Extensible Proof Engineering in Intensional Type Theory

Gregory Malecha
gmalecha@cs.harvard.edu

PhD Defense
Harvard SEAS

February 2, 2015

Mechanized Reasoning Tools

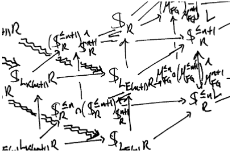
Mathematics

Software

Mechanized Reasoning Tools

Mathematics

Proofs

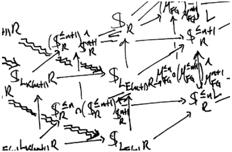


Software

Mechanized Reasoning Tools

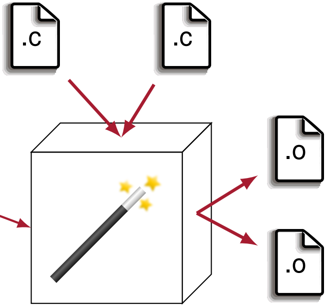
Mathematics | **Software**

Proofs



Compilers

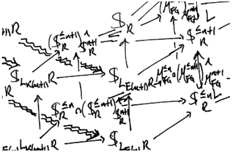
Optimizations



Mechanized Reasoning Tools

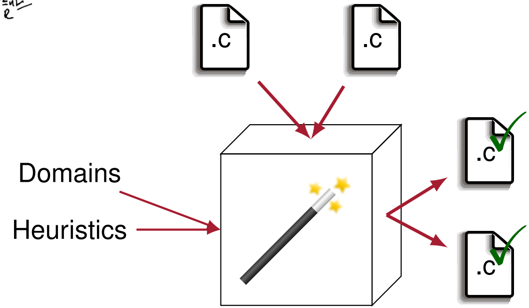
Mathematics

Proofs



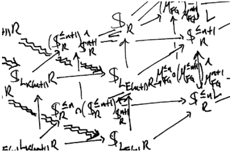
Software

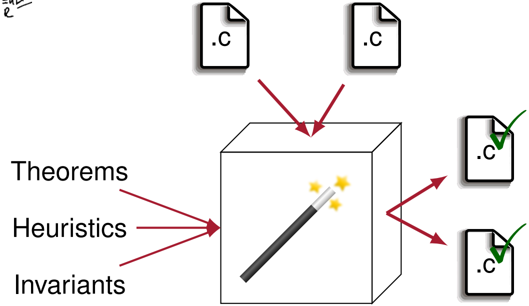
Compilers Analyzers



Mechanized Reasoning Tools

Mathematics	**Software**		
Proofs	Compilers	Analyzers	Verifiers

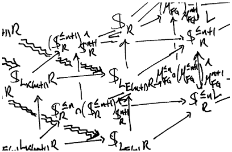




Mechanized Reasoning Tools

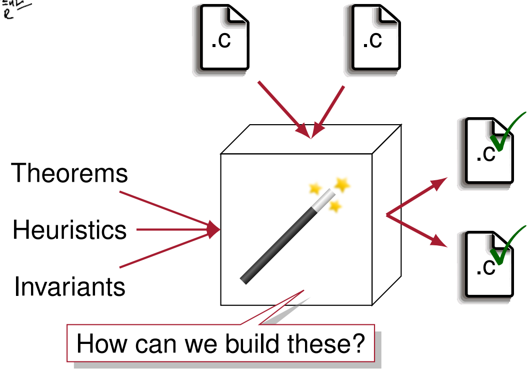
Mathematics

Proofs



Software

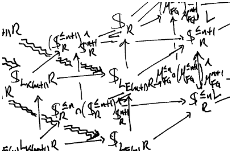
Compilers Analyzers Verifiers



Mechanized Reasoning Tools

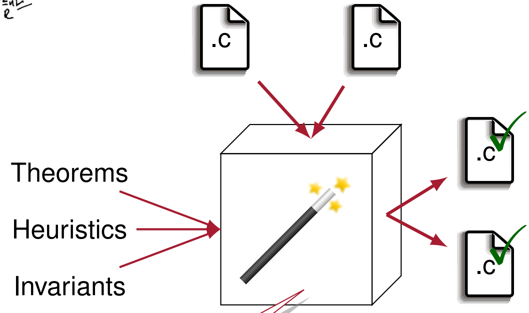
Mathematics

Proofs



Software

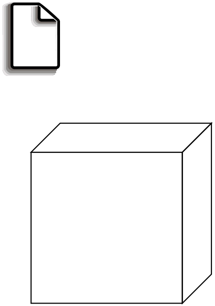
Compilers Analyzers **Verifiers**



How can we build these?

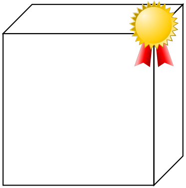
Thesis

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



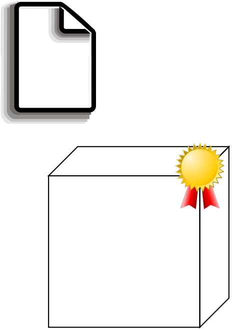
Thesis

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



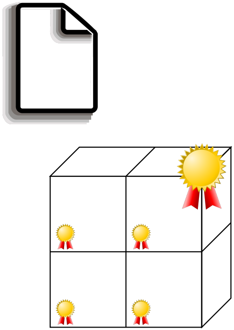
Thesis

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



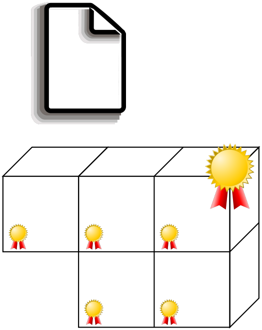
Thesis

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



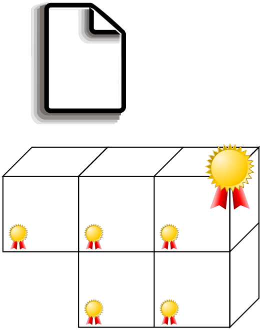
Thesis

