Schematic Polymorphism in the Abella Proof Assistant

Gopalan Nadathur $^1$ Yuting Wang $^2$

$^1$University of Minnesota, Twin Cities
$^2$Yale University

PPDP, Frankfurt am main, September 2018
An interactive theorem proving system with the following characteristics

- Based on a (first-order) logic over lambda terms that incorporates (least and greatest) fixed point definitions
- Embeds an executable (first-order) specification logic also over lambda terms
- Supports higher-order abstract syntax
An interactive theorem proving system with the following characteristics

- Based on a (first-order) logic over lambda terms that incorporates (least and greatest) fixed point definitions
- Embeds an executable (first-order) specification logic also over lambda terms
- Supports higher-order abstract syntax

Abella provides a vehicle for implementing and verifying rule-based systems exploiting higher-order abstract syntax
An interactive theorem proving system with the following characteristics

- Based on a (first-order) logic over lambda terms that incorporates (least and greatest) fixed point definitions
- Embeds an executable (first-order) specification logic also over lambda terms
- Supports higher-order abstract syntax

Abella provides a vehicle for implementing and verifying rule-based systems exploiting higher-order abstract syntax

**One limitation**: both the reasoning logic and the specification logic are simply typed
The Problems with Monomorphic Typing

In implementation and reasoning tasks, we often need to treat library data structures and operations at different types.

Our goal: to make them more concise and modular without changing the theoretical underpinnings of Abella.
In implementation and reasoning tasks, we often need to treat library data structures and operations at different types.

For example, in verified compilation we may need to

- specify lists and operations on them for managing bound variables in different intermediate languages
- prove properties concerning these data structures
The Problems with Monomorphic Typing

In implementation and reasoning tasks, we often need to treat library data structures and operations at different types. For example, in verified compilation we may need to:

- specify lists and operations on them for managing bound variables in different intermediate languages
- prove properties concerning these data structures

With monomorphic typing, such developments have to be repeated several times.
The Problems with Monomorphic Typing

In implementation and reasoning tasks, we often need to treat library data structures and operations at different types.

For example, in verified compilation we may need to:

- specify lists and operations on them for managing bound variables in different intermediate languages;
- prove properties concerning these data structures.

With monomorphic typing, such developments have to be repeated several times.

**Our goal:** to make them more concise and modular without changing the theoretical underpinnings of Abella.
Predicate constants are treated as defined symbols whose meanings are given by clauses of the form

$$\forall x_1 : \alpha_1, \ldots, x_n : \alpha_n. \ p \ t_1 \ \cdots \ t_n \triangleq B$$
Predicate constants are treated as defined symbols whose meanings are given by clauses of the form

$$\forall x_1 : \alpha_1, \ldots, x_n : \alpha_n. \; p \; t_1 \; \ldots \; t_n \triangleq B$$

The logic is parameterized by a definition, which is a collection of such clauses introduced in *definition blocks*.

**Example:**

$$\forall l : list. \; \text{app} \; \text{nil} \; l \; l \triangleq \top$$

$$\forall x : \iota, l : list, l_2 : list, l_3 : list. \;
\text{app} \; (x :: l_1) \; l_2 \; (x :: l_3) \triangleq \text{app} \; l_1 \; l_2 \; l_3$$
The Treatment of Fixed-Point Definitions

Predicate constants are treated as defined symbols whose meanings are given by clauses of the form

$$\forall x_1 : \alpha_1, \ldots, x_n : \alpha_n. \ p \ t_1 \ldots \ t_n \triangleq B$$

The logic is parameterized by a definition, which is a collection of such clauses introduced in definition blocks

Example:

$$\forall \ell : \text{list}. \ \text{app} \ \text{nil} \ \ell \ \ell \triangleq \top$$

$$\forall x : \iota, \ell : \text{list}, \ell_2 : \text{list}, \ell_3 : \text{list}. \ \text{app} (x :: \ell_1) \ \ell_2 (x :: \ell_3) \triangleq \text{app} \ \ell_1 \ \ell_2 \ \ell_3$$

Definitions are given a fixed-point interpretation via rules for introducing atomic formulas in a sequent-style presentation

- The right introduction rule realizes the idea of backchaining
- The left introduction rule codifies case analysis, which builds in equality based on term structure
Let $S$ be the sequent $\Sigma : \Gamma, A \rightarrow F$, where $\Sigma$ represents the eigenvariable context.

