- Uses the Elm architecture where every application consists of:
  - A **model**, i.e. the state of the application;

Web Development in Haskell

The Miso Framework

Miso is a Haskell framework for front-end web development:

- Embeds HTML in Haskell
- Compiles to WebAssembly or JavaScript (depending on the GHC backend used)
- Uses the Elm architecture where every application consists of:
  - A **model**, i.e. the state of the application;
  - A **view function** which turns the state into HTML;

Web Development in Haskell

The Miso Framework

Miso is a Haskell framework for front-end web development:

- Embeds HTML in Haskell
- Compiles to WebAssembly or JavaScript (depending on the GHC backend used)
- Uses the Elm architecture where every application consists of:
  - A **model**, i.e. the state of the application;
  - A **view function** which turns the state into HTML;
  - An **update function** for updating the model based on actions.

Web Development in Haskell

Example: Counter Application

An application with an incrementable counter:

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

update :: Action -> Effect ROOT Model Action
update Increment = modify (+ 1)

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

Web Development in Haskell

Example: Drag and Drop

- Miso can listen to browser events and trigger the `update` function when events fire.

Web Development in Haskell

Example: Drag and Drop

- Miso can listen to browser events and trigger the `update` function when events fire.
- HTML offers native drag and drop functionality using lifecycle events for drag operations:

Web Development in Haskell

Example: Drag and Drop

- Miso can listen to browser events and trigger the `update` function when events fire.
- HTML offers native drag and drop functionality using lifecycle events for drag operations:
  - `dragstart`

Web Development in Haskell

Example: Drag and Drop

- Miso can listen to browser events and trigger the `update` function when events fire.
- HTML offers native drag and drop functionality using lifecycle events for drag operations:
  - `dragstart`
  - `drag`

Web Development in Haskell

Example: Drag and Drop

- Miso can listen to browser events and trigger the `update` function when events fire.
- HTML offers native drag and drop functionality using lifecycle events for drag operations:
  - `dragstart`
  - `drag`
  - `dragend`

Web Development in Haskell

Example: Drag and Drop

- Miso can listen to browser events and trigger the `update` function when events fire.
- HTML offers native drag and drop functionality using lifecycle events for drag operations:
  - `dragstart`
  - `drag`
  - `dragend`
- and events for when an element gets dragged over:

Web Development in Haskell

Example: Drag and Drop

- Miso can listen to browser events and trigger the `update` function when events fire.
- HTML offers native drag and drop functionality using lifecycle events for drag operations:
  - `dragstart`
  - `drag`
  - `dragend`
- and events for when an element gets dragged over:
  - `dragenter`

Web Development in Haskell

Example: Drag and Drop

- Miso can listen to browser events and trigger the `update` function when events fire.
- HTML offers native drag and drop functionality using lifecycle events for drag operations:
  - `dragstart`
  - `drag`
  - `dragend`
- and events for when an element gets dragged over:
  - `dragenter`
  - `dragover`

Web Development in Haskell

Example: Drag and Drop

- Miso can listen to browser events and trigger the `update` function when events fire.
- HTML offers native drag and drop functionality using lifecycle events for drag operations:
  - `dragstart`
  - `drag`
  - `dragend`
- and events for when an element gets dragged over:
  - `dragenter`
  - `dragover`
  - `dragleave`

Web Development in Haskell

Example: Drag and Drop

- Miso can listen to browser events and trigger the `update` function when events fire.
- HTML offers native drag and drop functionality using lifecycle events for drag operations:
  - `dragstart`
  - `drag`
  - `dragend`
- and events for when an element gets dragged over:
  - `dragenter`
  - `dragover`
  - `dragleave`
  - `drop`

1. Demo
2. Web Development in Haskell
3. **Representing Fitch in Haskell**
4. Verifying Natural Deduction Proofs
5. Conclusion and Future Work

Representing Fitch in Haskell

Anatomy of a Fitch Proof

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

Representing Fitch in Haskell

Anatomy of a Fitch Proof

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

```
data Term = Var Name | Fun Name [Term]
```

Representing Fitch in Haskell

Anatomy of a 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

```haskell
data Term = Var Name | Fun Name [Term]
data Formula = Pred Name [Term]
  | Opr Name [Formula]
  | Quantifier Name Name Formula
```

Representing Fitch in Haskell

Anatomy of a Fitch Proof

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

```haskell
data Term = Var Name | Fun Name [Term]
data Formula = Pred Name [Term]
  | Opr Name [Formula]
  | Quantifier Name Name Formula
data Assumption = FreshVar Name
                | Assumption Formula
```

