Extensible Proof Engineering in Intensional Type Theory

Gregory Malecha
gmalecha@cs.harvard.edu

PhD Defense
Harvard SEAS

February 2, 2015
## Mechanized Reasoning Tools

<table>
<thead>
<tr>
<th>Mathematics</th>
<th>Software</th>
</tr>
</thead>
</table>

Theorems

How can we build these?
Mechanized Reasoning Tools

Mathematics

Proofs

Software

Motivation | Extensible Proof Engineering in Intensional Type Theory
Mechanized Reasoning Tools

Mathematics

Proofs

Software

Compilers

Optimizations

Motivation | Extensible Proof Engineering in Intensional Type Theory
Mechanized Reasoning Tools

Mathematics

Proofs

Domains

Heuristics

Software

Compilers

Analyzers

Motivation | Extensible Proof Engineering in Intensional Type Theory
Mechanized Reasoning Tools

Mathematics
- Proofs

Software
- Compilers
- Analyzers
- Verifiers

Theorems
- Heuristics
- Invariants

Motivation | Extensible Proof Engineering in Intensional Type Theory
Mechanized Reasoning Tools

Mathematics
Proofs

Software
Compilers
Analyzers
Verifiers

Theorems
Heuristics
Invariants

How can we build these?
Mechanized Reasoning Tools

Mathematics

Proofs

Software

Compilers

Analyzers

Verifiers

Theorems

Heuristics

Invariants

How can we build these?
Open computational reflection in intensional type theories can lower the cost of writing automation that is simultaneously trustworthy, scalable, composable, and customizable.
Open computational reflection in intensional type theories can lower the cost of writing automation that is simultaneously trustworthy, scalable, composable, and customizable.
Open computational reflection in intensional type theories can lower the cost of writing automation that is simultaneously trustworthy, scalable, composable, and customizable.
Open computational reflection in intensional type theories can lower the cost of writing automation that is simultaneously trustworthy, scalable, \textbf{composable}, and customizable.
Open computational reflection in intensional type theories can lower the cost of writing automation that is simultaneously trustworthy, scalable, composable, and customizable.
Open computational reflection in intensional type theories can lower the cost of writing automation that is simultaneously trustworthy, scalable, composable, and customizable.
Trustworthiness from a Logic
Trustworthiness from a Logic

Foundational

- ACL2
- Isabelle
- HOL
- Coq
- Agda
- Andromeda
- LF/ELF/TWELF
- Nuprl
- Small kernel (DeBruijn criterion)

The Foundations | Extensible Proof Engineering in Intensional Type Theory
Foundational Proofs for Simple Entailments

\[
\begin{align*}
A \land (B \land C) & \vdash A \land (B \land C) \\
A \land (B \land C) & \vdash (A \land B) \land C \\
A \land (B \land C) & \vdash C \land (A \land B) \\
A \land (B \land C) & \vdash C \land (B \land A)
\end{align*}
\]

\text{Proof tree}

\text{Foundational proofs require that we make all steps explicit.}

\text{Ltac Automation}

\text{Ltac my_tauto := \text{repeat first \[ reflexivity | apply \land-ASSOC | apply \land-COMM | ... \].}

\text{The Foundations} | Extensible Proof Engineering in Intensional Type Theory
Foundational Proofs for Simple Entailments

\[
\frac{A \land (B \land C) \vdash A \land (B \land C)}{
A \land (B \land C) \vdash (A \land B) \land C} \quad \land\text{-ASSOC}
\]
\[
\frac{A \land (B \land C) \vdash C \land (A \land B)}{
A \land (B \land C) \vdash C \land (B \land A)} \quad \land\text{-COMM}
\]

Proof tree

Foundational proofs require that we make all steps explicit.
Foundational Proofs for Simple Entailments

\[
\begin{align*}
A \land (B \land C) & \vdash A \land (B \land C) \\
A \land (B \land C) & \vdash (A \land B) \land C \\
A \land (B \land C) & \vdash C \land (A \land B) \\
A \land (B \land C) & \vdash C \land (B \land A)
\end{align*}
\]

\(-\text{ASSOC} \quad \text{^-\text{ASSOC}}
\quad \text{^-\text{COMM}} \\
\text{^-\text{COMM}}

Foundational proofs require that we make all steps explicit.

Ltac Automation

\[
\text{Ltac my_tauto := repeat first [ reflexivity } \\
\mid \text{ apply ^-Comm } \\
\mid \text{ apply ^-Assoc } \\
\mid \ldots \] .
\]
Foundational Proofs for Simple Entailments

\[
\frac{A \land (B \land C)}{A \land (B \land C)} & \vdash A \land (B \land C) \\
\frac{A \land (B \land C)}{(A \land B) \land C} & \vdash (A \land B) \land C \\
\frac{A \land (B \land C)}{C \land (A \land B)} & \vdash C \land (A \land B) \\
\frac{A \land (B \land C)}{C \land (B \land A)} & \vdash C \land (B \land A)
\]

- \text{Assoc}
- \text{Comm}
- \text{Comm}

Foundational proofs require that we make all steps explicit.

\text{Ltac Automation}

\begin{verbatim}
Ltac my_tauto :=
  repeat first [ reflexivity |
  apply \text{Comm} |
  apply \text{Assoc} |
  ... ].
\end{verbatim}
Foundational Proofs for Simple Entailments

Kernel cannot use custom algorithms!

\[
\begin{align*}
A \land (B \land C) & \vdash A \land (B \land C) & \land\text{-Assoc} \\
A \land (B \land C) & \vdash (A \land B) \land C & \land\text{-Comm} \\
A \land (B \land C) & \vdash C \land (A \land B) & \land\text{-Comm} \\
A \land (B \land C) & \vdash C \land (B \land A) & \land\text{-Comm}
\end{align*}
\]

Still have to build & check the proof

builds

Still have to build & check the proof

Foundational proofs require that we make all steps explicit.

