A Higher-Order Abstract Syntax Approach to Verified Compilation of Functional Programs

Yuting Wang and Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota, Minneapolis

ESOP 2016, Eindhoven, Netherlands
Formal verification is the only way to guarantee the absolute correctness of software systems.
Motivation for Verified Compilation

Formal verification is the only way to guarantee the absolute correctness of software systems.

Gap in the formal verification of programs:

- Programs are proved correct relative to the model of the high-level language in which they are written.
- Programs are executed only after compilation into low-level code.
Motivation for Verified Compilation

Formal verification is the only way to guarantee the absolute correctness of software systems.

Gap in the formal verification of programs:

- Programs are proved correct relative to the model of the high-level language in which they are written.
- Programs are executed only after compilation into low-level code.

To close the gap, we must also formally verify the compilation process.
Motivation for Verified Compilation

Formal verification is the only way to guarantee the absolute correctness of software systems.

Gap in the formal verification of programs:

- Programs are proved correct relative to the model of the high-level language in which they are written.
- Programs are executed only after compilation into low-level code.

To close the gap, we must also formally verify the compilation process.

Our interest is in verifying compiler transformations for functional programming languages.
Compilation consists of two phases:

- Transforming arbitrary functional programs into a simplified form
- Using standard techniques to compile the simplified programs
Compilation consists of two phases:

- Transforming arbitrary functional programs into a simplified form
- Using standard techniques to compile the simplified programs

Our focus is on the implementation and verification of the first phase
Compilation consists of two phases:

- Transforming arbitrary functional programs into a simplified form
- Using standard techniques to compile the simplified programs

Our focus is on the implementation and verification of the first phase

Characteristics of the transformations in the first phase:

- Transformations are naturally described via syntax-directed rules
- Transformations manipulate binding structure in complex ways
Compilation consists of two phases:

- Transforming arbitrary functional programs into a simplified form
- Using standard techniques to compile the simplified programs

Our focus is on the implementation and verification of the first phase

Characteristics of the transformations in the first phase:

- Transformations are naturally described via syntax-directed rules
- Transformations manipulate binding structure in complex ways

The content of our work

A rich form of higher-order abstract syntax (HOAS) has benefits in implementing and verifying such transformations
We make the case using a framework comprising the specification language \( \lambda \text{Prolog} \) and the interactive theorem prover Abella. We show that \( \lambda \text{Prolog} \) supports a concise, declarative implementation of the transformations. We show that using Abella we can construct elegant proofs of correctness for the \( \lambda \text{Prolog} \) programs. We argue that these benefits in fact derive from the underlying support for HOAS and rule-based relational specifications. This talk focuses on typed closure conversion to make these points.
We make the case using a framework comprising the specification language $\lambda$Prolog and the interactive theorem prover Abella

- We show that $\lambda$Prolog supports a concise, declarative implementation of the transformations

- We show that using Abella we can construct elegant proofs of correctness for the $\lambda$Prolog programs

- We argue that these benefits in fact derive from the underlying support for HOAS and rule-based relational specifications
We make the case using a framework comprising the specification language $\lambda$Prolog and the interactive theorem prover Abella

- We show that $\lambda$Prolog supports a concise, declarative implementation of the transformations

- We show that using Abella we can construct elegant proofs of correctness for the $\lambda$Prolog programs

- We argue that these benefits in fact derive from the underlying support for HOAS and rule-based relational specifications

This talk focuses on *typed closure conversion* to make these points
The Closure Conversion Transformation

A transformation that replaces (nested) functions by closed functions paired with environments with bindings for the free variables.

For example,

```plaintext
let x = 3 in let y = 4 in fn z => x + y + z
```

is transformed into

```plaintext
let x = 3 in let y = 4 in
<(fn z e => e.1 + e.2 + z), (x, y)>
```

Binding structure and substitution are central to this transformation:

- Calculating the free variables in a nested function
- Replacing these variables with projections from an environment

Not only must these operations be implemented, the implementations must also be shown to preserve meanings of programs.

Yuting Wang and Gopalan Nadathur

Verified Transformations on Functional Programs
The Closure Conversion Transformation

A transformation that replaces (nested) functions by closed functions paired with environments with bindings for the free variables

For example,

```ml
let x = 3 in let y = 4 in
fn z => x + y + z
```

is transformed into

```ml
let x = 3 in let y = 4 in
<(fn z e => e.1 + e.2 + z), (x, y)>
```
The Closure Conversion Transformation

A transformation that replaces (nested) functions by closed functions paired with environments with bindings for the free variables

For example,

\[
\begin{align*}
\text{let } x = 3 \text{ in let } y = 4 \text{ in } \\
\text{ fn } z \Rightarrow x + y + z
\end{align*}
\]

is transformed into

\[
\begin{align*}
\text{let } x = 3 \text{ in let } y = 4 \text{ in } \\
\langle\text{fn } z \ e \Rightarrow e.1 + e.2 + z\rangle, \ (x, \ y) >
\end{align*}
\]

Binding structure and substitution are central to this transformation:

- Calculating the free variables in a nested function
- Replacing these variables with projections from an environment
The Closure Conversion Transformation

A transformation that replaces (nested) functions by closed functions paired with environments with bindings for the free variables

For example,

```plaintext
let x = 3 in let y = 4 in
    fn z => x + y + z
```

is transformed into

```plaintext
let x = 3 in let y = 4 in
    <(fn z e => e.1 + e.2 + z), (x, y)>
```

Binding structure and substitution are central to this transformation:

- Calculating the free variables in a nested function
- Replacing these variables with projections from an environment

Not only must these operations be implemented, the implementations must also be shown to preserve meanings of programs
The language is based on logic programming style clauses that transparently encode *rule-based relational specifications*.
The language is based on logic programming style clauses that transparently encode *rule-based relational specifications*

For example, consider the append relation specified by the rules

\[
\begin{align*}
\text{append } [] & \mapsto \text{[]} \\
\text{append } (X :: L1) & \mapsto (X :: L3)
\end{align*}
\]

Notation: 

\[ L \vdash G \] asserts that \( G \) is derivable from a set \( L \) of clauses.
The language is based on logic programming style clauses that transparently encode *rule-based relational specifications*

For example, consider the append relation specified by the rules

\[
\begin{align*}
\text{append} & \quad [\text{nil} \mid l] \quad [\text{l} \\ l] \\
\text{append} & \quad (x :: l_1) \quad l_2 \quad (x :: l_3)
\end{align*}
\]