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



Thesis

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

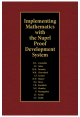
ACL2

Isabelle

HOL

Coq

Agda



Nuprl

Andromeda

LF/ELF/TWELF

Trustworthiness from a Logic

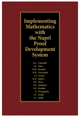
Foundational

ACL2

HOL

Coq

Agda



Nuprl

Andromeda

LF/ELF/TWELF

Small kernel

(DeBruijn criterion)

Foundational Proofs for Simple Entailments

$$\left.\begin{array}{cl} A \land (B \land C) \vdash A \land (B \land C) & \\ \hline A \land (B \land C) \vdash (A \land B) \land C & \land\text{-}\textsc{Assoc} \\ \hline A \land (B \land C) \vdash C \land (A \land B) & \land\text{-}\textsc{Comm} \\ \hline A \land (B \land C) \vdash C \land (B \land A) & \land\text{-}\textsc{Comm} \end{array}\right\} \text{Proof tree}$$

Foundational Proofs for Simple Entailments

$$\left.\dfrac{\dfrac{\dfrac{A \land (B \land C) \vdash A \land (B \land C)}{A \land (B \land C) \vdash (A \land B) \land C}\ \land\text{-}\textsc{Assoc}}{A \land (B \land C) \vdash C \land (A \land B)}\ \land\text{-}\textsc{Comm}}{A \land (B \land C) \vdash C \land (B \land A)}\ \land\text{-}\textsc{Comm}\right\}\ \text{Proof tree}$$

Foundational proofs require that we make all steps explicit.

Foundational Proofs for Simple Entailments

$$\dfrac{\dfrac{\dfrac{A \wedge (B \wedge C) \vdash A \wedge (B \wedge C)}{A \wedge (B \wedge C) \vdash (A \wedge B) \wedge C}\ \wedge\text{-}\textsc{Assoc}}{A \wedge (B \wedge C) \vdash C \wedge (A \wedge B)}\ \wedge\text{-}\textsc{Comm}}{A \wedge (B \wedge C) \vdash C \wedge (B \wedge A)}\ \wedge\text{-}\textsc{Comm}$$

builds

Foundational proofs require that we make all steps explicit.