\texttt{Ltac my\_tauto := repeat first} [ \texttt{reflexivity} \\
| \texttt{apply \, \land\text{-Comm}} \\
| \texttt{apply \, \land\text{-Assoc}} \\
| \texttt{... \,}.]
Trustworthiness from a Logic

Foundational

HOL

Agda

Andromeda

LF/ELF/TWELF

Small kernel
(DeBruijn criterion)
Trustworthiness from a Logic

<table>
<thead>
<tr>
<th>Foundational</th>
<th>Intensional</th>
<th>Extensional</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOL</td>
<td>Coq</td>
<td>Nuprl</td>
</tr>
<tr>
<td>Agda</td>
<td>Andromeda</td>
<td></td>
</tr>
<tr>
<td>LF/ELF/TWELF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Small kernel (DeBruijn criterion)
Computation in Logic/Type Theory

\[ P \equiv Q \quad \vdash Q \quad \vdash P \]

\text{Meta-logic equality}

\^ Abbreviated from the actual type theory rule.
Computation in Logic/Type Theory

Ext. Type Theory

\[ \vdash P = Q \quad P \equiv Q \]

Int. Type Theory

\[ \vdash Q \]

\[ \vdash P \]

\[ \text{CONV}^\dagger \]

\[ \dagger \text{ Abbreviated from the actual type theory rule.} \]
Computation in Logic/Type Theory

Ext. Type Theory

\[ \vdash P = Q \]

Int. Type Theory

\[ P \equiv Q \]

Execute the term

\[ \vdash Q \]

\[ \vdash P \]

\[ \text{CONV}^\dagger \]

\[ \dagger \text{ Abbreviated from the actual type theory rule.} \]
Trustworthiness from a Logic

<table>
<thead>
<tr>
<th>Foundational</th>
<th>Decidable</th>
<th>Undecidable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL2</td>
<td>HOL</td>
<td>Nuprl</td>
</tr>
<tr>
<td>Isabelle</td>
<td>Agda</td>
<td>LF/ELF/TWELF</td>
</tr>
<tr>
<td>HOL</td>
<td>Andromeda</td>
<td>Small kernel</td>
</tr>
<tr>
<td>Agda</td>
<td></td>
<td>(DeBruijn criterion)</td>
</tr>
</tbody>
</table>
Trustworthiness from a Logic

<table>
<thead>
<tr>
<th>Simple(r) Types</th>
<th>Dependent Types</th>
<th>Undecidable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foundational</td>
<td>Decidable</td>
<td></td>
</tr>
<tr>
<td>ACL2</td>
<td>HOL</td>
<td>Coq</td>
</tr>
<tr>
<td>Isabelle</td>
<td>Agda</td>
<td>Andromeda</td>
</tr>
<tr>
<td>Nuprl</td>
<td>LF/ELF/TWELF</td>
<td></td>
</tr>
</tbody>
</table>

Computational

Use this to compress proofs

Small kernel (DeBruijn criterion)
### Trustworthiness from a Logic

<table>
<thead>
<tr>
<th>Simple(r) Types</th>
<th>Dependent Types</th>
<th>Undecidable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foundational</td>
<td>Decidable</td>
<td></td>
</tr>
</tbody>
</table>

- **Simple(r) Types Foundational**
  - ACL2
  - Isabelle
  - HOL
  - Computational

- **Dependent Types Decidable**
  - Coq
  - Agda
  - Andromeda

- **Undecidable**
  - Nuprl
  - LF/ELF/TWELF
  - Small kernel (DeBruijn criterion)

*Use this to compress proofs*
Computation in Logic/Type Theory

Ext. Type Theory

\[ \vdash P = Q \]

\[ P \equiv Q \]

\[ \vdash Q \]

Int. Type Theory

\[ P \rightsquigarrow^* Q \]

\[ \vdash P \]

\[ \text{Many steps!} \]

\[ \text{CONV}^† \]

† Abbreviated from the actual type theory rule.
Computational Reflection [Bou97]

\[
\begin{align*}
A \land (B \land C) & \vdash A \land (B \land C) \\
A \land (B \land C) & \vdash (A \land B) \land C \\
A \land (B \land C) & \vdash C \land (A \land B) \\
A \land (B \land C) & \vdash C \land (B \land A)
\end{align*}
\]
Computational Reflection [Bou97]

Syntactic  Semantic

\[
\begin{align*}
&A \land (B \land C) \vdash C \land (B \land A) \\
\end{align*}
\]

\[
\begin{align*}
&A \land (B \land C) \vdash A \land (B \land C) \\
&A \land (B \land C) \vdash (A \land B) \land C \\
&A \land (B \land C) \vdash C \land (A \land B) \\
&A \land (B \land C) \vdash C \land (B \land A)
\end{align*}
\]
Computational Reflection [Bou97]

\[
\text{true} = \text{true}
\]

\[
\text{rtauto}(A \land (B \land C) \vdash C \land (B \land A)) = \text{true}
\]

\[
\left[ A \land (B \land C) \vdash C \land (B \land A) \right]_{\text{Prop}}
\]

Function

Soundness proof

\[
\text{Thm } \text{rtauto\_sound} : \forall g, \text{ rtauto } g = \text{true} \rightarrow \left[ g \right]_{\text{Prop}}.
\]

Proof ... Qed.
Computational Reflection [Bou97]

Thm rtauto_sound : \( \forall g, \)  
\[ \text{rtauto} \, g = \text{true} \rightarrow [g]_{\text{Prop}}. \]  
Proof. ... Qed.

Syntactic Semantic

Small proof, custom algorithm

Large proof

Function

true = true

rtauto(\( A \land (B \land C) \vdash C \land (B \land A) \)) = true

\[ [A \land (B \land C) \vdash C \land (B \land A)]_{\text{Prop}} \]

Soundness proof

\[ A \land (B \land C) \vdash (A \land B) \land C \]

\[ A \land (B \land C) \vdash C \land (A \land B) \]

\[ A \land (B \land C) \vdash C \land (B \land A) \]
Open **computational reflection** in **intensional type theories** can lower the cost of writing automation that is simultaneously **trustworthy**, **scalable**, composable, and customizable.
Open computational reflection in intensional type theories can lower the cost of writing automation that is simultaneously trustworthy, scalable, composable, and customizable.
<table>
<thead>
<tr>
<th>Logic</th>
<th>Arithmetic</th>
<th>Logic+Arith</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtauto</td>
<td>arith</td>
<td>rtauto_arith</td>
</tr>
</tbody>
</table>
Composing Reflective Procedures

```
Logic
True \land X
```

```
Arithmetic
a + b = b + a
```

```
Logic+Arith
True \land (a + b = b + a)
```

```
\begin{align*}
\text{rtauto} & \quad \text{arith} \\
\otimes & \quad = \\
\text{rtauto_arith} & 
\end{align*}
```
Composing Reflective Procedures

**Logic**
- **rtauto**
  - \( p ::= \text{True} | p_1 \land p_2 \)

**Arithmetic**
- \( a+b=b+a \)
  - \( p ::= a_1 \equiv a_2 \)
  - \( a ::= a_1 + a_2 \)

**Logic+Arith**
- \( \text{True} \land (a+b=b+a) \)
  - \( = \text{rtauto}_\text{arith} \)


Composing Reflective Procedures

Logic

\[ \text{True} \land X \]

\[ \text{rtauto} \]

\[ pr ::= \text{True} | r \land r \]