Representing Fitch in Haskell

Anatomy of a Fitch Proof

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

```haskell
data Term = Var Name | Fun Name [Term]
data Formula = Pred Name [Term]
  | Opr Name [Formula]
  | Quantifier Name Name Formula
data Assumption = FreshVar Name
                | Assumption Formula
data Reference
  = LineReference Int
  | ProofReference Int Int
```

Representing Fitch in Haskell

Anatomy of a 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

```haskell
data Term = Var Name | Fun Name [Term]
data Formula = Pred Name [Term]
  | Opr Name [Formula]
  | Quantifier Name Name Formula
data Assumption = FreshVar Name
                | Assumption Formula
data Reference
  = LineReference Int
  | ProofReference Int Int
data RuleApplication
  = RuleApplication Name [Reference]
```

Representing Fitch in Haskell

Anatomy of a Fitch Proof

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

```
data Term = Var Name | Fun Name [Term]
data Formula = Pred Name [Term]
  | Opr Name [Formula]
  | Quantifier Name Name Formula
data Assumption = FreshVar Name
                | Assumption Formula
data Reference
  = LineReference Int
  | ProofReference Int Int
data RuleApplication
  = RuleApplication Name [Reference]
data Derivation
  = Derivation Formula RuleApplication
```

Representing Fitch in Haskell

Anatomy of a 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

```haskell
data Term = Var Name | Fun Name [Term]
data Formula = Pred Name [Term]
  | Opr Name [Formula]
  | Quantifier Name Name Formula
data Assumption = FreshVar Name
                | Assumption Formula
data Reference
  = LineReference Int
  | ProofReference Int Int
data RuleApplication
  = RuleApplication Name [Reference]
data Derivation
  = Derivation Formula RuleApplication
data Proof
  = Proof [Assumption]
    [Either Derivation Proof] Derivation
```

Representing Fitch in Haskell

Anatomy of a 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

```
data Term = Var Name | Fun Name [Term]
data Formula = Pred Name [Term]
  | Opr Name [Formula]
  | Quantifier Name Name Formula
data Assumption = FreshVar Name
                | Assumption Formula
data Reference
  = LineReference Int
  | ProofReference Int Int
data RuleApplication
  = RuleApplication Name [Reference]
data Derivation
  = Derivation Formula RuleApplication
data Proof
  = Proof [Assumption]
    [Either Derivation Proof] Derivation
```

Representing Fitch in Haskell

Modifying Proofs

- **Problem:** indexing this data structure is inconvenient, consider defining a function
  `lookup :: Int -> Proof -> Maybe (Either Assumption Derivation)`

Representing Fitch in Haskell

Modifying Proofs

- **Problem:** indexing this data structure is inconvenient, consider defining a function
  `lookup :: Int -> Proof -> Maybe (Either Assumption Derivation)`
  ↪ Requires recursion on `Proof` while keeping track of the index.

Representing Fitch in Haskell

Modifying Proofs

- **Problem:** indexing this data structure is inconvenient, consider defining a function
  `lookup :: Int -> Proof -> Maybe (Either Assumption Derivation)`
  ↪ Requires recursion on `Proof` while keeping track of the index.
  ↪ Becomes even more complicated in functions for inserting and (re-)moving lines.

Representing Fitch in Haskell

Modifying Proofs

- **Problem:** indexing this data structure is inconvenient, consider defining a function
  `lookup :: Int -> Proof -> Maybe (Either Assumption Derivation)`
  ↪ Requires recursion on `Proof` while keeping track of the index.
  ↪ Becomes even more complicated in functions for inserting and (re-)moving lines.
- **Instead:** use addresses that resemble the recursive structure of `Proof`:

Representing Fitch in Haskell

Modifying Proofs

- **Problem:** indexing this data structure is inconvenient, consider defining a function
  `lookup :: Int -> Proof -> Maybe (Either Assumption Derivation)`
  ↪ Requires recursion on `Proof` while keeping track of the index.
  ↪ Becomes even more complicated in functions for inserting and (re-)moving lines.
- **Instead:** use addresses that resemble the recursive structure of `Proof`:

```
data NodeAddr
  = NAAssumption Int
  | NAConclusion
  | NALine Int
  | NAProof Int NodeAddr
```

Representing Fitch in Haskell

Modifying Proofs

- **Problem:** indexing this data structure is inconvenient, consider defining a function
  `lookup :: Int -> Proof -> Maybe (Either Assumption Derivation)`
  ↪ Requires recursion on `Proof` while keeping track of the index.
  ↪ Becomes even more complicated in functions for inserting and (re-)moving lines.
- **Instead:** use addresses that resemble the recursive structure of `Proof`:

```haskell
data NodeAddr
  = NAAssumption Int
  | NAConclusion
  | NALine Int
  | NAProof Int NodeAddr
```

```haskell
data ProofAddr
  = PAProof Int
  | PANested Int ProofAddr
```

Representing Fitch in Haskell

Modifying Proofs (Cont.)

