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

The Abella Proof Assistant

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

The Abella Proof Assistant

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

The Abella Proof Assistant

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

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

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

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 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:

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

$$\forall x : \iota, \ell : \texttt{list}, \ell_2 : \texttt{list}, \ell_3 : \texttt{list}.$$

$$\texttt{app}\ (x :: \ell_1)\ \ell_2\ (x :: \ell_3) \triangleq \texttt{app}\ \ell_1\ \ell_2\ \ell_3$$

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

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

The Left Introduction Rule for Definitions

Let $\mathcal{S}$ be the sequent $\Sigma : \Gamma, A \longrightarrow F$, where Σ represents the eigenvariable context

The Left Introduction Rule for Definitions

Let $\mathcal{S}$ be the sequent $\Sigma : \Gamma, A \longrightarrow F$, where Σ 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 $\mathcal{S}$ be the sequent $\Sigma : \Gamma, A \longrightarrow F$, where Σ 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 $cases(\mathcal{S}, \mathcal{D})$ be the set of sequents

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

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

The Left Introduction Rule for Definitions

Let $\mathcal{S}$ be the sequent $\Sigma : \Gamma, A \longrightarrow F$, where Σ 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 $cases(\mathcal{S}, \mathcal{D})$ be the set of sequents

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

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

Then the left introduction rule is the following

$$\frac{cases(\Sigma : \Gamma, A \longrightarrow F, \mathcal{D})}{\Sigma : \Gamma, A \longrightarrow F}$$

The Left Introduction Rule for Definitions

Let $\mathcal{S}$ be the sequent $\Sigma : \Gamma, A \longrightarrow F$, where Σ represents the eigenvariable context

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

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

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

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

Then the left introduction rule is the following

$$\frac{\mathit{cases}(\Sigma : \Gamma, A \longrightarrow F, \mathcal{D})}{\Sigma : \Gamma, A \longrightarrow F}$$

A point to note: this rule is sensitive to type information

Encoding the Specification Logic

The specification logic is encoded by capturing its derivation relation in a definition

Encoding the Specification Logic

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

$$\texttt{seq}\ \texttt{true} \triangleq \top$$
$$\forall g_1 : \circ, g_2 : \circ.\ \texttt{seq}\ (g_1\ \&\ g_2) \triangleq (\texttt{seq}\ g_1) \wedge (\texttt{seq}\ g_2)$$
$$\forall a : \circ.\ \texttt{seq}\ (\texttt{atm}\ a) \triangleq \exists g : \circ.(\texttt{prog}\ a\ g) \wedge (\texttt{seq}\ g)$$

Encoding the Specification Logic

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

$$\texttt{seq}\ \texttt{true} \triangleq \top$$
$$\forall g_1 : o, g_2 : o.\ \texttt{seq}\ (g_1\ \&\ g_2) \triangleq (\texttt{seq}\ g_1) \wedge (\texttt{seq}\ g_2)$$
$$\forall a : o.\ \texttt{seq}\ (\texttt{atm}\ a) \triangleq \exists g : o.(\texttt{prog}\ a\ g) \wedge (\texttt{seq}\ g)$$

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

$$\forall \ell : \texttt{list}.\ \texttt{prog}\ (\texttt{append}\ \texttt{nil}\ \ell\ \ell)\ \texttt{true} \triangleq \top$$
$$\forall x : \iota, \ell_1 : \texttt{list}, \ell_2 : \texttt{list}, \ell_3 : \texttt{list}.$$
$$\texttt{prog}\ (\texttt{append}\ (x :: \ell_1)\ \ell_2\ (x :: \ell_3))\ (\texttt{atm}\ (\texttt{append}\ \ell_1\ \ell_2\ \ell_3)) \triangleq \top$$

Encoding the Specification Logic

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

$$\texttt{seq}\ \texttt{true} \triangleq \top$$
$$\forall g_1 : \texttt{o}, g_2 : \texttt{o}.\ \texttt{seq}\ (g_1\ \&\ g_2) \triangleq (\texttt{seq}\ g_1) \wedge (\texttt{seq}\ g_2)$$
$$\forall a : \texttt{o}.\ \texttt{seq}\ (\texttt{atm}\ a) \triangleq \exists g : \texttt{o}.(\texttt{prog}\ a\ g) \wedge (\texttt{seq}\ g)$$

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

$$\forall \ell : \texttt{list}.\ \texttt{prog}\ (\texttt{append}\ \texttt{nil}\ \ell\ \ell)\ \texttt{true} \triangleq \top$$
$$\forall x : \iota, \ell_1 : \texttt{list}, \ell_2 : \texttt{list}, \ell_3 : \texttt{list}.$$
$$\texttt{prog}\ (\texttt{append}\ (x :: \ell_1)\ \ell_2\ (x :: \ell_3))\ (\texttt{atm}\ (\texttt{append}\ \ell_1\ \ell_2\ \ell_3)) \triangleq \top$$