Arithmetic

\[ a + b = b + a \]

\[ \text{arith} \]

\[ p ::= a_1 = a_2 \]

\[ ar ::= r + r \]

Logic + Arith

\[ True \land (a + b = b + a) \]

\[ \text{rtauto}_\text{arith} \]

Datatypes \textit{\texttt{a.la. carte}} [Swi08]

Metatheory \textit{\texttt{a.la. carte}} [DdSOS13]
Composing Reflective Procedures

Logic

\[ \text{True} \land X \]

\[ \text{rtauto} \]

\[ p ::= \text{True} \mid p_1 \land p_2 \]

Arithmetic

\[ a + b = b + a \]

\[ \text{arith} \]

\[ p ::= a_1 \equiv a_2 \]

\[ a ::= a_1 + a_2 \]

Logic+Arith

\[ \text{True} \land \left( a + b = b + a \right) \]

\[ \text{rtauto}_\text{arith} \]

\[ \chi \land \chi \text{True} \]

\[ (\chi = \chi \left( \chi + \chi a \chi b \right) \chi \left( \chi + \chi b \chi a \right)) \]

Key Insight!

\[ \lambda (X) \]
Composing Reflective Procedures

Logic

Arithmetic

Logic+Arith

\[ \text{True} \land X \]

\[ a + b = b + a \]

\[ \text{True} \land (a + b = b + a) \]

\[ p ::= \text{True} | p_1 \land p_2 \]

\[ p ::= a_1 \equiv a_2 \]

\[ a ::= a_1 \pm a_2 \]

\[ t ::= T \# | t_1 \rightarrow t_2 \]

\[ e ::= X \# | e_1 @ e_2 | \lambda t. e | x_n \]

Define independently

\[ \lambda (X) \]

Semantic

\[ X \land @ X \text{True} @ (X = @ (X + @ a @ b) @ (X + @ b @ a)) \]
Composing Reflective Procedures

Logic

- \( \text{True} \triangleq X \)
- \( \text{rtauto} \)
- \( p ::= \text{True} \mid p_1 \triangleq p_2 \)

Arithmetic

- \( a + b = b + a \)
- \( \text{arith} \)
- \( p ::= a_1 \equiv a_2 \)
- \( a ::= a_1 + a_2 \)

Logic + Arith

- \( \text{True} \triangleq (a + b = b + a) \)
- \( \text{rtauto}_\text{arith} \)
- \( x_\lambda (X) \)
- \( e ::= X_\# \mid e_1 @ e_2 \mid \lambda t. e \mid x_n \)
- \( t ::= T_\# \mid t_1 \rightarrow t_2 \)
- \( (X = @ (X + @ a @ b) @ (X + @ b @ a)) \)

Semantic

Define independently

Soundness theorems

reason about the denotation function
Composing Reflective Automation

\[ \lambda (\land, +, =) \]
Composing Reflective Automation

Two ways to achieve this:
- Explicit equality proofs
- Definitional equality (reduction)

\( \lambda (\land, +, =) \)
Composing Reflective Automation

Two ways to achieve this:

1. Explicit equality proofs
2. Definitional equality (reduction)

\[ \lambda (\wedge, +, =) \]

Must agree on overlap
Composing Reflective Automation

Two ways to achieve this:
- Explicit equality proofs
- Definitional equality (reduction)

Must agree on overlap

A Composable Reflective Core | Extensible Proof Engineering in Intensional Type Theory
Composing Reflective Automation

Two ways to achieve this

- Explicit equality proofs
- Definitional equality (reduction)
Composing Reflective Automation

Two ways to achieve this

- Explicit equality proofs
- Definitional equality (reduction)
Composable Automation

Language Symbols

\[ \begin{align*}
\text{Var} & \quad \text{tyProp} : \text{typ}. \\
\text{Var} & \quad \text{sTr} \; \text{sAnd} : \text{sym}.
\end{align*} \]

Reflective Procedure

\[ \begin{align*}
\text{Def} & \quad \text{rtauto} \ (g : \text{expr}) : \text{bool} := \\
& \quad \text{match} \ g \ \text{with} \\
& \quad \quad | \ X_{sTr} \Rightarrow \text{true} \\
& \quad \quad | \ X_{sAnd} \ @ \ l \ @ \ r \Rightarrow \\
& \quad \quad \quad \quad \quad \text{rtauto} \ l \ \&\& \ \text{rtauto} \ r \\
& \quad \quad | \ _ \Rightarrow \text{false} \\
& \quad \text{end}.
\end{align*} \]

Soundness Proof

\[ \begin{align*}
\text{Thm} \ & \text{rtauto\_sound} \\
& : \ \forall \ g, \ \text{rtauto} \ g = \text{true} \rightarrow \\
& \quad \quad \ [g]_{\text{tyProp}}. \\
\text{Proof.} \ldots \ \text{Qed}.
\end{align*} \]
Composable Automation

Language Symbols

\{ Var tyProp : typ. \\
Var sTr sAnd : sym. \}

Reflective Procedure

\{ Def rtauto (g : expr) : bool := \\
   match g with \\
   | X sTr ⇒ true \\
   | X sAnd @ l @ r ⇒ rtauto l && rtauto r \\
   | _ ⇒ false \\
end. \}

Language Constraints

\{ Var pfP : [ tyProp ] = Prop. \\
Var pfTr : [ sTr ]_tyProp = True. \\
Var pfAnd : [ sAnd ]_... = ∧. \}

Soundness Proof

\{ Thm rtauto_sound \\
   : ∀ g, rtauto g = true → \\
   [ g ]_tyProp. \\
Proof. ... Qed. \}
Composable Automation

\[
\begin{aligned}
\text{Var } & \text{tyProp : typ.} \\
\text{Var } & \text{sTr sAnd : sym.} \\
\text{Def } & \text{rtauto } (g: \text{expr}) : \text{bool} := \\
& \text{match } g \text{ with} \\
& \mid X_{sTr} \Rightarrow \text{true} \\
& \mid X_{sAnd} \circ l \circ r \Rightarrow \\
& \quad \text{rtauto } l \land \text{rtauto } r \\
& \mid _\_ \Rightarrow \text{false} \\
& \text{end.} \\
\end{aligned}
\]

\[
\begin{aligned}
\text{Var } & \text{pfP } : [\text{tyProp}] = \text{Prop.} \\
\text{Var } & \text{pfTr } : [\text{sTr}]_{\text{tyProp}} = \text{True.} \\
\text{Var } & \text{pfAnd } : [\text{sAnd}]_\ldots = \land. \\
\text{Thm } & \text{rtauto_sound : } \forall g, \text{rtauto } g = \text{true} \land \\
& [g]_{\text{tyProp}}. \\
\text{Proof. } \ldots \text{Qed.} \\
\end{aligned}
\]

Type Error!

\[ [\text{tyProp}] \neq \text{Prop} \]
Explicit casts

\[ \text{Ha} : \text{cast}_{pfP} \ A \]
\[ \text{Hb} : \text{cast}_{pfP} \ B \]

\[ \text{cast}_{pfP} (A \land B) \]

\begin{verbatim}
Var tyProp : typ.
Var sTr sAnd : sym.