- Using `NodeAddr` and Haskell's viewpatterns, we can elegantly define:

```
lookup :: NodeAddr -> Proof -> Maybe (Either Assumption Derivation)
lookup (NAAss n)    (Proof (maybeAt n -> Just a) ps c)          = Just (Left a)
lookup  NACon       (Proof as ps c)                             = Just (Right c)
lookup (NALn n)     (Proof as (maybeAt n -> Just (Left d)) c)   = Just (Right d)
lookup (NAPrf n na) (Proof as (maybeAt n -> Just (Right p)) c) = lookup na p
lookup  _           _                                           = Nothing
```

Representing Fitch in Haskell

Modifying Proofs (Cont.)

- Using `NodeAddr` and Haskell's viewpatterns, we can elegantly define:

```
lookup :: NodeAddr -> Proof -> Maybe (Either Assumption Derivation)
lookup (NAAss n)    (Proof (maybeAt n -> Just a) ps c)          = Just (Left a)
lookup  NACon       (Proof as ps c)                             = Just (Right c)
lookup (NALn n)     (Proof as (maybeAt n -> Just (Left d)) c)   = Just (Right d)
lookup (NAPrf n na) (Proof as (maybeAt n -> Just (Right p)) c) = lookup na p
lookup  _           _                                           = Nothing
```

- **Remaining Problem:** sometimes `Int` indices are still needed (rules use `Int` references!)

Representing Fitch in Haskell

Modifying Proofs (Cont.)

- Using `NodeAddr` and Haskell's viewpatterns, we can elegantly define:

```
lookup :: NodeAddr -> Proof -> Maybe (Either Assumption Derivation)
lookup (NAAss n)    (Proof (maybeAt n -> Just a) ps c)          = Just (Left a)
lookup  NACon       (Proof as ps c)                             = Just (Right c)
lookup (NALn n)     (Proof as (maybeAt n -> Just (Left d)) c)   = Just (Right d)
lookup (NAPrf n na) (Proof as (maybeAt n -> Just (Right p)) c) = lookup na p
lookup  _           _                                           = Nothing
```

- **Remaining Problem:** sometimes `Int` indices are still needed (rules use `Int` references!)
- Define conversion functions:

```
naToInt :: NodeAddr -> Proof -> Maybe Int
intToNA :: Int      -> Proof -> Maybe NodeAddr
```

Representing Fitch in Haskell

Modifying Proofs (Cont.)

- Using `NodeAddr` and Haskell's viewpatterns, we can elegantly define:

```
lookup :: NodeAddr -> Proof -> Maybe (Either Assumption Derivation)
lookup (NAAss n)    (Proof (maybeAt n -> Just a) ps c)          = Just (Left a)
lookup  NACon       (Proof as ps c)                             = Just (Right c)
lookup (NALn n)     (Proof as (maybeAt n -> Just (Left d)) c)   = Just (Right d)
lookup (NAPrf n na) (Proof as (maybeAt n -> Just (Right p)) c) = lookup na p
lookup  _           _                                           = Nothing
```

- **Remaining Problem:** sometimes `Int` indices are still needed (rules use `Int` references!)
- Define conversion functions:

```
naToInt :: NodeAddr -> Proof -> Maybe Int
intToNA :: Int      -> Proof -> Maybe NodeAddr
```

- Definition is still hard (and error prone) but only needed once.

Representing Fitch in Haskell

Notes on Testing

- Property based testing is used to check properties like:

```haskell
prop_naToIntInverse1 :: Proof -> Property
prop_naToIntInverse1 p
  = forAll (chooseInt (1, pLength p)) $ \n ->
    ((`naToInt` p) <$> intToNA n p) === Just (Just n)
prop_naToIntInverse2 :: Proof -> Property
prop_naToIntInverse2 p
  = forAll (arbitraryNodeAddrFor p) $ \a ->
    ((`intToNA` p) <$> naToInt a p) === Just (Just a)
```