$\mathscr{L}_{tac}$ Automation

```
Ltac my_tauto := repeat
  first [ reflexivity
        | apply ∧-Comm
        | apply ∧-Assoc
        | ... ].
```

Foundational Proofs for Simple Entailments

$$\dfrac{\dfrac{\dfrac{A \wedge (B \wedge C) \vdash A \wedge (B \wedge C)}{A \wedge (B \wedge C) \vdash (A \wedge B) \wedge C}\ \wedge\text{-}\textsc{Assoc}}{A \wedge (B \wedge C) \vdash C \wedge (A \wedge B)}\ \wedge\text{-}\textsc{Comm}}{A \wedge (B \wedge C) \vdash C \wedge (B \wedge A)}\ \wedge\text{-}\textsc{Comm}$$

Still have to build & check the proof

builds

Foundational proofs require that we make all steps explicit.

$\mathscr{L}_{tac}$ Automation

```
Ltac my_tauto := repeat
  first [ reflexivity
        | apply ∧-Comm
        | apply ∧-Assoc
        | ... ].
```

Foundational Proofs for Simple Entailments

Kernel cannot use custom algorithms!

$$\cfrac{\cfrac{\cfrac{A \wedge (B \wedge C) \vdash A \wedge (B \wedge C)}{A \wedge (B \wedge C) \vdash (A \wedge B) \wedge C}\ \wedge\text{-}\textsc{Assoc}}{A \wedge (B \wedge C) \vdash C \wedge (A \wedge B)}\ \wedge\text{-}\textsc{Comm}}{A \wedge (B \wedge C) \vdash C \wedge (B \wedge A)}\ \wedge\text{-}\textsc{Comm}$$

Still have to build & check the proof

builds

Foundational proofs require that we make all steps explicit.

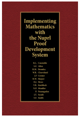
$\mathcal{L}_{tac}$ Automation

```
Ltac my_tauto := repeat
  first [ reflexivity
        | apply ∧-Comm
        | apply ∧-Assoc
        | ... ].
```

Trustworthiness from a Logic

Foundational

Coq



Nuprl

HOL

Agda

Andromeda

LF/ELF/TWELF

Small kernel
(DeBruijn criterion)

Trustworthiness from a Logic

Intensional | Extensional

Foundational | **Decidable** | Undecidable

ACL2

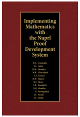
HOL

Coq

Agda

Andromeda

LF/ELF/TWELF



Nuprl

Small kernel

(DeBruijn criterion)

Computation in Logic/Type Theory

$$\frac{P \equiv Q \qquad \vdash Q}{\vdash P}\ \textsc{Conv}^{\dagger}$$

Meta-logic equality

† Abbreviated from the actual type theory rule.

Computation in Logic/Type Theory

Ext. Type Theory **Int. Type Theory**

$$\frac{\overbrace{}^{}\underbrace{\vdash P = Q \qquad P \Longrightarrow^* Q}_{P \equiv Q} \qquad \vdash Q}{\vdash P}\ \textsc{Conv}^{\dagger}$$

† Abbreviated from the actual type theory rule.

Computation in Logic/Type Theory

Ext. Type Theory　　　**Int. Type Theory**

$$\underbrace{\vdash P = Q \qquad P \Longrightarrow^{*} Q}$$

Execute the term

$$\frac{P \equiv Q \qquad \vdash Q}{\vdash P}\ \textsc{Conv}^{\dagger}$$

† Abbreviated from the actual type theory rule.