Def rtauto (g : expr) : bool :=
  match g with
  | X_{sTr} ⇒ true
  | X_{sAnd @ l @ r} ⇒
    rtauto l && rtauto r
  | _ ⇒ false
end.

Var pfP : \[\text{tyProp}\] = Prop.
Var pfTr : \text{cast}_{pfP} [sTr]_{tyProp} = True.
Var pfAnd : \text{cast}_{pfP} [sAnd]... = \land.

Thm rtauto_sound :
  \( \forall g, \text{rtauto } g = true \rightarrow \text{cast}_{pfP} [g]_{tyProp} \)
  Proof.... Qed.
\end{verbatim}
Composable Automation

- Explicit casts

\[ \text{Ha} : \text{cast}_{pfP} A \]
\[ \text{Hb} : \text{cast}_{pfP} B \]

\[ \text{cast}_{pfP} (A \land B) \]

- Composable only when proofs match up exactly

\[ \text{Ha} : \text{cast}_{pfP} A \]

\[ \text{cast}_{pfQ} A \]

✓ Very flexible

✗ Verbose

\[ \text{Var} \ \text{tyProp} : \text{typ}. \]
\[ \text{Var} \ \text{sTr} \ \text{sAnd} : \text{sym}. \]

\[ \text{Def} \ rtauto (g : \text{expr}) : \text{bool} := \]
\[ \text{match} \ g \ \text{with} \]
\[ \ | \ X_{sTr} \Rightarrow \text{true} \]
\[ \ | \ X_{sAnd} \circ l \circ r \Rightarrow \]
\[ rtauto \ l \&\& rtauto \ r \]
\[ \ | \ _ \Rightarrow \text{false} \]
\[ \text{end.} \]

\[ \text{Var} \ \text{pfP} : [\ \text{tyProp} ] = \text{Prop}. \]
\[ \text{Var} \ \text{pfTr} : \text{cast}_{pfP} [\text{sTr}]_{\text{tyProp}} = \text{True}. \]
\[ \text{Var} \ \text{pfAnd} : \text{cast}_{pfP} [\text{sAnd}]... = \land. \]

\[ \text{Thm} \ rtauto\_\text{sound} \]
\[ : \forall g, \ rtauto \ g = \text{true} \rightarrow \]
\[ \text{cast}_{pfP} [g]_{\text{tyProp}}. \]

\[ \text{Proof.} \ ... \ \text{Qed.} \]
Composing Reflective Automation

Two ways to achieve this

- Explicit equality proofs
- **Definitional equality (reduction)**

Must agree on overlap
Let \( \text{tyProp} := T_0 \). (* typ *)
Let \( \text{sTr} := X_0 \).
Let \( \text{sAnd} := X_1 \).

Def \( \text{rtauto} \ (g : \text{expr}) : \text{bool} := \)

match \( g \) with
| \( X_{\text{sTr}} \) \( \Rightarrow \) true
| \( X_{\text{sAnd}} @ 1 @ r \Rightarrow \)
  \( \text{rtauto} \ l \ & \ & \text{rtauto} \ r \)
| _ \( \Rightarrow \) false
end.

Var \( \text{ts} : \text{list} \ \text{Type} \).
Var \( \text{fs} : \text{list} \ ... \)

Thm \( \text{rtauto\_sound} \)
: \( \forall g, \text{rtauto} \ g = \text{true} \rightarrow \)
\( \text{ts} \ [g]_{\text{tyProp}} \)
\( \text{fs} \ [g]_{\text{tyProp}} \).
Proof. ... Qed.
Let \( \text{tyProp} := T_0 \). (* typ *)
Let \( \text{sTr} := X_0 \).
Let \( \text{sAnd} := X_1 \).

Def \( \text{rtauto} (g: \text{expr}): \text{bool} := \)
  match \( g \) with
  | \( X_{sTr} \) \( \Rightarrow \) true
  | \( X_{sAnd} \) \( \@ \) l \( \@ \) r \( \Rightarrow \)
    \( \text{rtauto} \) l \&\& \( \text{rtauto} \) r
  | _ \( \Rightarrow \) false
end.

Thm \( \text{rtauto\_sound} \):
\( \forall g, \text{rtauto} g = \text{true} \rightarrow \)
\( ts \) \( fs \) \[ g \] \( \text{tyProp} \).
Proof. ... Qed.

\( \text{Var ts:} \ [\ \tau_0 \ | \ \tau_1 \ | \ \tau_2 \ | \ \ldots \ ] \)
Let $\text{tyProp} := T_0$. (* typ *)
Let $\text{sTr} := X_0$.
Let $\text{sAnd} := X_1$.

Def $\text{rtauto} (g : \text{expr}) : \text{bool} :=$
  match $g$ with
  $| \text{XsTr} \Rightarrow \text{true}$
  $| \text{XsAnd}@l@r \Rightarrow$
    $\text{rtauto } l \&\& \text{rtauto } r$
  $| _\Rightarrow \text{false}$
end.

Var $\text{ts} : [\tau_0 \, | \, \tau_1 \, | \, \tau_2 \, | \, ... ]$

Thm $\text{rtauto\_sound}$
: $\forall g, \text{rtauto } g = \text{true} \rightarrow$
  $\text{ts} \, [\, g \, ]_{\text{tyProp}}$
Proof. ... Qed.

Var $\text{ts} : \text{list Type}$.
Var $\text{fs} : \text{list } ...$

Thm $\text{rtauto\_sound}$
: $\forall g, \text{rtauto } g = \text{true} \rightarrow$
  $\text{ts} \, [\, g \, ]_{\text{tyProp}}$
Proof. ... Qed.

Var $\text{ts} : \text{list Type}$.