These rules are captured directly in Prolog-like logical clauses:

\[
\begin{align*}
\text{append} \; \text{nil} & \; L \; L. \\
\text{append} \; (X :: L_1) & \; L_2 \; (X :: L_3) : - \; \text{append} \; L_1 \; L_2 \; L_3.
\end{align*}
\]
The language is based on logic programming style clauses that transparently encode *rule-based relational specifications*

For example, consider the append relation specified by the rules

\[
\begin{align*}
\text{append} & \quad \text{[]} / \text{I} \\
\text{append} & \quad \text{(X :: L1)} \text{L2} (\text{X :: L3}) & \quad \text{:- append L1 L2 L3}.
\end{align*}
\]

These rules are captured directly in Prolog-like logical clauses:

\[
\begin{align*}
\text{append} & \quad \text{nil L L.} \\
\text{append} & \quad \text{(X :: L1)} \text{L2} (\text{X :: L3}) & \quad \text{:- append L1 L2 L3.}
\end{align*}
\]

A key point: These clauses are *both* logical specifications *and* executable as programs
The language is based on logic programming style clauses that transparently encode *rule-based relational specifications*.

For example, consider the append relation specified by the rules:

\[
\begin{align*}
\text{append} & \quad [] \quad [] \\
\text{append} & \quad (X :: L1) \quad L2 \quad (X :: L3)
\end{align*}
\]

These rules are captured directly in Prolog-like logical clauses:

\[
\begin{align*}
\text{append} & \quad \text{nil} \quad L \quad L. \\
\text{append} & \quad (X :: L1) \quad L2 \quad (X :: L3) \quad :- \quad \text{append} \quad L1 \quad L2 \quad L3.
\end{align*}
\]

A key point: These clauses are *both* logical specifications *and* executable as programs.

Notation: \( L \vdash G \) asserts that \( G \) is derivable from a set \( L \) of clauses.
A higher-order treatment of abstract syntax is supported in λProlog through the following devices:

A simply typed λ-calculus is used to represent objects. Object-level binding can be encoded via meta-level abstraction:

\[
\begin{align*}
\text{abs} &: (\text{tm} \rightarrow \text{tm}) \rightarrow \text{tm} \\
\text{app} &: \text{tm} \rightarrow \text{tm} \rightarrow \text{tm}
\end{align*}
\]

\[\left(\lambda x.\lambda y. x\ y\right) = \Rightarrow \text{abs}\ (x\ \text{abs}\ (y\ \text{app}\ x\ y))\]

Capturing substitution related notions through β-conversion. Substitution modulo β-reduction respects meta-level binding.

Supporting binding-sensitive structure analysis through unification modulo λ-convertibility. Realizing recursion over binding structure via hypothetical and generic goals:

\[
\Gamma, x: \alpha \vdash t: \beta \\
\Gamma \vdash \lambda x. t: \alpha \rightarrow \beta
\]

\[x / \in \text{dom}(\Gamma) \Rightarrow \text{of}\ (\text{abs}\ T)\ (\text{arr}\ Ty1\ Ty2):
\]

\[
\text{of}\ x\ Ty1 \Rightarrow \text{of}\ (T\ x)\ Ty2.
\]
A higher-order treatment of abstract syntax is supported in λProlog through the following devices:

- A simply typed λ-calculus is used to represent objects
Treating Binding Structure in \( \lambda \)Prolog

A higher-order treatment of abstract syntax is supported in \( \lambda \)Prolog through the following devices:

- A simply typed \( \lambda \)-calculus is used to represent objects
  
Object-level binding can be encoded via *meta-level abstraction*

\[
\text{abs} : (\text{tm} \to \text{tm}) \to \text{tm} \quad \text{app} : \text{tm} \to \text{tm} \to \text{tm}
\]

\[
(\lambda x. \lambda y. x y) = \Rightarrow \text{abs} (x \text{abs} (y \text{app} x y))
\]

Capturing substitution related notions through \( \beta \)-conversion

Substitution modulo \( \beta \)-reduction respects meta-level binding

Supporting binding-sensitive structure analysis through unification modulo \( \lambda \)-convertibility

Realizing recursion over binding structure via hypothetical and generic goals

\[
\Gamma, x : \alpha \vdash t : \beta \\
\Gamma \vdash \lambda x. t : \alpha \to \beta \\
x / \in \text{dom} (\Gamma)
\Rightarrow \text{of} (\text{abs} T) \ (\text{arr} Ty1 Ty2):=
\]

\[
\pi x \Rightarrow \text{of} x Ty1 \Rightarrow \text{of} (T x) Ty2.
\]

Yuting Wang and Gopalan Nadathur
Verified Transformations on Functional Programs
A higher-order treatment of abstract syntax is supported in λProlog through the following devices:

- A simply typed λ-calculus is used to represent objects
  
  Object-level binding can be encoded via *meta-level abstraction*

\[
\begin{align*}
\text{abs} & : (\text{tm} \to \text{tm}) \to \text{tm} \\
\text{app} & : \text{tm} \to \text{tm} \to \text{tm} \\
(\lambda x. \lambda y. x \, y) & \implies \text{abs} (x \setminus \text{abs} (y \setminus \text{app} \, x \, y))
\end{align*}
\]
A higher-order treatment of abstract syntax is supported in λProlog through the following devices:

- A simply typed λ-calculus is used to represent objects
  
  Object-level binding can be encoded via *meta-level abstraction*

  \[
  \text{abs} : (\text{tm} \rightarrow \text{tm}) \rightarrow \text{tm} \quad \text{app} : \text{tm} \rightarrow \text{tm} \rightarrow \text{tm} \\
  (\lambda x. \lambda y. x \ y) \Rightarrow \text{abs} (x\ \text{abs} (y\ \text{app} x y))
  \]

- Capturing substitution related notions through \(\beta\)-conversion
A higher-order treatment of abstract syntax is supported in \( \lambda \)Prolog through the following devices:

- A simply typed \( \lambda \)-calculus is used to represent objects
  
  Object-level binding can be encoded via *meta-level abstraction*
  
  \[
  \begin{align*}
  \text{abs} & : (\text{tm} \to \text{tm}) \to \text{tm} \quad \text{app} : \text{tm} \to \text{tm} \to \text{tm} \\
  (\lambda x. \lambda y. x \ y) & \implies \text{abs} (x \ \text{abs} (y \ \text{app} x \ y))
  \end{align*}
  \]