A point to note: this rule is sensitive to type information.
The Left Introduction Rule for Definitions

Let $S$ be the sequent $\Sigma : \Gamma, A \rightarrow F$, where $\Sigma$ represents the eigenvariable context

Let $CSU(A, A')$ represents a complete set of unifiers for (the atomic formulas or terms) $A$ and $A'$
The Left Introduction Rule for Definitions

Let $\Sigma$ be the sequent $\Sigma : \Gamma, A \rightarrow F$, where $\Sigma$ represents the eigenvariable context.

Let $\text{CSU}(A, A')$ represents a complete set of unifiers for (the atomic formulas or terms) $A$ and $A'$

For definition $\mathcal{D}$, let $\text{cases}(S, \mathcal{D})$ be the set of sequents

$$\{ \Sigma \theta : \Gamma \theta, B \theta \rightarrow F \theta \mid \forall \overline{x}. A' \triangleq B \in \mathcal{D} \text{ and } \theta \in \text{CSU}(A, A') \}$$

where $\Sigma \theta$ removes eigenvariables in the domain of $\theta$ and adds those in its range.
The Left Introduction Rule for Definitions

Let $S$ be the sequent $\Sigma : \Gamma, A \rightarrow F$, where $\Sigma$ represents the eigenvariable context.

Let $CSU(A, A')$ represents a complete set of unifiers for (the atomic formulas or terms) $A$ and $A'$.

For definition $\mathcal{D}$, let $\text{cases}(S, \mathcal{D})$ be the set of sequents

$$\{ \Sigma \theta : \Gamma \theta, B \theta \rightarrow F \theta \mid \forall x. A' \triangleq B \in \mathcal{D} \text{ and } \theta \in CSU(A, A') \}$$

where $\Sigma \theta$ removes eigenvariables in the domain of $\theta$ and adds those in its range.

Then the left introduction rule is the following

$$\frac{\text{cases}(\Sigma : \Gamma, A \rightarrow F, \mathcal{D})}{\Sigma : \Gamma, A \rightarrow F}$$
The Left Introduction Rule for Definitions

Let $S$ be the sequent $\Sigma : \Gamma, A \rightarrow F$, where $\Sigma$ represents the eigenvariable context.

Let $CSU(A, A')$ represents a complete set of unifiers for (the atomic formulas or terms) $A$ and $A'$.

For definition $\mathcal{D}$, let $cases(S, \mathcal{D})$ be the set of sequents

$$\{ \Sigma \theta : \Gamma \theta, B \theta \rightarrow F \theta \mid \forall \alpha. A' \equiv B \in \mathcal{D} \text{ and } \theta \in CSU(A, A') \}$$

where $\Sigma \theta$ removes eigenvariables in the domain of $\theta$ and adds those in its range.

Then the left introduction rule is the following

$$cases(\Sigma : \Gamma, A \rightarrow F, \mathcal{D}) \Rightarrow \Sigma : \Gamma, A \rightarrow F$$

A point to note: this rule is sensitive to type information.
The specification logic is encoded by capturing its derivation relation in a definition.

\[
\text{seq } \text{true} \equiv \top \\
\forall g_1 : o, g_2 : o. \text{seq}(g_1 \& g_2) \equiv (\text{seq } g_1) \land (\text{seq } g_2)
\]

\[
\forall a : o. \text{seq}(\text{atm } a) \equiv \exists g : o. (\text{prog } a g) \land (\text{seq } g)
\]

Note: this encoding relies on the specifications also being simply typed.
The specification logic is encoded by capturing its derivation relation in a definition.

For example, limiting to the Horn clause fragment, the latter can be done by the following definition for the seq predicate:

\[
\begin{align*}
\text{seq true} & \triangleq \top \\
\forall g_1 : o, g_2 : o. \text{seq } (g_1 \land g_2) & \triangleq (\text{seq } g_1) \land (\text{seq } g_2) \\
\forall a : o. \text{seq } (\text{atm } a) & \triangleq \exists g : o. (\text{prog } a g) \land (\text{seq } g)
\end{align*}
\]
The specification logic is encoded by capturing its derivation relation in a definition.

For example, limiting to the Horn clause fragment, the latter can be done by the following definition for the \texttt{seq} predicate

\[
\text{seq true} \triangleq \top \\
\forall g_1 : o, g_2 : o. \text{seq} (g_1 \land g_2) \triangleq (\text{seq } g_1) \land (\text{seq } g_2) \\
\forall a : o. \text{seq} (\text{atm } a) \triangleq \exists g : o. (\text{prog } a g) \land (\text{seq } g)
\]

where \texttt{prog} is used to encode particular specifications, e.g.

\[
\forall l : \text{list}. \text{prog} (\text{append } \text{nil } l l) \text{true} \triangleq \top \\
\forall x : \iota, l_1 : \text{list}, l_2 : \text{list}, l_3 : \text{list}. \\
\text{prog} (\text{append} (x :: l_1) l_2 (x :: l_3)) (\text{atm} (\text{append} l_1 l_2 l_3)) \triangleq \top
\]
The specification logic is encoded by capturing its derivation relation in a definition.