Representing Fitch in Haskell

Notes on Testing

- Property based testing is used to check properties like:

```haskell
prop_naToIntInverse1 :: Proof -> Property
prop_naToIntInverse1 p
  = forAll (chooseInt (1, pLength p)) $ \n ->
    ((`naToInt` p) <$> intToNA n p) === Just (Just n)
prop_naToIntInverse2 :: Proof -> Property
prop_naToIntInverse2 p
  = forAll (arbitraryNodeAddrFor p) $ \a ->
    ((`intToNA` p) <$> naToInt a p) === Just (Just a)
```

- Using generators like:

```haskell
arbitraryNodeAddrFor :: Proof -> Gen NodeAddr
arbitraryProof :: Gen Proof
```

Representing Fitch in Haskell

Rules

$$(\forall E)\ \frac{\forall x.\phi}{\phi[E/x]}$$

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

$$(=E)\ \frac{\phi[E/x] \qquad E = D}{\phi[D/x]}$$

Representing Fitch in Haskell

Rules

$$(\forall E)\ \frac{\forall x.\phi}{\phi[E/x]}$$

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

$$(=E)\ \frac{\phi[E/x] \qquad E = D}{\phi[D/x]}$$

```
data TermSpec = TVar Name
              | TFun Name [TermSpec]
              | TPlaceholder Name
```

Representing Fitch in Haskell

Rules

$$(\forall E)\ \frac{\forall x.\phi}{\phi[E/x]}$$

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

$$(=E)\ \frac{\phi[E/x] \qquad E = D}{\phi[D/x]}$$

```
data TermSpec = TVar Name
              | TFun Name [TermSpec]
              | TPlaceholder Name
data FormulaSpec = FSubst Name (Subst Name)
                 | FPlaceholder Name
                 | FPred Name [TermSpec]
                 | FOpr Name [FormulaSpec]
                 | FQuantifier Name Name FormulaSpec
```

Representing Fitch in Haskell

Rules

$$(\forall E)\ \frac{\forall x.\phi}{\phi[E/x]}$$

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

$$(=E)\ \frac{\phi[E/x] \qquad E = D}{\phi[D/x]}$$

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

Representing Fitch in Haskell

Rules

$$(\forall E)\ \frac{\forall x.\phi}{\phi[E/x]}$$

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

$$(=E)\ \frac{\phi[E/x] \qquad E = D}{\phi[D/x]}$$

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

Representing Fitch in Haskell

Rules

$$(\forall E)\ \frac{\forall x.\phi}{\phi[E/x]}$$

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

$$(=E)\ \frac{\phi[E/x] \qquad E = D}{\phi[D/x]}$$

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

Representing Fitch in Haskell

Rules

$$(\forall E)\ \frac{\forall x.\phi}{\phi[E/x]}$$

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

$$(=E)\ \frac{\phi[E/x] \qquad E = D}{\phi[D/x]}$$

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

1. Demo
2. Web Development in Haskell
3. Representing Fitch in Haskell
4. **Verifying Natural Deduction Proofs**
5. Conclusion and Future Work

Verifying Natural Deduction Proofs

Verifying Proofs

Proofs are verified in three steps:

Verifying Natural Deduction Proofs

Verifying Proofs

Proofs are verified in three steps:

1. Check for syntax errors and wrong arities.

Verifying Natural Deduction Proofs

Verifying Proofs

Proofs are verified in three steps:

1. Check for syntax errors and wrong arities.
2. Verify freshness conditions.

Verifying Natural Deduction Proofs

Verifying Proofs

Proofs are verified in three steps:

1. Check for syntax errors and wrong arities.
2. Verify freshness conditions.
3. Verify all rule applications.

Verifying Natural Deduction Proofs

Verifying Rule Applications

1. Check that the rule exists.

Verifying Natural Deduction Proofs

Verifying Rule Applications

1. Check that the rule exists.
2. Check that the conclusion of the rule matches the formula it is applied to by unification on the placeholders.

Verifying Natural Deduction Proofs

Verifying Rule Applications

1. Check that the rule exists.
2. Check that the conclusion of the rule matches the formula it is applied to by unification on the placeholders.
3. Match the references to the assumptions of the rule in the same way.

Verifying Natural Deduction Proofs

Verifying Rule Applications

1. Check that the rule exists.
2. Check that the conclusion of the rule matches the formula it is applied to by unification on the placeholders.
3. Match the references to the assumptions of the rule in the same way.

Verifying Natural Deduction Proofs

Verifying Rule Applications

1. Check that the rule exists.
2. Check that the conclusion of the rule matches the formula it is applied to by unification on the placeholders.
3. Match the references to the assumptions of the rule in the same way.