- Capturing substitution related notions through \( \beta \)-conversion
  
  Substitution modulo \( \beta \)-reduction respects meta-level binding
A higher-order treatment of abstract syntax is supported in \( \lambda \text{Prolog} \) through the following devices:

- A simply typed \( \lambda \)-calculus is used to represent objects
  
  Object-level binding can be encoded via *meta-level abstraction*
  
  \[
  \begin{align*}
  \text{abs} & : (\text{tm} \rightarrow \text{tm}) \rightarrow \text{tm} \\
  \text{app} & : \text{tm} \rightarrow \text{tm} \rightarrow \text{tm}
  \end{align*}
  \]
  \[
  (\lambda x. \lambda y. x \ y) \implies \text{abs} \ (x \ \text{abs} \ (y \ \text{app} \ x \ y))
  \]

- Capturing substitution related notions through \( \beta \)-conversion
  
  Substitution modulo \( \beta \)-reduction respects meta-level binding

- Supporting *binding-sensitive structure analysis* through unification modulo \( \lambda \)-convertibility
A higher-order treatment of abstract syntax is supported in $\lambda$Prolog through the following devices:

- A simply typed $\lambda$-calculus is used to represent objects
  
  Object-level binding can be encoded via *meta-level abstraction*
  
  $$\text{abs} : (\text{tm} \rightarrow \text{tm}) \rightarrow \text{tm} \quad \text{app} : \text{tm} \rightarrow \text{tm} \rightarrow \text{tm}$$
  $$\lambda x.\lambda y. x \; y \quad \Rightarrow \quad \text{abs} (x \setminus \text{abs} (y \setminus \text{app} \; x \; y))$$

- Capturing substitution related notions through $\beta$-conversion
  
  Substitution modulo $\beta$-reduction respects meta-level binding

- Supporting *binding-sensitive structure analysis* through unification modulo $\lambda$-convertibility

- Realizing recursion over binding structure via *hypothetical* and *generic* goals
A higher-order treatment of abstract syntax is supported in \( \lambda \text{Prolog} \) through the following devices:

- **A simply typed \( \lambda \)-calculus is used to represent objects**
  
  Object-level binding can be encoded via *meta-level abstraction*
  
  \[
  \begin{align*}
  \text{abs} &: \ (\text{tm} \rightarrow \text{tm}) \rightarrow \text{tm} \\
  \text{app} &: \ \text{tm} \rightarrow \text{tm} \rightarrow \text{tm}
  \end{align*}
  \]
  
  \( (\lambda x.\lambda y.x\ y) \implies \text{abs} (x\ \text{abs} (y\ \text{app} x\ y)) \)

- **Capturing substitution related notions through \( \beta \)-conversion**
  
  Substitution modulo \( \beta \)-reduction respects meta-level binding

- **Supporting *binding-sensitive structure analysis*** through
  
  unification modulo \( \lambda \)-convertibility

- **Realizing recursion over binding structure via *hypothetical* and
  generic goals**

\[
\begin{align*}
\Gamma, x : \alpha \vdash t : \beta & \quad \Rightarrow \quad \text{of} \ (\text{abs} \ T) \ (\text{arr} \ Ty1 \ Ty2) ::- \\
\Gamma \vdash \lambda x.t : \alpha \rightarrow \beta & \quad \Rightarrow \quad \pi x\ \\
x \notin \text{dom}(\Gamma) & \quad \Rightarrow \quad \text{of} \ x \ Ty1 \Rightarrow \text{of} \ (T \ x) \ Ty2.
\end{align*}
\]
The transformation is parameterized by a mapping $\rho$ of (source language) free variables to target language expressions.
Rule-Based Specification of Closure Conversion

The transformation is parameterized by a mapping $\rho$ of (source language) free variables to target language expressions.

We represent the closure conversion judgment as follows:

$$\rho \triangleright M \leadsto M'$$
The transformation is parameterized by a mapping $\rho$ of (source language) free variables to target language expressions.

We represent the closure conversion judgment as follows:

$$\rho \triangleright M \leadsto M'$$

The key rule is for transforming (nested) functions into closures:

$$\begin{array}{c}
(x_1, \ldots, x_n) = \text{fvars}(\lambda x. M) \quad \rho \triangleright (x_1, \ldots, x_n) \leadsto M_e \quad \rho' \triangleright M \leadsto M' \\
\rho \triangleright \lambda x. M \leadsto \langle \lambda y. \lambda x_e. M', M_e \rangle
\end{array}$$

where $\rho' = [x \rightarrow y, x_1 \rightarrow \pi_1(x_e), \ldots, x_n \rightarrow \pi_n(x_e)]$ and $y, x_e$ are fresh variables.
Rule-Based Specification of Closure Conversion

The transformation is parameterized by a mapping $\rho$ of (source language) free variables to target language expressions.

We represent the closure conversion judgment as follows:

$$\rho \triangleright M \rightsquigarrow M'$$

The key rule is for transforming (nested) functions into closures

$$(x_1, \ldots, x_n) = \text{fvars}(\lambda x. M) \quad \rho \triangleright (x_1, \ldots, x_n) \rightsquigarrow M_e \quad \rho' \triangleright M \rightsquigarrow M' \quad \rho \triangleright \lambda x. M \rightsquigarrow \langle \lambda y. \lambda x_e. M', M_e \rangle$$

where $\rho' = [x \rightarrow y, x_1 \rightarrow \pi_1(x_e), \ldots, x_n \rightarrow \pi_n(x_e)]$ and $y, x_e$ are fresh variables.

Computing free variables in the abstraction

Yuting Wang and Gopalan Nadathur
Verified Transformations on Functional Programs 8/17
The transformation is parameterized by a mapping $\rho$ of (source language) free variables to target language expressions.

We represent the closure conversion judgment as follows:

$$\rho \triangleright M \leadsto M'$$

The key rule is for transforming (nested) functions into closures:

$$\begin{array}{c}
(x_1, \ldots, x_n) = \text{fvars}(\lambda x.M) \\
\rho \triangleright (x_1, \ldots, x_n) \leadsto M_e \\
\rho' \triangleright M \leadsto M'
\end{array}$$

where $\rho' = [x \rightarrow y, x_1 \rightarrow \pi_1(x_e), \ldots, x_n \rightarrow \pi_n(x_e)]$ and $y, x_e$ are fresh variables

Creating an environment from bindings for the free variables...
Rule-Based Specification of Closure Conversion

The transformation is parameterized by a mapping $\rho$ of (source language) free variables to target language expressions.