For example, limiting to the Horn clause fragment, the latter can be done by the following definition for the `seq` predicate:

\[
\begin{align*}
  \text{seq} \, \text{true} & \triangleq \top \\
  \forall g_1 : o, g_2 : o. \, \text{seq}(g_1 \land g_2) & \triangleq (\text{seq} \, g_1) \land (\text{seq} \, g_2) \\
  \forall a : o. \, \text{seq}(\text{atm} \, a) & \triangleq \exists g : o. (\text{prog} \, a \, g) \land (\text{seq} \, g)
\end{align*}
\]

where `prog` is used to encode particular specifications, e.g.

\[
\begin{align*}
  \forall l : \text{list}. \, \text{prog}(\text{append} \, \text{nil} \, l \, l) \, \text{true} & \triangleq \top \\
  \forall x : i, l_1 : \text{list}, l_2 : \text{list}, l_3 : \text{list}. \, \text{prog}(\text{append}(x :: l_1) \, l_2 \, (x :: l_3)) \, (\text{atm} \, (\text{append} \, l_1 \, l_2 \, l_3)) & \triangleq \top
\end{align*}
\]

**Note:** this encoding relies on the specifications also being simply typed.
Schematizing the Language

Realized by introducing type variables and mechanisms for using them in type and term formation

Examples:
-\[A\rightarrow (\text{list}\ A)\rightarrow (\text{list}\ A)\]
-\[[\text{int}]\]
-\[[\text{bool}]\]
-\[[\text{int}\rightarrow\text{bool}]\]

Permit type instantiation for constants in the type checking process underlying term formation.

Terms with type variables in their types represent a collection of simply typed terms.
Schematizing the Language

Realized by introducing type variables and mechanisms for using them in type and term formation

- Add type constructors and permit variables in type expressions
- Type term constants with type schemata that make explicit the parameterization, e.g.
  \[ : : : [A]A \rightarrow (\text{list}A) \rightarrow (\text{list}A) \]

Instances of constants depicted using types as subscripts, e.g., \( : : [\text{int}] \), \( : : [\text{bool}] \), \( : : [\text{int} \rightarrow \text{bool}] \)

- Permit type instantiation for constants in the type checking process underlying term formation
Schematizing the Language

Realized by introducing type variables and mechanisms for using them in type and term formation

- Add type constructors and permit variables in type expressions
- Type term constants with type schemata that make explicit the parameterization, e.g.

\[ A \rightarrow (\text{list}A) \rightarrow (\text{list}A) \]

Instances of constants depicted using types as subscripts, e.g., \([\text{int}], [\text{bool}], [\text{int}\rightarrow\text{bool}]\)

- Permit type instantiation for constants in the type checking process underlying term formation

Terms with type variables in their types represent a collection of simply typed terms
A clause parameterized by a list of type variables $\Psi$:

$$[[\Psi]] \forall x : \alpha. A \triangleq B$$

A proviso: all the type variables in the body must appear in the head of the clause.
Schematic Clauses

A clause parameterized by a list of type variables $\Psi$:

$$[\Psi] \forall x : \alpha. \ A \triangleq B$$

A proviso: all the type variables in the body must appear in the head of the clause

Such a clause represents a possibly infinite collection of clauses under type instantiation
Schematic Clauses

A clause parameterized by a list of type variables $\Psi$:

$$[\Psi] \forall x : \alpha. A \triangleq B$$

A proviso: all the type variables in the body must appear in the head of the clause

Such a clause represents a possibly infinite collection of clauses under type instantiation

This kind of parameterization permits the encoding of schematic specification logic clauses, e.g.

$$[A] \forall \ell : \text{list } A. \text{prog}(\text{append}[A] \text{ nil}[A] \ell \ell) \text{ true} \triangleq \top$$

$$[A] \forall x : A, \ell_1 : \text{list } A, \ell_2 : \text{list } A, \ell_3 : \text{list } A. \text{prog}(\text{append}[A] (x :: [A] \ell_1) \ell_2 (x :: [A] \ell_3)) \triangleq \top$$
A block can also be parameterized by type variable header