Trustworthiness from a Logic

Intensional | Extensional

Foundational | **Decidable** | Undecidable

ACL2

Isabelle

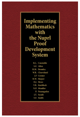
HOL

Coq

Agda

Andromeda

LF/ELF/TWELF



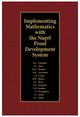
Nuprl

Small kernel

(DeBruijn criterion)

Trustworthiness from a Logic

	Simple(r) Types — Foundational	**Dependent Types** — Decidable	Undecidable
ACL2	Isabelle HOL	Coq Agda Andromeda	Nuprl
Computational			
		LF/ELF/TWELF	



Use this to compress proofs

Small kernel
(DeBruijn criterion)

Trustworthiness from a Logic

Simple(r) Types | Dependent Types

Foundational | Decidable | Undecidable

ACL2

Isabelle

HOL

Coq

Agda

Andromeda

Nuprl

Computational

Use this to compress proofs

LF/ELF/TWELF

Small kernel

(DeBruijn criterion)

Computation in Logic/Type Theory

Ext. Type Theory **Int. Type Theory**

$$\underbrace{\vdash P = Q \qquad P \Longrightarrow^{*} Q}$$

Many steps!

$$\frac{P \equiv Q \qquad \vdash Q}{\vdash P}\ \textsc{Conv}^{\dagger}$$

† Abbreviated from the actual type theory rule.

Computational Reflection [Bou97]

$$\dfrac{\dfrac{\dfrac{A \wedge (B \wedge C) \vdash A \wedge (B \wedge C)}{A \wedge (B \wedge C) \vdash (A \wedge B) \wedge C}}{A \wedge (B \wedge C) \vdash C \wedge (A \wedge B)}}{A \wedge (B \wedge C) \vdash C \wedge (B \wedge A)}$$

Computational Reflection [Bou97]

Syntactic | Semantic

$$\cfrac{\cfrac{\cfrac{A \wedge (B \wedge C) \vdash A \wedge (B \wedge C)}{A \wedge (B \wedge C) \vdash (A \wedge B) \wedge C}}{A \wedge (B \wedge C) \vdash C \wedge (A \wedge B)}}{A \wedge (B \wedge C) \vdash C \wedge (B \wedge A)}$$

$$\frac{}{[\![A \,\underline{\wedge}\, (B \,\underline{\wedge}\, C) \vdash C \,\underline{\wedge}\, (B \,\underline{\wedge}\, A)]\!]_{Prop}} \xleftarrow{\textsc{Conv}}$$

Computational Reflection [Bou97]

Syntactic | Semantic

Function

$$\dfrac{\dfrac{\text{true} = \text{true}}{\text{rtauto}(A \underline{\wedge} (B \underline{\wedge} C) \vdash C \underline{\wedge} (B \underline{\wedge} A)) = \text{true}}}{[\![A \underline{\wedge} (B \underline{\wedge} C) \vdash C \underline{\wedge} (B \underline{\wedge} A)]\!]_{Prop}}$$

Soundness proof

$$\dfrac{\dfrac{\dfrac{A \wedge (B \wedge C) \vdash A \wedge (B \wedge C)}{A \wedge (B \wedge C) \vdash (A \wedge B) \wedge C}}{A \wedge (B \wedge C) \vdash C \wedge (A \wedge B)}}{A \wedge (B \wedge C) \vdash C \wedge (B \wedge A)}$$

```
Thm rtauto_sound : ∀ g,
  rtauto g = true → [[ g ]]_Prop.
Proof. ... Qed.
```

Computational Reflection [Bou97]

Syntactic | Semantic

Small proof, custom algorithm

Large proof

Function

$$\cfrac{\cfrac{\text{true} = \text{true}}{\text{rtauto}(A \underline{\triangle} (B \underline{\triangle} C) \vdash C \underline{\triangle} (B \underline{\triangle} A)) = \text{true}}}{[\![A \underline{\triangle} (B \underline{\triangle} C) \vdash C \underline{\triangle} (B \underline{\triangle} A)]\!]_{Prop}}$$