We represent the closure conversion judgment as follows:

$$\rho \triangleright M \leadsto M'$$

The key rule is for transforming (nested) functions into closures

$$\frac{(x_1, \ldots, x_n) = \text{fvars}(\lambda x. M) \quad \rho \triangleright (x_1, \ldots, x_n) \leadsto M_e \quad \rho' \triangleright M \leadsto M'}{\rho \triangleright \lambda x. M \leadsto \langle \lambda y. \lambda x_e. M', M_e \rangle}$$

where $\rho' = [x \rightarrow y, x_1 \rightarrow \pi_1(x_e), \ldots, x_n \rightarrow \pi_n(x_e)]$ and $y, x_e$ are fresh variables.

Creating a mapping from free variables to projections to the environment.
The transformation is parameterized by a mapping $\rho$ of (source language) free variables to target language expressions.

We represent the closure conversion judgment as follows:

$$\rho \triangleright M \leadsto M'$$

The key rule is for transforming (nested) functions into closures:

$$\begin{array}{c}
(x_1, \ldots, x_n) = \text{fvars}(\lambda x. M) \quad \rho \triangleright (x_1, \ldots, x_n) \leadsto M_e \quad \rho' \triangleright M \leadsto M' \\
\rho \triangleright \lambda x. M \leadsto \langle \lambda y. \lambda x_e. M', M_e \rangle \\
\end{array}$$

where $\rho' = [x \rightarrow y, x_1 \rightarrow \pi_1(x_e), \ldots, x_n \rightarrow \pi_n(x_e)]$ and $y, x_e$ are fresh variables.
Computing Free Variables

We want to define $fvars$ such that $fvars \; M \; Vs \; FVs$ holds if

- $M$ is a source language term
- $Vs$ contains all the free variables in $M$
- $FVs$ contains exactly the free variables in $M$
We want to define \( fvars \) such that \( fvars M Vs FVs \) holds if

- \( M \) is a source language term
- \( Vs \) contains all the free variables in \( M \)
- \( FVs \) contains exactly the free variables in \( M \)

The difficulty: \( M \) may contain abstractions and then we will need to distinguish between free and bound variables in it.
We want to define \textit{fvars} such that \textit{fvars M Vs FVs} holds if
\begin{itemize}
  \item \textit{M} is a source language term
  \item \textit{Vs} contains all the free variables in \textit{M}
  \item \textit{FVs} contains exactly the free variables in \textit{M}
\end{itemize}

The difficulty: \textit{M} may contain abstractions and then we will need to distinguish between free and bound variables in it.

We can organize this computation in a \textit{logical} way in \textit{\lambda}Prolog:
\begin{itemize}
  \item For each abstraction encountered in the recursion over \textit{M}, introduce a new constant and mark it as bound
  \item Collect the variables encountered in \textit{M} that are not so marked
\end{itemize}
We want to define \textit{fvars} such that \textit{fvars }\textit{M Vs FVs} holds if

- \textit{M} is a source language term
- \textit{Vs} contains all the free variables in \textit{M}
- \textit{FVs} contains exactly the free variables in \textit{M}

The difficulty: \textit{M} may contain abstractions and then we will need to distinguish between free and bound variables in it

We can organize this computation in a \textit{logical} way in \textit{\lambda}Prolog:

- For each abstraction encountered in the recursion over \textit{M}, introduce a new constant and mark it as bound
- Collect the variables encountered in \textit{M} that are not so marked

Some clauses in the definition of \textit{fvars} that illustrate these ideas

\begin{verbatim}
fvars (abs M) Vs FVs :-
    pi y\ bound y => fvars (M y) Vs FVs.
fvars X _ nil :- bound X.
fvars Y Vs (Y :: nil) :- member Y Vs.
...
\end{verbatim}
We need to generate environments representing bindings for free variables and mappings from such environments for these variables.
We need to generate environments representing bindings for free variables and mappings from such environments for these variables.

We realize this by defining the predicates \( \text{mapvar} \) and \( \text{mapenv} \) s.t.

- \( \text{mapenv} \ Map \ FVs \ Env \) holds if \( Env \) is the reified environment for \( FVs \) based on \( Map \).

- \( \text{mapvar} \ FVs \ E \ Map \) holds if \( Map \) is the projection map on \( E \) for the variables in \( FVs \).
Creating Maps and Reifying the Environment

We need to generate environments representing bindings for free variables and mappings from such environments for these variables.

We realize this by defining the predicates `mapvar` and `mapenv` s.t.

- \( \text{mapenv } \text{Map } \text{FVs } \text{Env} \) holds if \( \text{Env} \) is the reified environment for \( \text{FVs} \) based on \( \text{Map} \)

- \( \text{mapvar } \text{FVs } E \text{ Map} \) holds if \( \text{Map} \) is the projection map on \( E \) for the variables in \( \text{FVs} \)

These definitions are easy once we have fixed representations for environments and mappings.
Creating Maps and Reifying the Environment

We need to generate environments representing bindings for free variables and mappings from such environments for these variables.

We realize this by defining the predicates \( \text{mapvar} \) and \( \text{mapenv} \) s.t.

\[
\text{mapenv} \ Map \ FVs \ Env \text{ holds if } Env \text{ is the reified environment for } FVs \text{ based on } Map
\]

\[
\text{mapvar} \ FVs \ E \ Map \text{ holds if } Map \text{ is the projection map on } E \text{ for the variables in } FVs
\]

These definitions are easy once we have fixed representations for environments and mappings.