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

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.

  $$:: \quad : \quad [A]A \to (\texttt{list}\, A) \to (\texttt{list}\, A)$$

  Instances of constants depicted using types as subscripts, e.g., $::_{[\texttt{int}]}$, $::_{[\texttt{bool}]}$, $::_{[\texttt{int}\to\texttt{bool}]}$
- 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.

  $$:: \quad : \quad [A]A \to (\texttt{list}\, A) \to (\texttt{list}\, A)$$

  Instances of constants depicted using types as subscripts, e.g., $::_{[\texttt{int}]}$, $::_{[\texttt{bool}]}$, $::_{[\texttt{int}\to\texttt{bool}]}$
- 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

Schematic Clauses

A clause parameterized by a list of type variables Ψ:

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

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 Ψ:

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

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 Ψ:

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

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.

$$[A]\forall \ell : \texttt{list}\ A.\ \texttt{prog}\ (\texttt{append}_{[A]}\ \texttt{nil}_{[A]}\ \ell\ \ell)\ \texttt{true} \triangleq \top$$
$$[A]\forall x : A, \ell_1 : \texttt{list}\ A, \ell_2 : \texttt{list}\ A, \ell_3 : \texttt{list}\ A.$$
$$\texttt{prog}\ (\texttt{append}_{[A]}\ (x ::_{[A]} \ell_1)\ \ell_2\ (x ::_{[A]} \ell_3))$$
$$(\texttt{atm}\ (\texttt{append}_{[A]}\ \ell_1\ \ell_2\ \ell_3)) \triangleq \top$$

Schematic Definition Blocks

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

Schematic Definition Blocks

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

$$\mathtt{app} : [A]\mathtt{list}\ A \to \mathtt{list}\ A \to \mathtt{list}\ A \to \mathtt{prop}$$

is defined by the following block parameterized by A:

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

Schematic Definition Blocks

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

$$\texttt{app} : [A]\texttt{list}\ A \to \texttt{list}\ A \to \texttt{list}\ A \to \texttt{prop}$$

is defined by the following block parameterized by A:

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

A schematic block represents actual definition blocks generated by type instantiation

Schematic Theorems and their Proofs

A schematic theorem is a formula with type variables that is provable at *all its potential type instances*

Schematic Theorems and their Proofs

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

Schematic Theorems and their Proofs

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

$$[\Psi]\ \Sigma : \Gamma \longrightarrow F$$

Schematic Theorems and their Proofs

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

$$[\Psi]\ \Sigma : \Gamma \longrightarrow F$$

- Proof rules are lifted in such a way that they hold fixed the collection of parameterizing type variables

Schematic Theorems and their Proofs

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

$$[\Psi]\ \Sigma : \Gamma \longrightarrow F$$

- 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

Schematic Theorems and their Proofs

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

$$[\Psi]\ \Sigma : \Gamma \longrightarrow F$$

- 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

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

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:

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:

- Formalizing *type-generic CSUs* for type-schematic terms

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:

- 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

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:

- 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 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:

- 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

Soundness and Completeness

The schematic proof system is sound

Theorem:

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

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

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 $\mathtt{p} : [A]A \to \iota$ and $\mathtt{g} : \iota \to \mathtt{prop}$ defined by the clause $\forall x : \mathtt{nat}.\ \mathtt{g}\ (\mathtt{p}_{[\mathtt{nat}]}\ x) \triangleq \top$, consider

$$[A]\forall x : A.(\mathtt{g}\ (\mathtt{p}_{[A]}\ x)) \vee (\mathtt{g}\ (\mathtt{p}_{[A]}\ x) \supset \bot)$$

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 $\mathtt{p} : [A]A \to \iota$ and $\mathtt{g} : \iota \to \mathtt{prop}$ defined by the clause $\forall x : \mathtt{nat}.\ \mathtt{g}\ (\mathtt{p}_{[\mathtt{nat}]}\ x) \triangleq \top$, consider

$$[A]\forall x : A.(\mathtt{g}\ (\mathtt{p}_{[A]}\ x)) \vee (\mathtt{g}\ (\mathtt{p}_{[A]}\ x) \supset \bot)$$

Every type instance of this formula has a proof:

- $A = \mathtt{nat}$: prove the left formula by backchaining
- $A \neq \mathtt{nat}$: 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 $\mathtt{p} : [A]A \to \iota$ and $\mathtt{g} : \iota \to \mathtt{prop}$ defined by the clause $\forall x : \mathtt{nat}.\ \mathtt{g}\ (\mathtt{p}_{[\mathtt{nat}]}\ x) \triangleq \top$, consider

$$[A]\forall x : A.(\mathtt{g}\ (\mathtt{p}_{[A]}\ x)) \vee (\mathtt{g}\ (\mathtt{p}_{[A]}\ x) \supset \bot)$$

Every type instance of this formula has a proof:

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

However, there is no schematic proof for the formula

Conclusion

- 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 λProlog [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!