[169x251]Composition with Environments

[7x236]Let $\text{tyProp} := T_0$. (* typ *)
Let $\text{sTr} := X_0$.
Let $\text{sAnd} := X_1$.

Def $\text{rtauto} (g : \text{expr}) : \text{bool} :=$
  match $g$ with
  $| \text{XsTr} \Rightarrow \text{true}$
  $| \text{XsAnd}@l@r \Rightarrow$
    $\text{rtauto } l \&\& \text{rtauto } r$
  $| _\Rightarrow \text{false}$
end.

Var $\text{ts} : [\tau_0 \, | \, \tau_1 \, | \, \tau_2 \, | \, ... ]$

Thm $\text{rtauto\_sound}$
: $\forall g, \text{rtauto } g = \text{true} \rightarrow$
  $\text{ts} \, [\, g \, ]_{\text{tyProp}}$
Proof. ... Qed.

Var $\text{ts} : \text{list Type}$.
Var $\text{fs} : \text{list } ...$

Thm $\text{rtauto\_sound}$
: $\forall g, \text{rtauto } g = \text{true} \rightarrow$
  $\text{ts} \, [\, g \, ]_{\text{tyProp}}$
Proof. ... Qed.
Composition with Environments

Let $\text{tyProp} := T_0$. (* typ *)
Let $\text{sTr} := X_0$.
Let $\text{sAnd} := X_1$.

Def $\text{rtauto (g:expr)} : \text{bool} :=$
match $g$ with
| $X_{\text{sTr}}$ ⇒ true
| $X_{\text{sAnd}} \circ l \circ r$ ⇒
  \begin{align*}
  & \text{rtauto } l \&\& \text{rtauto } r \\
  & _{\text{false}}
  \end{align*}
| _ ⇒ false
end.

Var $\text{ts : list Type}$.
Var $\text{fs : list ...}$

Thm $\text{rtauto_sound}$:
$\forall g, \text{rtauto } g = \text{true} \rightarrow$
$\forall ts, \forall fs[\text{g}]_{\text{tyProp}}$
Proof. ... Qed.

[$T_0] \equiv \text{P}
Let \( \text{tyProp} := T_0. \) (* typ *)
Let \( \text{sTr} := X_0. \)
Let \( \text{sAnd} := X_1. \)

Def \( \text{rtauto} (g: \text{expr}): \text{bool} := \) match \( g \) with
| \( X_{\text{sTr}} \) ⇒ true
| \( X_{\text{sAnd}} \circ l \circ r \) ⇒ \( \text{rtauto} l \&\& \text{rtauto} r \)
| _ ⇒ false
end.

Var \( ts: \) list Type.
Var \( fs: \) list ...

Thm \( \text{rtauto\_sound} \)
: \( \forall g, \text{rtauto} g = \text{true} \rightarrow \)
\( \text{ts} \oplus c \llbracket g \rrbracket_{\text{tyProp}}. \)
Proof. ... Qed.
Generic Reflective Automation

- Some tasks are very easy to automate

\[ \vdash x \in (\{x, y\} \cup \{z\}) \]

Proof
Generic Reflective Automation

- Some tasks are very easy to automate

\[
\vdash x \in \{x, y\} \\
\vdash x \in (\{x, y\} \cup \{z\}) \\
\text{LEM2}
\]

Proof
Some tasks are very easy to automate

∀ a B C,
   a ∈ B →
   a ∈ (B ∪ C)

\[ \vdash x \in \{x, y\} \]
\[ \vdash x \in (\{x, y\} \cup \{z\}) \]

Syntax

Proof

\[ \text{LEM2} \]
Some tasks are very easy to automate

\[ \forall aB C, \quad ?a \in ?B \rightarrow ?a \in (?B \cup ?C) \]

What expressions make the conclusion match the goal?

\[ \vdash x \in \{x, y\} \]

\[ \vdash x \in (\{x, y\} \cup \{z\}) \]
Some tasks are very easy to automate

⊢ $x^2 (f x; y g[z g])$

⊢ $x^2 f x; y g$

⊢ $x = x$

⊢ $x \in \{x, y\}$

⊢ $x \in (\{x, y\} \cup \{z\})$

Syntax

Proofs

LEM3

LEM2

Unification

"Hint Database"
Some tasks are very easy to automate

Generic procedures make it easy to quickly build simple automation

\[
\frac{}{\vdash x = x} \quad \text{LEM3}
\]
\[
\frac{}{\vdash x \in \{x, y\}} \quad \text{LEM2}
\]
\[
\frac{}{\vdash x \in (\{x, y\} \cup \{z\})}
\]
Open computational reflection in intensional type theories can lower the cost of writing automation that is simultaneously trustworthy, scalable, *composable*, and *customizable*. 
Open computational reflection in intensional type theories can lower the cost of writing automation that is simultaneously trustworthy, scalable, composable, and customizable.
BEDROCK: Composability, Customizability & Scalability

- BEDROCK [Chl11] is a Coq library for imperative program verification.
- Verified thousands of lines of low-level code!
  - Basic data structures [MCB14]
  - Garbage Collector
  - Thread library and Web server [Chl15]
  - Robot Operating System [Chl15]
**BEDROCK: Composability, Customizability & Scalability**

- **BEDROCK** [Chl11] is a Coq library for imperative program verification.
- Verified thousands of lines of low-level code!
  - Basic data structures [MCB14]
  - Garbage Collector
  - Thread library and Web server [Chl15]
  - Robot Operating System [Chl15]

- Reasonable proof burden.

<table>
<thead>
<tr>
<th>Module</th>
<th>Program</th>
<th>Invar.</th>
<th>Tactics</th>
<th>Other</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedList</td>
<td>42</td>
<td>26</td>
<td>27</td>
<td>31</td>
<td>2.0</td>
</tr>
<tr>
<td>Malloc</td>
<td>43</td>
<td>16</td>
<td>112</td>
<td>94</td>
<td>5.2</td>
</tr>
<tr>
<td>ListSet</td>
<td>50</td>
<td>31</td>
<td>23</td>
<td>46</td>
<td>2.0</td>
</tr>
<tr>
<td>TreeSet</td>
<td>108</td>
<td>40</td>
<td>25</td>
<td>45</td>
<td>1.0</td>
</tr>
<tr>
<td>Queue</td>
<td>53</td>
<td>22</td>
<td>80</td>
<td>93</td>
<td>3.7</td>
</tr>
<tr>
<td>Memoize</td>
<td>26</td>
<td>13</td>
<td>56</td>
<td>50</td>
<td>4.6</td>
</tr>
</tbody>
</table>