For the latter, we use a list of items of the form \((\text{map} \ X \ T)\) encoding the mapping of the variable \(X\) to the term \(T\).
We want to define the predicate \( cc \) so that \( cc \ Map \ Vs \ M \ M' \) holds if:

- \( Map \) is a mapping of the free variables to target language terms
- \( Vs \) contains all the free variables in \( M \)
- \( M \) is a source language term
- \( M' \) is the result of the transformation
Implementing Closure Conversion

We want to define the predicate $cc$ so that $cc\ Map\ Vs\ M\ M'$ holds if

- $Map$ is a mapping of the free variables to target language terms
- $Vs$ contains all the free variables in $M$
- $M$ is a source language term
- $M'$ is the result of the transformation

The clause in the definition of this predicate that encodes the rule for transforming an abstraction:

$$cc\ Map\ Vs\ M\ M'::-
\begin{array}{l}
\pi\ x\ \pi\ y\ \\
\pi\ xenv\ \\
\text{fvars}\ (abs\ M)\ Vs\ FVs,\ \\
\text{mapenv}\ Map\ FVs\ PE,\ \\
\text{mapvar}\ FVs\ xenv\ NMap,
\end{array}
\text{cc}\ ((\text{map}\ x\ y)\ ::\ NMap)\ (x\ ::\ FVs)\ (M\ x)\ (P\ xenv\ y).$$

Note how the side conditions relating to names and all other aspects of the rule are given a logical treatment.
We want to define the predicate $cc$ so that $cc \ Map \ Vs \ M \ M'$ holds if

- $Map$ is a mapping of the free variables to target language terms
- $Vs$ contains all the free variables in $M$
- $M$ is a source language term
- $M'$ is the result of the transformation

The clause in the definition of this predicate that encodes the rule for transforming an abstraction:

$cc \ Map \ Vs$

$\ (abs \ M)$

$(clos \ (abs' \ (y\ abs' \ (xenv\ \ \ \ )))))$

$_\ )$
We want to define the predicate \textit{cc} so that \textit{cc} Map Vs M M' holds if

- \textit{Map} is a mapping of the free variables to target language terms
- \textit{Vs} contains all the free variables in \textit{M}
- \textit{M} is a source language term
- \textit{M'} is the result of the transformation

The clause in the definition of this predicate that encodes the rule for transforming an abstraction:

\[
\text{cc Map Vs (abs M) (clos (abs' (y \ abs' (xenv\ ________))))) __) :- (fvars (abs M) Vs FVs,)}
\]
Implementing Closure Conversion

We want to define the predicate \( cc \) so that \( cc \ Map \ Vs \ M \ M' \) holds if

- \( Map \) is a mapping of the free variables to target language terms
- \( Vs \) contains all the free variables in \( M \)
- \( M \) is a source language term
- \( M' \) is the result of the transformation

The clause in the definition of this predicate that encodes the rule for transforming an abstraction:

\[
cc \ Map \ Vs \\
(abs \ M) \\
(clos (abs' (y \ abs' (xenv\ ________))))
\]

\[
PE) :- \\
( \\
fvars (abs M) Vs FVs, \\
mapenv Map FVs PE, \\
).
\]
Implementing Closure Conversion

We want to define the predicate $cc$ so that $cc \ Map \ Vs \ M \ M'$ holds if

- $Map$ is a mapping of the free variables to target language terms
- $Vs$ contains all the free variables in $M$
- $M$ is a source language term
- $M'$ is the result of the transformation

The clause in the definition of this predicate that encodes the rule for transforming an abstraction:

$$
cc \ Map \ Vs \\
(abs \ M) \\
(clos \ (abs' \ (y \ abs' \ (xenv\ ________)))) \\
PE) :- \\
( \\
pi \ xenv\ \\
fvars \ (abs \ M) \ Vs \ FVs, \\
mapenv \ Map \ FVs \ PE, \\
mapvar \ FVs \ xenv \ NMap, \\
).
$$
We want to define the predicate \( \text{cc} \) so that \( \text{cc} \ Map \ Vs \ M \ M' \) holds if

- \( Map \) is a mapping of the free variables to target language terms
- \( Vs \) contains all the free variables in \( M \)
- \( M \) is a source language term
- \( M' \) is the result of the transformation

The clause in the definition of this predicate that encodes the rule for transforming an abstraction:

\[
\text{cc Map Vs (abs M) (clos (abs' (y \ abs' (xenv\ P xenv y)))) PE)} :- \\
(pi x \ pi y \ pi xenv\ fvars (abs M) Vs FVs, \\
mapenv Map FVs PE, \\
mapvar FVs xenv NMap, \\
cc ((map x y) :: NMap) (x :: FVs) (M x) (P xenv y)).
\]
Implementing Closure Conversion

We want to define the predicate $cc$ so that $cc \ Map \ Vs \ M \ M'$ holds if

- $Map$ is a mapping of the free variables to target language terms
- $Vs$ contains all the free variables in $M$
- $M$ is a source language term
- $M'$ is the result of the transformation

The clause in the definition of this predicate that encodes the rule for transforming an abstraction:

$$cc \ Map \ Vs$$
$$\quad (abs \ M)$$
$$\quad (clos \ (abs' \ (y \ abs' \ (xenv \ P \ xenv \ y))))$$
$$\quad PE) :-$$
$$\quad (pi \ x \ pi \ y \ pi \ xenv$$
$$\quad \ fvars \ (abs \ M) \ Vs \ FVs,$$
$$\quad \ mapenv \ Map \ FVs \ PE,$$
$$\quad \ mapvar \ FVs \ xenv \ NMap,$$
$$\quad cc \ ((map \ x \ y) :: NMap) \ (x :: FVs) \ (M \ x) \ (P \ xenv \ y)) .$$

Note how the side conditions relating to names and all other aspects of the rule are given a logical treatment.
Abella also encodes relational specifications but does this in a way that we can *reason* about them.
Abella also encodes relational specifications but does this in a way that we can *reason* about them.

- Relations are encoded through clauses of the form:

  \[ \forall \vec{X}. H(\vec{X}) \triangleq B(\vec{X}) \]

Such definitions get a fixed-point interpretation, allowing for case analysis based reasoning.

In fact, definitions can be given a least fixed-point interpretation, leading to inductive reasoning.

\[ \forall L_1 L_2, \text{append} \text{nil} L_1 L_2 \supset L_1 = L_2 \]

Abella also uses λ-terms for representing objects and has a special quantifier \( \bigtriangledown \) for a proof-level treatment of such binders.
Abella also encodes relational specifications but does this in a way that we can *reason* about them.

