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

Motivation for Verified Compilation

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.

Verified Compilation of Functional Programs

Compilation consists of two phases:

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

Verified Compilation of Functional 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

Verified Compilation of Functional 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

Characteristics of the transformations in the first phase:

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

Verified Compilation of Functional 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

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

An Overview of the Talk

We make the case using a framework comprising the specification language λProlog and the interactive theorem prover Abella

An Overview of the Talk

We make the case using a framework comprising the specification language λProlog and the interactive theorem prover Abella

- We show that λProlog supports a concise, declarative implementation of the transformations
- We show that using Abella we can construct elegant proofs of correctness for the λProlog programs
- We argue that these benefits in fact derive from the underlying support for HOAS and rule-based relational specifications

An Overview of the Talk

We make the case using a framework comprising the specification language λProlog and the interactive theorem prover Abella

- We show that λProlog supports a concise, declarative implementation of the transformations
- We show that using Abella we can construct elegant proofs of correctness for the λProlog 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

The Closure Conversion Transformation

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

For example,

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

is transformed into

```
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,

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

is transformed into

```
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

The Closure Conversion Transformation

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

For example,

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

is transformed into

```
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 Specification Language λProlog

The language is based on logic programming style clauses that transparently encode *rule-based relational specifications*

The Specification Language λProlog

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

$$\frac{}{append\ []\ l\ l} \qquad \frac{append\ l_1\ l_2\ l_3}{append\ (x::l_1)\ l_2\ (x::l_3)}$$

The Specification Language λProlog

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

$$\frac{}{\mathit{append}\ []\ l\ l} \qquad \frac{\mathit{append}\ l_1\ l_2\ l_3}{\mathit{append}\ (x :: l_1)\ l_2\ (x :: l_3)}$$

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

```
append nil L L.
append (X :: L1) L2 (X :: L3) :- append L1 L2 L3.
```

The Specification Language λProlog

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

$$\frac{}{append\ []\ l\ l} \qquad \frac{append\ l_1\ l_2\ l_3}{append\ (x::l_1)\ l_2\ (x::l_3)}$$

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

```
append nil L L.
append (X :: L1) L2 (X :: L3) :- append L1 L2 L3.
```

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

The Specification Language λProlog

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

$$\frac{}{append\ []\ l\ l} \qquad \frac{append\ l_1\ l_2\ l_3}{append\ (x::l_1)\ l_2\ (x::l_3)}$$

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

```
append nil L L.
append (X :: L1) L2 (X :: L3) :- append L1 L2 L3.
```

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.

Treating Binding Structure in λProlog

A higher-order treatment of abstract syntax is supported in λProlog through the following devices:

Treating Binding Structure in λProlog

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 λProlog

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*

Treating Binding Structure in λProlog

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*

```
abs : (tm -> tm) -> tm    app : tm -> tm -> tm
```

$(\lambda x.\lambda y.x\ y) \implies$ `abs (x\ abs (y\ app x y))`

Treating Binding Structure in λProlog

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*

  ```
  abs : (tm -> tm) -> tm    app : tm -> tm -> tm
  ```
  $(\lambda x.\lambda y.x\ y)$ $\Longrightarrow$ `abs (x\ abs (y\ app x y))`

- Capturing substitution related notions through β-conversion

Treating Binding Structure in λProlog

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*

```
abs : (tm -> tm) -> tm    app : tm -> tm -> tm
```

$(\lambda x.\lambda y.x\ y) \implies$ `abs (x\ abs (y\ app x y))`

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

Treating Binding Structure in λProlog

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*

  ```
  abs : (tm -> tm) -> tm    app : tm -> tm -> tm
  ```
  $(\lambda x.\lambda y.x\ y) \Longrightarrow$ `abs (x\ abs (y\ 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

Treating Binding Structure in λProlog

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*

  ```
  abs : (tm -> tm) -> tm    app : tm -> tm -> tm
  ```

  $(\lambda x.\lambda y.x\ y) \Longrightarrow$ `abs (x\ abs (y\ 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

Treating Binding Structure in λProlog

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*