"Overhead of verification" < 20x
BEDROCK: Macro Performance

- Does open computational reflection make verification faster? Yes
BEDROCK: Macro Performance

- Does open computational reflection make verification faster? **Yes**
- Does it make verification fast? **Reasonably**
BEDROCK: Macro Performance

- Does open computational reflection make verification faster? **Yes**
- Does it make verification fast? **Reasonably**
BEDROCK: Macro Performance

- Does open computational reflection make verification faster? Yes
- Does it make verification fast? Reasonably
BEDROCK: Macro Performance

- Does open computational reflection make verification faster? Yes
- Does it make verification fast? Reasonably

Evaluation

Extensible Proof Engineering in Intensional Type Theory
BEDROCK: Macro Performance

- Does open computational reflection make verification faster? Yes
- Does it make verification fast? Reasonably
**BEDROCK: Macro Performance**

- Does open computational reflection make verification faster? **Yes**
- Does it make verification fast? **Reasonably**

![Diagram showing the process of VC-gen, HO, Sym Eval, and HO]

---

**Evaluation** | Extensible Proof Engineering in Intensional Type Theory
BEDROCK: Macro Performance

- Does open computational reflection make verification faster? Yes
- Does it make verification fast? Reasonably

Diagram:
- VC-gen
- HO
- Sym Eval
- HO
- Entailment

Evaluation | Extensible Proof Engineering in Intensional Type Theory
**BEDROCK: Macro Performance**

- Does open computational reflection make verification faster? **Yes**
- Does it make verification fast? **Reasonably**

\[ \text{\% Time Spent} \]

- \( \sim 29\% \) reflective automation
- \( \sim 71\% \) \( \text{Ltac} \)

† The division of the 71% is for illustrative purposes only, the results simply states that 71% of the total time is spent in \( \text{Ltac} \).

† The division of the 71% is for illustrative purposes only, the results simply states that 71% of the total time is spent in \( \text{Ltac} \).
**BEDROCK: Customizability & Performance**

- Customizability is essential for good performance.

---

**Linked List Length**

```c
int length(llist* x){
    int n = 0;
    while (x != 0) { // c1
        /* <loop invariant> */
        n = n + 1; // c2
        x = x->next; // c3
    }
    return n;
}
```

---

However, customizability introduces a cost for entering the "reflected" world. Evaluation

| Extensible Proof Engineering in Intensional Type Theory | 21 / 29 |
BEDROCK: Customizability & Performance

- Customizability is essential for good performance.

```
Linked List Length

int length(llist* x) {
    int n = 0;
    while (x ≠ 0) { // c1
        /* <loop invariant> */
        n = n + 1; // c2
        x = x->next; // c3
    }
    return n;
}
```

<table>
<thead>
<tr>
<th>Time (sec)</th>
<th>w/ Custom</th>
<th>w/o Custom</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.30</td>
<td></td>
<td>0.30</td>
</tr>
</tbody>
</table>
BEDROCK: Customizability & Performance

- Customizability is essential for good performance.

```
int length(llist* x) {
    int n = 0;
    while (x != 0) {  // c1
        /* <loop invariant> */
        n = n + 1;  // c2
        x = x->next;  // c3
    }
    return n;
}
```

Linked List Length

**Evaluation** | Extensible Proof Engineering in Intensional Type Theory 21 / 29
**BEDROCK: Customizability & Performance**

- Customizability is essential for good performance.

### Linked List Length

```c
int length(llist* x) {
    int n = 0;
    while (x != 0) {  // c1
        /* <loop invariant> */
        n = n + 1;  // c2
        x = x->next;  // c3
    }
    return n;
}
```

### Evaluation

<table>
<thead>
<tr>
<th>Time (sec)</th>
<th>w/ Custom</th>
<th>w/o Custom</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.30</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>0.58</td>
<td>1.77</td>
</tr>
</tbody>
</table>

- Reflective
- $\mathcal{L}_{ac}$
**BEDROCK: Customizability & Performance**

- Customizability is essential for good performance.

---

**Linked List Length**

```c
int length(llist* x) {
    int n = 0;
    while (x != 0) {  // c1
        /* <loop invariant> */
        n = n + 1;  // c2
        x = x->next;  // c3
    }
    return n;
}
```

---

<table>
<thead>
<tr>
<th>Time (sec)</th>
<th>w/ Custom</th>
<th>w/o Custom</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.38</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>0.89</td>
<td>0.58</td>
</tr>
<tr>
<td></td>
<td>1.77s</td>
<td></td>
</tr>
</tbody>
</table>

**Reflective**

**Łac**
**BEDROCK: Customizability & Performance**

- Customizability is essential for good performance.

```c
int length(llist* x) {
    int n = 0;
    while (x != 0) {  // c1
        /* <loop invariant> */
        n = n + 1;       // c2
        x = x->next;     // c3
    }
    return n;
}
```

**Linked List Length**

<table>
<thead>
<tr>
<th>Time (sec)</th>
<th>w/ Custom</th>
<th>w/o Custom</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/ Custom</td>
<td>0.38</td>
<td>0.30</td>
</tr>
<tr>
<td>w/o Custom</td>
<td>~5x faster</td>
<td>0.89</td>
</tr>
</tbody>
</table>

**5x overall speedup**

- Cost for entering “reflected” world

- Evaluation of Extensible Proof Engineering in Intensional Type Theory
Open computational reflection in intensional type theories can lower the cost of writing automation that is simultaneously trustworthy, scalable, **composable**, and customizable.
A Whole Range of Reflective Procedures

Permutations
Arith
Complex

Lists
Sets
Simple

\textbf{\textit{Rac}}: \textbf{Reflective Building Blocks} | Extensible Proof Engineering in Intensional Type Theory
A Whole Range of Reflective Procedures

- Build a language/library for writing/composing reflective procedures

```plaintext
Fix verify p c :=
  match c with
  | Write p v ⇒
    (* apply write lemma *)
  | Read v e ⇒
    (* apply read lemma *)
  | ...
end.

Fix use_hints hints goal :=
  match hints with
  | [] ⇒ false
  | h :: hs ⇒
    (* apply h and recurse
    * or
    * try the remaining hints
    *)
end.
```

- Combining rich procedures

- Permutations

- Arith

- Complex

- Lists

- Quantifiers & hypotheses

- Sets

- Simple

---

R_tac: Reflective Building Blocks | Extensible Proof Engineering in Intensional Type Theory
A Whole Range of Reflective Procedures

- Build a language/library for writing/composing reflective procedures
- Capture backtracking proof search (similar to $\texttt{L_tac}$)

```haskell
Fix verify p c :=
  match c with
  | Write p v ⇒
    (* apply write lemma *)
  | Read v e ⇒
    (* apply read lemma *)
  | ...
end.

Fix use_hints hints goal :=
  match hints with
  | [] ⇒ false
  | h :: hs ⇒
    (* apply h and recurse or try the remaining hints *)
end.
```

- Combining rich procedures
- Quantifiers & hypotheses

Permutations

Arith

Complex

Lists

Sets

Simple
Program Verification using Combinators

**Ltac Automation**

\[
\text{Ltac verify} := \text{repeat first}
\]

\[
[ \text{eapply step_read};[|\text{side_condition}]
\]

\[
| ... 
\]

\[
| \text{tauto} .
\]
Ltac Automation

Ltac verify := repeat first
[ eapply step_read ;[ | side_condition ]
| ...
| tauto ].