- Relations are encoded through clauses of the form:

\[ \forall \vec{X}. H(\vec{X}) \triangleq B(\vec{X}) \]

- Examples of definitions:

\[
\begin{align*}
\text{append } \text{nil } L L & \triangleq \top; \\
\text{append } (X :: L_1) L_2 (X :: L_3) & \triangleq \text{append } L_1 L_2 L_3
\end{align*}
\]
Abella also encodes relational specifications but does this in a way that we can *reason* about them

- Relations are encoded through clauses of the form:

\[ \forall \vec{X}. H(\vec{X}) \triangleq B(\vec{X}) \]

\[
\text{append } \text{nil } L L \triangleq \top;
\]

\[
\text{append } (X :: L_1) L_2 (X :: L_3) \triangleq \text{append } L_1 L_2 L_3
\]

- Such definitions get a *fixed-point* interpretation, allowing for case analysis based reasoning
Abella also encodes relational specifications but does this in a way that we can *reason* about them.

- Relations are encoded through clauses of the form:
  \[ \forall \vec{X}. H(\vec{X}) \triangleq B(\vec{X}) \]
  
  \[
  \begin{align*}
  \text{append nil} \ L \ L & \triangleq \top; \\
  \text{append} \ (X : : L_1) \ L_2 \ (X : : L_3) & \triangleq \text{append} \ L_1 \ L_2 \ L_3
  \end{align*}
  \]

- Such definitions get a *fixed-point* interpretation, allowing for case analysis based reasoning.
  \[ \forall L_1, L_2, \text{append nil} \ L_1 \ L_2 \supset L_1 = L_2 \]
Abella also encodes relational specifications but does this in a way that we can *reason* about them.

- Relations are encoded through clauses of the form:

  $$\forall \vec{X}. H(\vec{X}) \triangleq B(\vec{X})$$

  $\text{append } \text{nil } L \ L \triangleq \top$

  $\text{append } (X :: L_1) \ L_2 \ (X :: L_3) \triangleq \text{append } L_1 \ L_2 \ L_3$

- Such definitions get a *fixed-point* interpretation, allowing for case analysis based reasoning

  $$\forall L_1 \ L_2, \text{append } \text{nil } L_1 \ L_2 \supset L_1 = L_2$$

- In fact, definitions can be given a *least* fixed-point interpretation, leading to inductive reasoning
Abella also encodes relational specifications but does this in a way that we can *reason* about them.

- Relations are encoded through clauses of the form:
  \[ \forall \vec{X}. H(\vec{X}) \triangleq B(\vec{X}) \]

  \[
  \text{append } \text{nil } L L \triangleq \top; \\
  \text{append } (X :: L_1) L_2 (X :: L_3) \triangleq \text{append } L_1 L_2 L_3
  \]

- Such definitions get a *fixed-point* interpretation, allowing for case analysis based reasoning

  \[ \forall L_1 L_2, \text{append } \text{nil } L_1 L_2 \supset L_1 = L_2 \]

- In fact, definitions can be given a *least* fixed-point interpretation, leading to inductive reasoning

  \[ \forall L_1 L_2 L_3 L_3', \text{append } L_1 L_2 L_3 \supset \text{append } L_1 L_2 L_3' \supset L_3 = L_3' \]
Abella also encodes relational specifications but does this in a way that we can *reason* about them.

- Relations are encoded through clauses of the form:
  \[ \forall \vec{X}. H(\vec{X}) \triangleq B(\vec{X}) \]
  append nil \[L_1\] \[L_2\] \triangleq \top;
  append \[(X :: L_1) \] \[L_2\] \[(X :: L_3) \triangleq append \ L_1 \ L_2 \ L_3\]

- Such definitions get a *fixed-point* interpretation, allowing for case analysis based reasoning
  \[ \forall L_1 \ L_2, append \ nil \ L_1 \ L_2 \supset L_1 = L_2 \]

- In fact, definitions can be given a *least* fixed-point interpretation, leading to inductive reasoning
  \[ \forall L_1 \ L_2 \ L_3 \ L_3', append \ L_1 \ L_2 \ L_3 \supset append \ L_1 \ L_2 \ L_3' \supset L_3 = L_3' \]

- Abella also uses \(\lambda\)-terms for representing objects and has a special quantifier \(\nabla\) for a proof-level treatment of such binders.
The full form of definitional clauses is actually the following:

\[ \forall \vec{X}. (\nabla \vec{z}. H(\vec{X}, \vec{z})) \equiv B(\vec{X}) \]

Such a clause signifies that an instance of \( H \) is true if the corresponding instance of \( B \) is true, provided \( \vec{z} \) is instantiated with distinct, "names" arising from \( \nabla \) quantifiers. \( \vec{X} \) is instantiated with terms not containing these names.

A classic use of this definitional form is to realize substitution for free variables in terms that are represented by \( \nabla \) quantified names. For example:

\[ \text{app subst } \text{nil } \text{M } \text{M} \equiv \top \]
\[ \nabla x, \text{app subst } ((\text{map } x \text{ V}) :: \text{ML}) \text{ (R x) M} \equiv \text{app subst } \text{ML} \text{ (R V) M.} \]

Here, the "pattern" \((R x)\) is used to bind \( R \) to the term with \( x \) abstracted out and applying \( R \) to \( V \) then realizes the substitution.
The full form of definitional clauses is actually the following

\[ \forall \vec{X}. (\bigvee \vec{z}. H(\vec{X}, \vec{z})) \triangleq B(\vec{X}) \]
The full form of definitional clauses is actually the following

$$\forall \vec{X}. (\bigtriangledown \vec{\bar{z}} \cdot H(\vec{\bar{X}}, \vec{\bar{z}})) \triangleq B(\vec{\bar{X}})$$

Such a clause signifies that an instance of $H$ is true if the corresponding instance of $B$ is true, provided

- $\vec{\bar{z}}$ is instantiated with distinct, “names” arising from $\bigtriangledown$ quantifiers
- $\vec{\bar{X}}$ is instantiated with terms not containing these names
Characterizing Variable Occurrences in Terms

The full form of definitional clauses is actually the following

$$\forall \vec{X}. (\forall \vec{z}. H(\vec{X}, \vec{z})) \equiv B(\vec{X})$$

Such a clause signifies that an instance of $H$ is true if the corresponding instance of $B$ is true, provided

- $\vec{z}$ is instantiated with distinct, “names” arising from $\forall$ quantifiers
- $\vec{X}$ is instantiated with terms not containing these names

A classic use of this definitional form is to realize substitution for free variables in terms that are represented by $\forall$ quantified names.
Characterizing Variable Occurrences in Terms

The full form of definitional clauses is actually the following

$$\forall \vec{X}. (\nabla \vec{z}. H(\vec{X}, \vec{z})) \triangleq B(\vec{X})$$

Such a clause signifies that an instance of $H$ is true if the corresponding instance of $B$ is true, provided