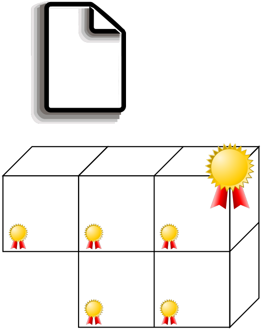
$$\cfrac{\cfrac{\cfrac{A \wedge (B \wedge C) \vdash A \wedge (B \wedge C)}{A \wedge (B \wedge C) \vdash (A \wedge B) \wedge C}}{A \wedge (B \wedge C) \vdash C \wedge (A \wedge B)}}{A \wedge (B \wedge C) \vdash C \wedge (B \wedge A)}$$

Soundness proof

```
Thm rtauto_sound : ∀ g,
  rtauto g = true → ⟦ g ⟧Prop.
Proof. ... Qed.
```

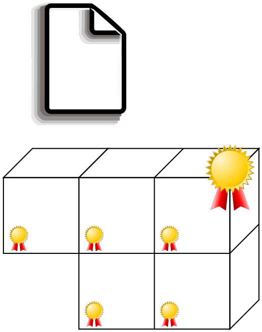
Thesis

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



Thesis

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



Composing Reflective Procedures

Logic	Arithmetic	Logic+Arith
`rtauto`	`arith`	`rtauto_arith`

Composing Reflective Procedures

Logic		Arithmetic		Logic+Arith
`True` $\wedge X$		$a + b = b + a$		`True` $\wedge (a + b = b + a)$
`rtauto`	$\oplus$	`arith`	$=$	`rtauto_arith`

Composing Reflective Procedures

Logic	$\oplus$	Arithmetic	$=$	Logic+Arith
$\texttt{True} \land X$		$a + b = b + a$		$\texttt{True} \land (a + b = b + a)$
`rtauto`		`arith`		`rtauto_arith`
$p ::= \texttt{True} \mid p_1 \land p_2$		$p ::= a_1 = a_2$ $a ::= a_1 + a_2$		

Composing Reflective Procedures

$\texttt{True} \wedge X$

Logic

rtauto

$pr ::= \texttt{True} \mid r \wedge r$

$\oplus$

$a + b \equiv b + a$

Arithmetic

arith

$p ::= a_1 \equiv a_2$

$ar ::= r + r$

$=$

$\texttt{True} \wedge (a + b \equiv b + a)$

Logic+Arith

rtauto_arith

Datatypes *a.la. carte* [Swi08]
Metatheory *a.la. carte* [DdSOS13]

Composing Reflective Procedures

$\texttt{True} \wedge X$

Logic

$\texttt{rtauto}$

$p ::= \texttt{True} \mid p_1 \wedge p_2$

$\oplus$

$a + b = b + a$

Arithmetic

$\texttt{arith}$

$p ::= a_1 = a_2$

$a ::= a_1 + a_2$

$=$

$\texttt{True} \wedge (a + b = b + a)$

Logic+Arith

$\texttt{rtauto\_arith}$

$\texttt{X}_{\wedge}\ @\ \texttt{X}_{True}\ @$
$(\texttt{X}_{=}\ @\ (\texttt{X}_{+}\ @\ \texttt{a}\ @\ \texttt{b})$
$@\ (\texttt{X}_{+}\ @\ \texttt{b}\ @\ \texttt{a}))$

Key Insight!

$t ::= \texttt{T}_{\#} \mid t_1 \to t_2$

$e ::= \texttt{X}_{\#} \mid e_1 @ e_2 \mid \lambda t.e \mid \texttt{x}_n$

$\lambda\ (X)$

Composing Reflective Procedures

$\texttt{True} \wedge X$

Logic

`rtauto`

$p ::= \texttt{True} \mid p_1 \wedge p_2$

$\oplus$

$a + b = b + a$

Arithmetic

`arith`

$p ::= a_1 = a_2$

$a ::= a_1 + a_2$

$=$

$\texttt{True} \wedge (a + b = b + a)$

Logic+Arith

`rtauto_arith`

$\texttt{X}_\wedge \ @\ \texttt{X}_{True}\ @$
$(\texttt{X}_= \ @\ (\texttt{X}_+ \ @\ \texttt{a}\ @\ \texttt{b})$
$\quad @\ (\texttt{X}_+ \ @\ \texttt{b}\ @\ \texttt{a}))$