Rtac Automation†

Def verify := repeat 10 first
[ eapply step_read_syn ;[ | side_condition ]
| ...
| rtauto ].

Thm verify_sound : rtac_sound verify.
Proof. derive soundness; ... Qed.

† Stylized Rtac syntax.
Program Verification using Combinators

Ltac Automation

Ltac verify := repeat first
  [ eapply step_read;[| side_condition]
  | ...
  | tauto ].

Rtac Automation†

Def verify := repeatₐ₀ first
  [ eapply step_read_syn;[| side_condition]
  | ...
  | rtauto ].

Thm verify_sound: rtac_sound verify.
Proof. derive soundness;... Qed.

† Stylized Rtac syntax.
Program Verification using Combinators

\( \text{Ltac Automation} \)

\[
\text{Ltac verify} := \text{repeat first}
\]
\[
[ \text{eapply step_read};[|\text{side_condition}]
\]
\[
| \ldots
\]
\[
| \text{tauto}.\]

\( \text{Rtac Automation} \)

\[
\text{Def verify} := \text{repeat}_{10} \text{first}
\]
\[
[ \text{eapply step_read_syn};[|\text{side_condition}]
\]
\[
| \ldots
\]
\[
| \text{rtauto}.\]

\[
\text{Thm verify_sound} : \text{rtac_sound verify.}
\]
\[
\text{Proof. derive soundness}; \ldots \text{Qed.}
\]

† Stylized \( \text{Rtac} \) syntax.
**Program Verification using Combinators**

**Ltac Automation**

\[
\text{Ltac verify := repeat first}
\begin{align*}
&\left[ \text{eapply step\_read;} \left| \text{side\_condition} \right. \right] \\
&\left| ... \right. \\
&\left| \text{tauto} \right].
\end{align*}
\]

**R\text{tac} Automation**

\[
\text{Def verify := repeat}_{10} \text{ first}
\begin{align*}
&\left[ \text{eapply step\_read\_syn;} \left| \text{side\_condition} \right. \right] \\
&\left| ... \right. \\
&\left| \text{rtauto} \right].
\end{align*}
\]

**Thm verify\_sound:** $\text{rtac\_sound verify}$.

**Proof.** derive soundness; ... Qed.

† Stylized $\text{R_{tac}}$ syntax.

Soundness derived composablely

Proof checked once and for all
Program Verification using Combinators

**Ltac Automation**

```ocaml
Ltac verify := repeat first
   [ eapply step_read; [ | side_condition ]
   | ... ]
   | tauto ].
```

**Rtac Automation†**

```ocaml
Def verify := repeat_10 first
   [ eapply step_read_syn; [ | side_condition ]
   | ... ]
   | rtauto ].
```

**Thm** verify_sound: rtac_sound verify.

**Proof.** derive soundness; ... Qed.

† Stylized Rtac syntax.

Builds the generic proof

Proof checked once and for all

Soundness derived composablely
Verifying $\mathcal{R}_{tac}$: Soundly Assembling Proofs

\[
\text{llist } x \mid \exists l \mid s \ n, \ldots \quad \{ \exists \mid l \mid s \ n, x \mapsto (l, n) \ast \text{llist } n \mid s \} \ c_2; \ c_3 \{ \ ?Q \} \\
\{ \text{llist } x \mid s \} \ c_1; \ c_2; \ c_3 \{ \ ?Q \}
\]
Verifying $\text{Tac}$: Soundly Assembling Proofs

Parallel obligations

\[
\text{lList } x \mid l \rightarrow \exists l \mid s n, \ldots
\]

\[
\exists l \mid s n, x \mapsto (l, n) \ast \text{lList } n \mid s \{c_2; c_3 \{?Q\}\}
\]

\[
\{\text{lList } x \mid s\} c_1; c_2; c_3 \{?Q\}
\]

“Free” unification variables
Verifying $\mathcal{R}_{\text{tac}}$: Soundly Assembling Proofs

$$\forall l \, l' \, n \, \{ x \mapsto (l, n) \ast \text{llist} \, n \, l \} : c_2; c_3 \{ ?Q \}$$

$$\exists l \, l' \, n \, \{ x \mapsto (l, n) \ast \text{llist} \, n \, l \} : c_2; c_3 \{ ?Q \}$$

Local Reasoning

Combine matching proofs

$$\text{llist} \, x \, l \, \{ ? \} \vdash \exists l \, l', n \, \{ x \mapsto (l, n) \ast \text{llist} \, n \, l \} : c_2; c_3 \{ ?Q \}$$

$$\{ \text{llist} \, x \, l \} : c_1; c_2; c_3 \{ ?Q \}$$
Verifying $\mathcal{R}_{\text{tac}}$: Soundly Assembling Proofs

**Phase-split:** Object-level terms must not affect $\mathcal{R}_{\text{tac}}$ invariants

Reason under binders

False $\rightarrow 1 = 2$ $\land \mathcal{R}_{\text{tac}}$-inv

$\forall l \; ls \; n, \{ x \leftrightarrow (l, n) \ast \text{llist} \; n \; ls \} c_2 ; c_3 \{ ?Q \}$

$\exists l \; ls \; n, x \leftrightarrow (l, n) \ast \text{llist} \; n \; ls \} c_2 ; c_3 \{ ?Q \}$

Local Reasoning

Combine matching proofs

$l\text{list} \; x \; ls \vdash \exists l \; ls \; n, \ldots$

$\exists l \; ls \; n, x \leftrightarrow (l, n) \ast \text{llist} \; n \; ls \} c_2 ; c_3 \{ ?Q \}$

$\{ l\text{list} \; x \; ls \} c_1 ; c_2 ; c_3 \{ ?Q \}$
Verifying $\mathcal{R}_{tac}$: Soundly Assembling Proofs