- All the type variables in the type of each predicate constant defined in the block must appear in the header
- Each defined predicate must appear at the most general type throughout the definition
A block can also be parameterized by type variable header

- All the type variables in the type of each predicate constant defined in the block must appear in the header
- Each defined predicate must appear at the most general type throughout the definition

For example, the polymorphic predicate

\[ \text{app} : [A]\text{list } A \rightarrow \text{list } A \rightarrow \text{list } A \rightarrow \text{prop} \]

is defined by the following block parameterized by \( A \):

\[
\forall \ell : \text{list } A. \ \text{app}[A] \ \text{nil}[A] \ \ell \ \ell \ \equiv \ \top \\
\forall x : A, \ell : \text{list } A, \ell_2 : \text{list } A, \ell_3 : \text{list } A. \ \\
\text{app}[A] (x :: [A] \ell_1) \ell_2 (x :: [A] \ell_3) \equiv \text{app}[A] \ \ell_1 \ \ell_2 \ \ell_3
\]
A block can also be parameterized by type variable header

- All the type variables in the type of each predicate constant defined in the block must appear in the header
- Each defined predicate must appear at the most general type throughout the definition

For example, the polymorphic predicate

\[
\text{app} : [A]\text{list } A \to \text{list } A \to \text{list } A \to \text{prop}
\]

is defined by the following block parameterized by \( A \):

\[
\forall \ell : \text{list } A. \ \text{app}[A] \text{ nil}[A] \ \ell \ \ell \ \triangleq \ \top
\]
\[
\forall x : A, \ell : \text{list } A, \ell_2 : \text{list } A, \ell_3 : \text{list } A. \\
\text{app}[A] (x :: [A] \ell_1) \ell_2 (x :: [A] \ell_3) \triangleq \text{app}[A] \ell_1 \ell_2 \ell_3
\]

A schematic block represents actual definition blocks generated by type instantiation
A schematic theorem is a formula with type variables that is provable at *all its potential type instances*
A schematic theorem is a formula with type variables that is provable at *all its potential type instances*

We allow only for the construction of proofs that are themselves schematic wrt types, based on the following ideas
A schematic theorem is a formula with type variables that is provable at \textit{all its potential type instances}.

We allow only for the construction of proofs that are themselves schematic wrt types, based on the following ideas:

- Proof states are represented as sequents parameterized by a set of type variables, i.e., of the form
  \[
  [\Psi] \Sigma : \Gamma \rightarrow F
  \]

  The lifting of proof rules is in fact straightforward except for the left introduction rule for definitions.
A schematic theorem is a formula with type variables that is provable at \textit{all its potential type instances}

We allow only for the construction of proofs that are themselves schematic wrt types, based on the following ideas:

- Proof states are represented as sequents parameterized by a set of type variables, i.e. of the form:
  \[
  [\psi] \Sigma : \Gamma \rightarrow F
  \]
  
- Proof rules are lifted in such a way that they hold fixed the collection of parameterizing type variables.
A schematic theorem is a formula with type variables that is provable at *all its potential type instances*

We allow only for the construction of proofs that are themselves schematic wrt types, based on the following ideas

- Proof states are represented as sequents parameterized by a set of type variables, i.e. of the form

\[
[\psi] \Sigma : \Gamma \rightarrow F
\]

- Proof rules are lifted in such a way that they hold fixed the collection of parameterizing type variables
- The type instantiation of the lifted rules must yield actual proof rules
A schematic theorem is a formula with type variables that is provable at *all its potential type instances*

We allow only for the construction of proofs that are themselves schematic wrt types, based on the following ideas

- Proof states are represented as sequents parameterized by a set of type variables, i.e. of the form
  \[ \[\psi\] \Sigma : \Gamma \rightarrow F \]

- Proof rules are lifted in such a way that they hold fixed the collection of parameterizing type variables
- The type instantiation of the lifted rules must yield actual proof rules

The lifting of proof rules is in fact straightforward except for the left introduction rule for definitions
Key difficulty in lifting the def-L rule: the CSUs for different type instances of terms may not have the same structure.

As a consequence, the precise structure of the def-L rule may be different at different type instances.
The Schematic Definition Left Rule

Key difficulty in lifting the def-L rule: the CSUs for different type instances of terms may not have the same structure. As a consequence, the precise structure of the def-L rule may be different at different type instances.

We overcome this difficulty by using the following ideas:
Key difficulty in lifting the def-L rule: the CSUs for different type instances of terms may not have the same structure.
As a consequence, the precise structure of the def-L rule may be different at different type instances.