$t ::= \texttt{T}_\# \mid t_1 \rightarrow t_2$

$e ::= \texttt{X}_\# \mid e_1 @ e_2 \mid \lambda t.e \mid \texttt{x}_n$

$\lambda\ (X)$

Define independently

Semantic

Composing Reflective Procedures

$\texttt{True} \wedge X$	$a + b = b + a$	$\texttt{True} \wedge (a + b = b + a)$

Logic		Arithmetic		Logic+Arith
`rtauto`	$\oplus$	`arith`	$=$	`rtauto_arith`

$p ::= \texttt{True} \mid p_1 \wedge p_2$

$p ::= a_1 = a_2$

$a ::= a_1 + a_2$

$\texttt{X}_\wedge$ @ $\texttt{X}_{True}$ @
($\texttt{X}_=$ @ ($\texttt{X}_+$ @ a @ b)
@ ($\texttt{X}_+$ @ b @ a))

$t ::= \texttt{T}_\# \mid t_1 \rightarrow t_2$

$e ::= \texttt{X}_\# \mid e_1 @ e_2 \mid \lambda t.e \mid \texttt{x}_n$

Define independently

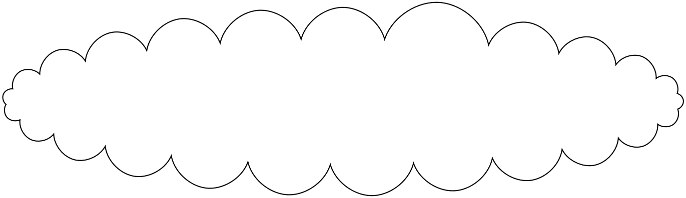
$\lambda\ (X)$

Semantic

Soundness theorems reason about the denotation function

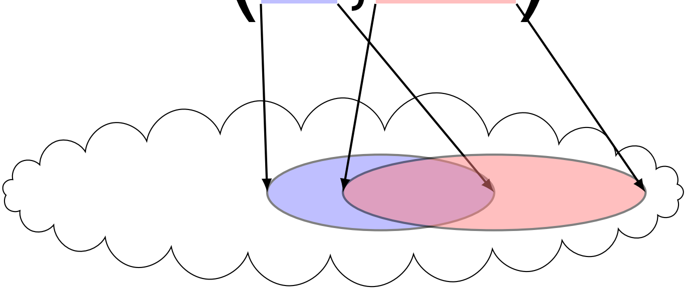
Composing Reflective Automation

$$\lambda\left(\boxed{\wedge}, \boxed{+, =}\right)$$

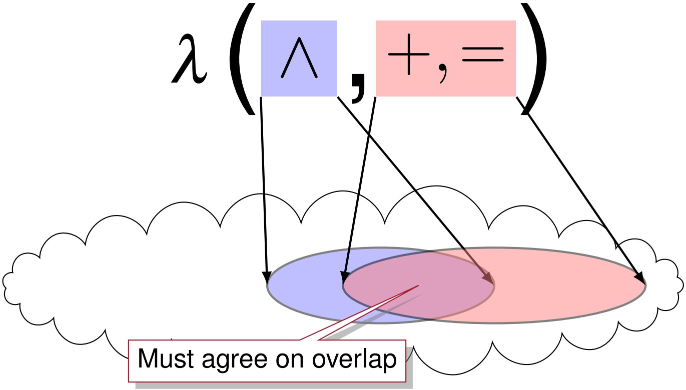


Composing Reflective Automation