- $\vec{z}$ is instantiated with distinct, “names” arising from $\nabla$ quantifiers
- $\vec{X}$ is instantiated with terms not containing these names

A classic use of this definitional form is to realize substitution for free variables in terms that are represented by $\nabla$ quantified names

```plaintext
app_subst nil M M \triangleq \top;
\nabla x, app_subst ((map x V) :: ML) (R x) M \triangleq
app_subst ML (R V) M.
```
The full form of definitional clauses is actually the following

\[ \forall \vec{X}. (\nabla \vec{Z}. H(\vec{X}, \vec{Z})) \triangleq B(\vec{X}) \]

Such a clause signifies that an instance of \( H \) is true if the corresponding instance of \( B \) is true, provided

- \( \vec{Z} \) is instantiated with distinct, “names” arising from \( \nabla \) quantifiers
- \( \vec{X} \) is instantiated with terms not containing these names

A classic use of this definitional form is to realize substitution for free variables in terms that are represented by \( \nabla \) quantified names

\[
\begin{align*}
app\_subst\ nil\ M\ M & \triangleq \top; \\
\nabla\ x,\ app\_subst\ ((map\ x\ V) :: ML)\ (R\ x)\ M & \triangleq \\
app\_subst\ ML\ (R\ V)\ M.
\end{align*}
\]

Here, the “pattern” \( (R\ x) \) is used to bind \( R \) to the term with \( x \) abstracted out and applying \( R \) to \( V \) then realizes the substitution
Abella supports this possibility via the *two-level logic approach*:
Abella supports this possibility via the *two-level logic approach*:

- The entire specification logic is itself encoded into Abella
  - The judgment $L \vdash G$ is represented by the Abella relation $\{L \vdash G\}$
  - The derivation rules are captured in a definition of $\{-\}$

Specifications in $\lambda$Prolog are introduced into Abella as a parameter of the definition of $\{-\}$

Finally, theorems about $\lambda$Prolog specifications become theorems about specific $\{-\}$ predicates

For example, the preservation of types by evaluation is stated as follows:

$$\forall M, T, V, \{\vdash M \to T\} \supset \{\vdash \text{eval}\ M \to V\} \supset \{\vdash \text{of}\ V \to T\}$$

This approach also allows us to exploit the meta-theory of the specification logic in reasoning and to capture informal styles of proof.
Abella supports this possibility via the *two-level logic approach*:

- The entire specification logic is itself encoded into Abella
  - The judgment $L \vdash G$ is represented by the Abella relation \{ $L \vdash G$ \}
  - The derivation rules are captured in a definition of \{ $-$ \}

- Specifications in $\lambda$Prolog are introduced into Abella as a parameter of the definition of \{ $-$ \}

For example, the preservation of types by evaluation is stated as follows:

\[ \forall M T V, \{\vdash \text{of} \ M \ T\} \supset \{\vdash \text{eval} \ M \ V\} \supset \{\vdash \text{of} \ V \ T\} \]
Abella supports this possibility via the \textit{two-level logic approach}:

- The entire specification logic is itself encoded into Abella
  - The judgment $L \vdash G$ is represented by the Abella relation $\{L \vdash G\}$
  - The derivation rules are captured in a definition of $\{\_\}$

- Specifications in $\lambda$Prolog are introduced into Abella as a parameter of the definition of $\{\_\}$

- Finally, theorems about $\lambda$Prolog specifications become theorems about specific $\{\_\}$ predicates
Abella supports this possibility via the *two-level logic approach*:

- The entire specification logic is itself encoded into Abella
  - The judgment $L \vdash G$ is represented by the Abella relation $\{L \vdash G\}$
  - The derivation rules are captured in a definition of $\{-\}$

- Specifications in $\lambda$Prolog are introduced into Abella as a parameter of the definition of $\{-\}$

- Finally, theorems about $\lambda$Prolog specifications become theorems about specific $\{-\}$ predicates

For example, the preservation of types by evaluation is stated as follows:

$$\forall M T V, \{\vdash \text{of } M T\} \supset \{\vdash \text{eval } M V\} \supset \{\vdash \text{of } V T\}$$
Abella supports this possibility via the *two-level logic approach*:

- The entire specification logic is itself encoded into Abella
  - The judgment \( L \vdash G \) is represented by the Abella relation \( \{ L \vdash G \} \)
  - The derivation rules are captured in a definition of \( \{ \_ \} \)

- Specifications in \( \lambda \text{Prolog} \) are introduced into Abella as a parameter of the definition of \( \{ \_ \} \)

- Finally, theorems about \( \lambda \text{Prolog} \) specifications become theorems about specific \( \{ \_ \} \) predicates

For example, the preservation of types by evaluation is stated as follows:

\[
\forall M T V, \{ \vdash \text{of} \ M \ T \} \supset \{ \vdash \text{eval} \ M \ V \} \supset \{ \vdash \text{of} \ V \ T \}
\]

This approach also allows us to exploit the meta-theory of the specification logic in reasoning and to capture informal styles of proof.
Equivalence between closed values in the source and target languages can be defined in a logical relation style:

- Values of atomic types are equivalent if they are identical.
- Values of function types are equivalent if they yield equivalent results given equivalent arguments.

Extended to arbitrary closed terms via evaluation. All this can be formalized in Abella by the definition of $\text{sim } T M M'$.

Actually, to state the correctness of closure conversion, what we need is equivalence between programs containing free variables. Such an equivalence can be based on equivalence of closed terms under equivalent closed substitutions. As seen with app_subst, substitutions and their equivalence can be formalized in a simple, logical way in Abella.
Equivalence between closed values in the source and target languages can be defined in a logical relation style:

- Values of atomic types are equivalent if they are identical
- Values of function types are equivalent if they yield equivalent results given equivalent arguments
Equivalence between closed values in the source and target languages can be defined in a logical relation style:

- Values of atomic types are equivalent if they are identical.
- Values of function types are equivalent if they yield equivalent results given equivalent arguments.