We overcome this difficulty by using the following ideas:

- Formalizing \textit{type-generic CSUs} for type-schematic terms.
Key difficulty in lifting the def-L rule: the CSUs for different type instances of terms may not have the same structure. As a consequence, the precise structure of the def-L rule may be different at different type instances.

We overcome this difficulty by using the following ideas:

- Formalizing *type-generic CSUs* for type-schematic terms
- Permitting case analysis against a clause only if one exists using such CSUs that covers all type instances
Key difficulty in lifting the def-L rule: the CSUs for different type instances of terms may not have the same structure.
As a consequence, the precise structure of the def-L rule may be different at different type instances.

We overcome this difficulty by using the following ideas:

- Formalizing *type-generic CSUs* for type-schematic terms
- Permitting case analysis against a clause only if one exists using such CSUs that covers all type instances
- Defining def-L to apply only if the concluding sequent has a type generic case analysis wrt *every* clause
Key difficulty in lifting the def-L rule: the CSUs for different type instances of terms may not have the same structure
As a consequence, the precise structure of the def-L rule may be different at different type instances

We overcome this difficulty by using the following ideas:

- Formalizing *type-generic CSUs* for type-schematic terms
- Permitting case analysis against a clause only if one exists using such CSUs that covers all type instances
- Defining def-L to apply only if the concluding sequent has a type generic case analysis wrt *every* clause

The premise sequents are then the collection of all the sequents resulting from case analysis on each clause
The schematic proof system is sound

**Theorem:**
Type instantiations of schematic proofs yield valid proofs in the underlying simply typed logic

For example, given $p : A \rightarrow \iota$ and $g : \iota \rightarrow \text{prop}$ defined by the clause:

$$\forall x : \text{nat}. g(p[nat][x]) \equiv \top,$$

consider $A = \text{nat}$: prove the left formula by backchaining $A \neq \text{nat}$: prove the right branch by case analysis

However, there is no schematic proof for the formula
The schematic proof system is sound

**Theorem:**
Type instantiations of schematic proofs yield valid proofs in the underlying simply typed logic

However, the system is not complete
Soundness and Completeness

The schematic proof system is sound

**Theorem:**
Type instantiations of schematic proofs yield valid proofs in the underlying simply typed logic

However, the system is not complete

For example, given \( p : [A]A \rightarrow \iota \) and \( g : \iota \rightarrow \text{prop} \) defined by the clause \( \forall x : \text{nat}. \ g (p_{\text{nat}} x) \triangleq \top \), consider

\[
[A] \forall x : A. (g (p_{A} x)) \lor (g (p_{A} x) \supset \bot)
\]
Soundness and Completeness

The schematic proof system is sound

**Theorem:**

Type instantiations of schematic proofs yield valid proofs in the underlying simply typed logic

However, the system is not complete

For example, given \( p : [A]A \rightarrow \iota \) and \( g : \iota \rightarrow \text{prop} \) defined by the clause \( \forall x : \text{nat}. \ g(p_{\text{nat}}\ x) \triangleq \top \), consider

\[
[A] \forall x : A. (g(p_{A} \ x)) \lor (g(p_{A} \ x) \supset \bot)
\]

Every type instance of this formula has a proof:

- \( A = \text{nat} \): prove the left formula by backchaining
- \( A \neq \text{nat} \): prove the right branch by case analysis
Soundness and Completeness

The schematic proof system is sound

**Theorem:**
Type instantiations of schematic proofs yield valid proofs in the underlying simply typed logic

However, the system is not complete

For example, given $p : [A]A \rightarrow \iota$ and $g : \iota \rightarrow \text{prop}$ defined by the clause $\forall x : \text{nat}. \ g (p_{\text{nat}}(x)) \triangleq \top$, consider

$$[A] \forall x : A. (g (p_{A}(x)) \lor (g (p_{A}(x)) \supset \bot))$$

Every type instance of this formula has a proof:

- $A = \text{nat}$: prove the left formula by backchaining
- $A \neq \text{nat}$: prove the right branch by case analysis

However, there is no schematic proof for the formula
These ideas have been developed to cover the full reasoning and specification logics underlying Abella.

They have also been implemented and used in our compiler verification work; see Yuting’s doctoral thesis.

This work builds on the approach to polymorphism in $\lambda$Prolog [Nadathur and Pfenning, 1992].

A light-weight approach that could be used in related systems like Twelf and Beluga.

Download Abella with schematic polymorphism at

https://github.com/abella-prover/abella/tree/schm-poly-type-unif

Official release coming soon!