These steps yield a list:

```
[Either (Assumption, AssumptionSpec) (Formula, FormulaSpec)]
```

Verifying Natural Deduction Proofs

Verifying Rule Applications (Cont.)

4. Collect all instances of term placeholders, E, D.

Verifying Natural Deduction Proofs

Verifying Rule Applications (Cont.)

4. Collect all instances of term placeholders, E, D.
5. Check that placeholders are only instantiated to equal terms.

Verifying Natural Deduction Proofs

Verifying Rule Applications (Cont.)

4. Collect all instances of term placeholders, $\mathtt{E}$, $\mathtt{D}$.
5. Check that placeholders are only instantiated to equal terms.
6. Collect all mappings from placeholders to formulae. Consider three cases:

Verifying Natural Deduction Proofs

Verifying Rule Applications (Cont.)

4. Collect all instances of term placeholders, E, D.
5. Check that placeholders are only instantiated to equal terms.
6. Collect all mappings from placeholders to formulae. Consider three cases:
   6.1 “Simple” specifications, i.e. ones that do not contain substitutions
   $\Rightarrow$ collect assignments for placeholders ϕ, ψ.

Verifying Natural Deduction Proofs

Verifying Rule Applications (Cont.)

4. Collect all instances of term placeholders, E, D.
5. Check that placeholders are only instantiated to equal terms.
6. Collect all mappings from placeholders to formulae. Consider three cases:
   - 6.1 "Simple" specifications, i.e. ones that do not contain substitutions
     $\Rightarrow$ collect assignments for placeholders ϕ, ψ.
   - 6.2 Substitutions $\varphi[E/x]$, where φ is known
     $\Rightarrow$ unify the actual formula with the assignment for the placeholder φ and compare the result with the previous mapping for E.

Verifying Natural Deduction Proofs

Verifying Rule Applications (Cont.)

4. Collect all instances of term placeholders, E, D.
5. Check that placeholders are only instantiated to equal terms.
6. Collect all mappings from placeholders to formulae. Consider three cases:
   - 6.1 "Simple" specifications, i.e. ones that do not contain substitutions
     $\Rightarrow$ collect assignments for placeholders ϕ, ψ.
   - 6.2 Substitutions $\varphi[E/x]$, where φ is known
     $\Rightarrow$ unify the actual formula with the assignment for the placeholder φ and compare the result with the previous mapping for E.
   - 6.3 Substitutions $\varphi[E/x]$, where E is known
     $\Rightarrow$ do backward substitution, yielding a list of possible formulae for ϕ.

Verifying Natural Deduction Proofs

Verifying Rule Applications (Cont.)

4. Collect all instances of term placeholders, E, D.
5. Check that placeholders are only instantiated to equal terms.
6. Collect all mappings from placeholders to formulae. Consider three cases:
   - 6.1 "Simple" specifications, i.e. ones that do not contain substitutions
     $\Rightarrow$ collect assignments for placeholders ϕ, ψ.
   - 6.2 Substitutions $\varphi[E/x]$, where φ is known
     $\Rightarrow$ unify the actual formula with the assignment for the placeholder φ and compare the result with the previous mapping for E.
   - 6.3 Substitutions $\varphi[E/x]$, where E is known
     $\Rightarrow$ do backward substitution, yielding a list of possible formulae for ϕ.
7. Reduce the resulting mappings.

1. Demo
2. Web Development in Haskell
3. Representing Fitch in Haskell
4. Verifying Natural Deduction Proofs
5. **Conclusion and Future Work**

Conclusion and Future Work

We have seen:

Conclusion and Future Work

We have seen:

- A demo of the Finch tool

Conclusion and Future Work

We have seen:

- A demo of the Finch tool
- A quick introduction to modern web development in Haskell

Conclusion and Future Work

We have seen:

- A demo of the Finch tool
- A quick introduction to modern web development in Haskell
- A (generic) verification algorithm for Fitch proofs

Conclusion and Future Work

We have seen:

- A demo of the Finch tool
- A quick introduction to modern web development in Haskell
- A (generic) verification algorithm for Fitch proofs

Next steps:

Conclusion and Future Work

We have seen:

- A demo of the Finch tool
- A quick introduction to modern web development in Haskell
- A (generic) verification algorithm for Fitch proofs

Next steps:

- Use Finch in teaching (GLoIn?)

Conclusion and Future Work

We have seen:

- A demo of the Finch tool
- A quick introduction to modern web development in Haskell
- A (generic) verification algorithm for Fitch proofs

Next steps:

- Use Finch in teaching (GLoIn?)
- Extend Finch to modal logic


Questions?