Extended to arbitrary closed terms via evaluation.
Equivalence between closed values in the source and target languages can be defined in a logical relation style:

- Values of atomic types are equivalent if they are identical
- Values of function types are equivalent if they yield equivalent results given equivalent arguments

Extended to arbitrary closed terms via evaluation

All this can be formalized in Abella by the definition of $\text{sim} \ T \ M \ M'$
Equivalence between closed values in the source and target languages can be defined in a logical relation style:

- Values of atomic types are equivalent if they are identical
- Values of function types are equivalent if they yield equivalent results given equivalent arguments

Extended to arbitrary closed terms via evaluation

All this can be formalized in Abella by the definition of $\text{sim } T M M'$

Actually, to state the correctness of closure conversion, what we need is equivalence between programs containing free variables
Semantics Preservation for Closure Conversion

Equivalence between closed values in the source and target languages can be defined in a logical relation style:

- Values of atomic types are equivalent if they are identical
- Values of function types are equivalent if they yield equivalent results given equivalent arguments

Extended to arbitrary closed terms via evaluation

All this can be formalized in Abella by the definition of $\text{sim } T M M'$

Actually, to state the correctness of closure conversion, what we need is equivalence between programs containing free variables

Such an equivalence can be based on equivalence of closed terms under equivalent closed substitutions
Equivalence between closed values in the source and target languages can be defined in a logical relation style:

- Values of atomic types are equivalent if they are identical
- Values of function types are equivalent if they yield equivalent results given equivalent arguments

Extended to arbitrary closed terms via evaluation

All this can be formalized in Abella by the definition of \( \text{sim } T M M' \)

Actually, to state the correctness of closure conversion, what we need is equivalence between programs containing free variables

Such an equivalence can be based on equivalence of closed terms under equivalent closed substitutions

As seen with \textit{app}\textsubscript{\textit{subst}}, substitutions and their equivalence can be formalized in a simple, logical way in Abella
The correctness property is as follows:

Assume $M$ is transformed into $M'$ by closure conversion, then under any equivalent and closed substitutions $\delta$ and $\delta'$, $M[\delta]$ is equivalent to $M'[\delta']$. 

This theorem can be proved by induction on $\{cc\ Map\ Vs\ M\ M'\}$. The logical nature of the specification, the meta-level treatment of substitution, etc, all conspire to yield a concise and transparent proof.
The correctness property is as follows:

Assume $M$ is transformed into $M'$ by closure conversion, then under any equivalent and closed substitutions $\delta$ and $\delta'$, $M[\delta]$ is equivalent to $M'[\delta']$.

We can define $\text{subst}_\text{equiv}$ such that $\text{subst}_\text{equiv} \ L \ ML \ ML'$ holds for substitutions $ML$ and $ML'$ equivalent in the typing context $L$. 
The correctness property is as follows:

Assume $M$ is transformed into $M'$ by closure conversion, then under any equivalent and closed substitutions $\delta$ and $\delta'$, $M[\delta]$ is equivalent to $M'[\delta']$.

We can define $\text{subst_equiv}$ such that $\text{subst_equiv} L ML ML'$ holds for substitutions $ML$ and $ML'$ equivalent in the typing context $L$.

Then the correctness theorem becomes the following:

$$\forall L ML ML' \text{ Map } T P P' M M',$$

$$\ldots$$

$$\text{subst_equiv} L ML ML' \sqsupset \{L \vdash \text{of } M T\} \sqsupset \{\text{cc Map Vs } M M'\} \sqsupset$$

$$\text{app_subst} ML M P \sqsupset \text{app_subst'} ML' M' P' \sqsupset \text{sim } T P P'.$$
The correctness property is as follows:

Assume $M$ is transformed into $M'$ by closure conversion, then under any equivalent and closed substitutions $\delta$ and $\delta'$, $M[\delta]$ is equivalent to $M'[\delta']$.

We can define $\text{subst\_equiv}$ such that $\text{subst\_equiv} \ L \ ML \ ML'$ holds for substitutions $ML$ and $ML'$ equivalent in the typing context $L$.

Then the correctness theorem becomes the following:

$$\forall \; L \; ML \; ML' \; \text{Map} \; T \; P \; P' \; M \; M', \ldots$$

$$\text{subst\_equiv} \ L \; ML \; ML' \supset \{L \vdash \text{of} \; M \; T\} \supset \{\text{cc} \; \text{Map} \; \text{Vs} \; M \; M'\} \supset \text{app\_subst} \; ML \; M \; P \supset \text{app\_subst'} \; ML' \; M' \; P' \supset \text{sim} \; T \; P \; P'.$$

This theorem can be proved by induction on $\{\text{cc} \; \text{Map} \; \text{Vs} \; M \; M'\}$.
The correctness property is as follows:

Assume $M$ is transformed into $M'$ by closure conversion, then under any equivalent and closed substitutions $\delta$ and $\delta'$, $M[\delta]$ is equivalent to $M'[\delta']$.

We can define $\text{subst\_equiv}$ such that $\text{subst\_equiv} L ML ML'$ holds for substitutions $ML$ and $ML'$ equivalent in the typing context $L$.

Then the correctness theorem becomes the following:

\[
\forall L ML ML' \text{ Map} T P P' M M', \ldots
\]

\[
\text{subst\_equiv} L ML ML' \supset \{L \vdash \text{of} M T\} \supset \{\text{cc Map Vs M M'}\} \supset \text{app\_subst} ML M P \supset \text{app\_subst'} ML' M' P' \supset \text{sim} T P P'.
\]

This theorem can be proved by induction on $\{\text{cc Map Vs M M'}\}$.

The logical nature of the specification, the meta-level treatment of substitution, etc, all conspire to yield a concise and transparent proof.
In this talk and the paper, we have
In this talk and the paper, we have

- argued for the usefulness of $\lambda$Prolog and Abella in realizing verified compiler transformations

- implemented closure conversion and other transformations in $\lambda$Prolog for a language with recursion

- verified these implementations using semantics preservation based on step-indexed logical relations

Future Work:

- Exploring the effectiveness of our approach when different or deeper notions of correctness are used
- Implementing and verifying compilation of real-world functional languages such as a subset of SML
- Building automation and polymorphism into Abella to further reduce the proof effort
Conclusion and Future Work

In this talk and the paper, we have

- argued for the usefulness of λProlog and Abella in realizing verified compiler transformations
- implemented closure conversion and other transformations in λProlog for a language with recursion
- verified these implementations using semantics preservation based on step-indexed logical relations

Future Work:

- Exploring the effectiveness of our approach when different or deeper notions of correctness are used
- Implementing and verifying compilation of real-world functional languages such as a subset of SML
- Building automation and polymorphism into Abella to further reduce the proof effort