**Phase-split**: Object-level terms must not affect $\mathcal{R}_{tac}$ invariants

**Reason under binders**

\[
\forall l \, ls \, n, \{ x \mapsto (l, n) \ast \text{llist } n \, ls \} c_2 ; c_3 \{ ?Q \} \\
\exists l \, ls \, n, x \mapsto (l, n) \ast \text{llist } n \, ls \} c_2 ; c_3 \{ ?Q \}
\]

**Local Reasoning**

Combine matching proofs

\[
\text{llist } x \, ls \vdash \exists l \, ls \, n, \ldots \\
\{ \exists l \, ls \, n, x \mapsto (l, n) \ast \text{llist } n \, ls \} c_2 ; c_3 \{ ?Q \} \\
\{ \text{llist } x \, ls \} c_1 ; c_2 ; c_3 \{ ?Q \}
\]
Verifying $\mathcal{R}_{\text{tac}}$: Soundly Assembling Proofs

\[ \frac{?Q = P}{P \vdash ?Q} \]
\[ \{ P \} - \{ ?Q \} \]

Global Reasoning

Instantiate unification variable

\[ \forall l \mid l s n, \{ x \mapsto (l, n) \ast l l i s n l s \} c_2; c_3 \{ ?Q \} \]

\[ \{ \exists l \mid l s n, x \mapsto (l, n) \ast l l i s n l s \} c_2; c_3 \{ ?Q \} \]

Local Reasoning

Combine matching proofs

\[ \text{llist } x \mid l s \vdash \exists l \mid l s n, \ldots \]

\[ \{ \exists l \mid l s n, x \mapsto (l, n) \ast l l i s n l s \} c_2; c_3 \{ ?Q \} \]

\[ \{ \text{llist } x \mid l s \} c_1; c_2; c_3 \{ ?Q \} \]
Verifying $R_{\text{ tac}}$: Soundly Assembling Proofs

Global Reasoning

\[ \?Q = P \rightarrow \forall l \cdot s \cdot n, \{ x \mapsto (l, n) \} \cdot \text{llist} \cdot n \cdot s \cdot c_2; c_3 \{ ?Q \} \]

Local Reasoning

\[ \text{llist} \cdot x \cdot s \cdot n, \ldots \rightarrow \?Q = P \rightarrow \exists l \cdot s \cdot n, x \mapsto (l, n) \cdot \text{llist} \cdot n \cdot s \cdot c_2; c_3 \{ ?Q \} \]

Ensure that the choice is valid in the stronger context

\[ \?Q = P \rightarrow \{ P \} - \{ ?Q \} \]

Combine matching proofs
**Open computational reflection** in intensional type theories can lower the cost of writing automation that is simultaneously trustworthy, scalable, composable, and customizable.
Enriching the Framework

The Lambda Cube

Like HOL

- Polymorphism
  - "Fake it" with specialized term algebras. Details in thesis.

- Type Functions
  - "Fake it" with specialized type algebras. Details in thesis.

- Term Dependency
  - Cyclic dependency between types and terms! Open problem with interesting ramifications. Topology [Shu14] Lots of work [Dan07, Cha09, McB10]
Enriching the Framework

- **Polymorphism ✓**
  - "Fake it" with specialized *term* algebras.
  - Details in thesis.
  - ✗ Do not support type variables.

The Lambda Cube

Like HOL

Like Coq
Enriching the Framework

- **Polymorphism ✓**
  - “Fake it” with specialized *term* algebras.
  - Details in thesis.
  - ✗ Do not support type variables.

- **Type Functions ✓**
  - “Fake it” with specialized *type* algebras
  - Details in thesis.

The Lambda Cube

- $\lambda$ to $\lambda_2$
- $\lambda_2$ to $\lambda$

Enriched Theories | Extensible Proof Engineering in Intensional Type Theory
Enriching the Framework

- **Polymorphism ✓**
  - “Fake it” with specialized *term* algebras.
  - Details in thesis.
  - Do not support type variables.

- **Type Functions ✓**
  - “Fake it” with specialized *type* algebras
  - Details in thesis.

The Lambda Cube

Like HOL

\[ \lambda 2 \rightarrow \lambda \omega \]

\[ \lambda \rightarrow \lambda \omega \]
Enriching the Framework

- **Polymorphism ✓**
  - “Fake it” with specialized *term* algebras.
  - Details in thesis.
  - × Do not support type variables.

- **Type Functions ✓**
  - “Fake it” with specialized *type* algebras
  - Details in thesis.

- **Term Dependency ×**
  - × Cyclic dependency between types and terms!
  - Open problem with interesting ramifications
    - Topology [Shu14]
    - Lots of work [Dan07, Cha09, McB10]

The Lambda Cube

Like HOL

Like Coq
Enriching the Framework

Like Coq

- **Polymorphism ✓**
  - “Fake it” with specialized term algebras.
  - Details in thesis.
  - Do not support type variables.

- **Type Functions ✓**
  - “Fake it” with specialized type algebras
  - Details in thesis.

- **Term Dependency ✗**
  - Cyclic dependency between types and terms!
  - Open problem with interesting ramifications
    - Topology [Shu14]
    - Lots of work [Dan07, Cha09, McB10]

The Lambda Cube

Polymorphism

Type Functions

Term Dependency
Revisiting the Thesis

Open computational reflection in intensional type theories can lower the cost of writing automation that is simultaneously trustworthy, scalable, composable, and customizable.
Thank You

Greg Morrisett
Adam Chlipala
Stephen Chong

Thomas Braibant
Jesper Bengtson
Ryan Wisnesky

Mom & Dad

Elizabeth Malecha
MD 309, PLV@MIT, Antonis Stampoulis, Uri Braun
References

Samuel Boutin.
Using reflection to build efficient and certified decision procedures.

James Chapman.
Type theory should eat itself.

Adam Chlipala.
Mostly-automated verification of low-level programs in computational separation logic.

Adam Chlipala.
From network interface to multithreaded web applications: A case study in modular program verification.
2015.
To Appear.

Nils Anders Danielsson.
A formalisation of a dependently typed language as an inductive-recursive family.

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers.
Meta-theory a la carte.

Conor McBride.
Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation.
Gregory Malecha, Adam Chlipala, and Thomas Braibant.
Compositional computational reflection.
In Interactive Theorem Proving, 2014.

Michael Shulman.
Homotopy type theory should eat itself (but so far, it’s too big to swallow), March 2014.

Wouter Swierstra.
Data types à la carte.