```
abs : (tm -> tm) -> tm     app : tm -> tm -> tm
```

$(\lambda x.\lambda y.x\ y) \implies$ `abs (x\ abs (y\ 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

$$\frac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x.t : \alpha \to \beta} \quad x \notin dom(\Gamma) \quad \Rightarrow$$

```
of (abs T) (arr Ty1 Ty2) :-
  pi x\
   of x Ty1 => of (T x) Ty2.
```

Rule-Based Specification of Closure Conversion

The transformation is parameterized by a mapping ρ of (source language) free variables to target language expressions

Rule-Based Specification of Closure Conversion

The transformation is parameterized by a mapping ρ of (source language) free variables to target language expressions

We represent the closure conversion judgment as follows:

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

Rule-Based Specification of Closure Conversion

The transformation is parameterized by a mapping ρ 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

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

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

Rule-Based Specification of Closure Conversion

The transformation is parameterized by a mapping ρ 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

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

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

Computing free variables in the abstraction

Rule-Based Specification of Closure Conversion

The transformation is parameterized by a mapping ρ 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

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

where $\rho' = [x \to y, x_1 \to \pi_1(x_e), \ldots, x_n \to \pi_n(x_e)]$ 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 ρ 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

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

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

Creating a mapping from free variables to projections to the environment

Rule-Based Specification of Closure Conversion

The transformation is parameterized by a mapping ρ of (source language) free variables to target language expressions

We represent the closure conversion judgment as follows:

$$\rho \rhd M \leadsto M'$$

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

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

where $\rho' = [x \to y, x_1 \to \pi_1(x_e), ..., x_n \to \pi_n(x_e)]$ 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*

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*

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

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*

The difficulty: *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 *logical* way in λProlog:

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

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*

The difficulty: *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 *logical* way in λProlog:

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

Some clauses in the definition of *fvars* that illustrate these ideas

```
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.
...
```

Creating Maps and Reifying the Environment

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

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.

- *mapenv Map FVs Env* holds if *Env* is the reified environment for *FVs* based on *Map*
- *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.

- *mapenv Map FVs Env* holds if *Env* is the reified environment for *FVs* based on *Map*
- *mapvar FVs E Map* holds if *Map* is the projection map on *E* for the variables in *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 *mapvar* and *mapenv* s.t.

- *mapenv Map FVs Env* holds if *Env* is the reified environment for *FVs* based on *Map*
- *mapvar FVs E Map* holds if *Map* is the projection map on *E* 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 `(map X T)` encoding the mapping of the variable `X` to the term `T`

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

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:

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\ _________))))
         __)
```

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\ ________))))
         __) :-
   (
        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,
                                                              ).
```

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\ 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
   (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) ).
```

Note how the side conditions relating to names and all other aspects of the rule are given a logical treatment

The Theorem Prover Abella

Abella also encodes relational specifications but does this in a way that we can *reason* about them

The Theorem Prover Abella

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:

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

The Theorem Prover Abella

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:

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

$\texttt{append}\ \texttt{nil}\ L\ L \triangleq \top;$

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

The Theorem Prover Abella

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:

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

$$\texttt{append}\ \texttt{nil}\ L\ L \triangleq \top;$$

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

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

The Theorem Prover Abella

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:

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

$$\texttt{append}\ \texttt{nil}\ L\ L \triangleq \top;$$

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

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

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

The Theorem Prover Abella

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:

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

$$\texttt{append nil } L\ L \triangleq \top;$$
$$\texttt{append } (X :: L_1)\ L_2\ (X :: L_3) \triangleq \texttt{append } L_1\ L_2\ L_3$$

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

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

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

The Theorem Prover Abella

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:

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

$$\texttt{append}\ \texttt{nil}\ L\ L \triangleq \top;$$

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

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

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

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

$$\forall L_1\ L_2\ L_3\ L_3', \texttt{append}\ L_1\ L_2\ L_3 \supset \texttt{append}\ L_1\ L_2\ L_3' \supset L_3 = L_3'$$

The Theorem Prover Abella

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:

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

$$\texttt{append nil}\ L\ L \triangleq \top;$$

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

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

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

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

$$\forall L_1\ L_2\ L_3\ L_3', \texttt{append}\ L_1\ L_2\ L_3 \supset \texttt{append}\ L_1\ L_2\ L_3' \supset L_3 = L_3'$$

- Abella also uses λ-terms for representing objects and has a special quantifier ∇ for a proof-level treatment of such binders

Characterizing Variable Occurrences in Terms

Characterizing Variable Occurrences in Terms

The full form of definitional clauses is actually the following

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

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 ∇ quantifiers
- $\vec{X}$ is instantiated with terms not containing these names

Characterizing Variable Occurrences in Terms

The full form of definitional clauses is actually the following

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

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 ∇ 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 ∇ 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 ∇ 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 ∇ quantified names

```
app_subst nil M M ≜ ⊤;
∇ x, app_subst ((map x V) :: ML) (R x) M ≜
  app_subst ML (R V) M.
```

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 ∇ 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 ∇ quantified names

```
app_subst nil M M ≜ ⊤;
∇ x, app_subst ((map x V) :: ML) (R x) M ≜
  app_subst ML (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

Reasoning About λProlog Programs Using Abella

Abella supports this possibility via the *two-level logic approach*:

Reasoning About λProlog Programs Using Abella

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 $\{-\}$

Reasoning About λProlog Programs Using Abella

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 λProlog are introduced into Abella as a parameter of the definition of $\{-\}$

Reasoning About λProlog Programs Using Abella

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 λProlog are introduced into Abella as a parameter of the definition of $\{-\}$
- Finally, theorems about λProlog specifications become theorems about specific $\{-\}$ predicates

Reasoning About λProlog Programs Using Abella

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 λProlog are introduced into Abella as a parameter of the definition of $\{-\}$
- Finally, theorems about λProlog specifications become theorems about specific $\{-\}$ predicates

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

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

Reasoning About λProlog Programs Using Abella

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 λProlog are introduced into Abella as a parameter of the definition of $\{-\}$
- Finally, theorems about λProlog specifications become theorems about specific $\{-\}$ predicates

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

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

This approach also allows us to exploit the meta-theory of the specification logic in reasoning and to capture informal styles of proof

Semantics Preservation for Closure Conversion

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

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

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

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 *sim T M M'*

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 *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 *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

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 *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

Formalizing the Correctness of Closure Conversion

The correctness property is as follows:

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

Formalizing the Correctness of Closure Conversion

The correctness property is as follows:

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

We can define `subst_equiv` such that `subst_equiv L ML ML'` holds for substitutions `ML` and `ML'` equivalent in the typing context `L`

Formalizing the Correctness of Closure Conversion

The correctness property is as follows:

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

We can define `subst_equiv` such that `subst_equiv L ML ML'` holds for substitutions `ML` and `ML'` equivalent in the typing context `L`

Then the correctness theorem becomes the following:

```
∀ L ML ML' Map T P P' M M',
  ...
  subst_equiv L ML ML' ⊃ {L ⊢ of M T} ⊃ {cc Map Vs M M'} ⊃
  app_subst ML M P ⊃ app_subst' ML' M' P' ⊃ sim T P P'.
```

Formalizing the Correctness of Closure Conversion

The correctness property is as follows:

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

We can define `subst_equiv` such that `subst_equiv L ML ML'` holds for substitutions `ML` and `ML'` equivalent in the typing context `L`

Then the correctness theorem becomes the following:

```
∀ L ML ML' Map T P P' M M',
  ...
  subst_equiv L ML ML' ⊃ {L ⊢ of M T} ⊃ {cc Map Vs M M'} ⊃
  app_subst ML M P ⊃ app_subst' ML' M' P' ⊃ sim T P P'.
```

This theorem can be proved by induction on `{cc Map Vs M M'}`

Formalizing the Correctness of Closure Conversion

The correctness property is as follows:

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

We can define `subst_equiv` such that `subst_equiv L ML ML'` holds for substitutions `ML` and `ML'` equivalent in the typing context `L`

Then the correctness theorem becomes the following:

```
∀ L ML ML' Map T P P' M M',
  ...
  subst_equiv L ML ML' ⊃ {L ⊢ of M T} ⊃ {cc Map Vs M M'} ⊃
  app_subst ML M P ⊃ app_subst' ML' M' P' ⊃ sim T P P'.
```

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

Conclusion and Future Work

In this talk and the paper, we have

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

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