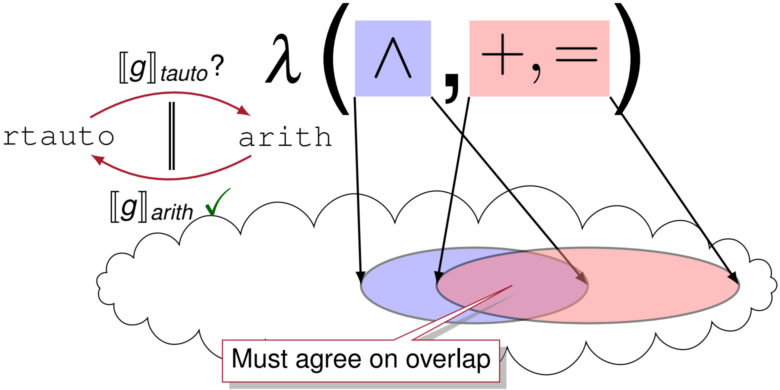
$$\lambda\left(\boxed{\land}, \boxed{+, =}\right)$$



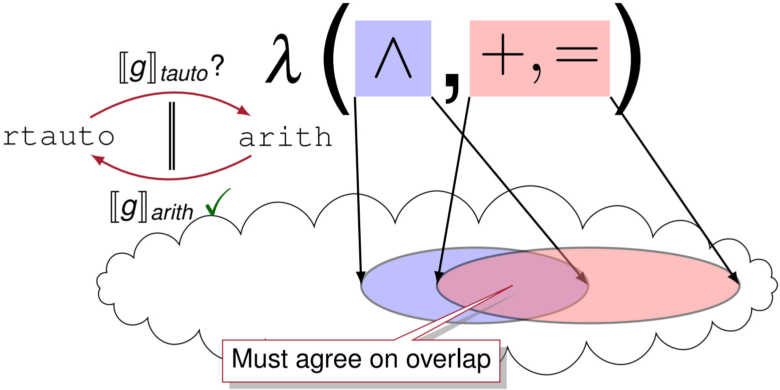
Composing Reflective Automation



Composing Reflective Automation



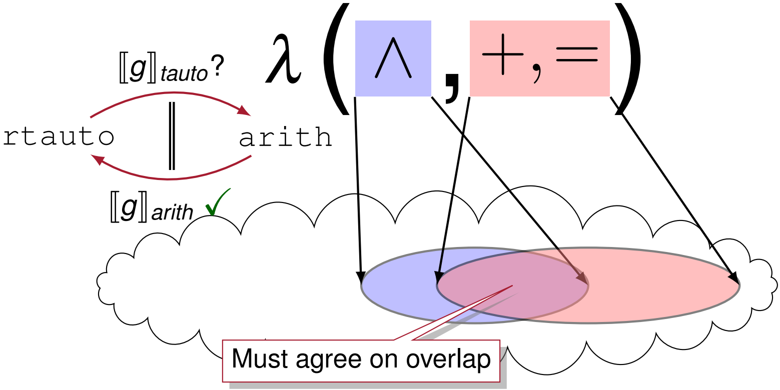
Composing Reflective Automation



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

```
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.
```

Soundness Proof

```
Thm rtauto_sound
: ∀ g, rtauto g = true →
   ⟦ g ⟧_tyProp.
Proof. ... Qed.
```

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

```
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 : ⟦ tyProp ⟧ = Prop.
Var pfTr : ⟦sTr⟧_tyProp = True.
Var pfAnd : ⟦sAnd⟧_... = ∧.

Thm rtauto_sour
: ∀ g, rtauto g =
  ⟦ g ⟧_tyProp.
Proof. ... Qed.
```

Type Error!

$[\![\mathtt{tyProp}]\!] \neq \mathtt{Prop}$

Composable Automation

- Explicit casts

$$\texttt{Ha} : \texttt{cast}_{pfP}\ \texttt{A}$$
$$\texttt{Hb} : \texttt{cast}_{pfP}\ \texttt{B}$$
====================
$$\texttt{cast}_{pfP}\ (\texttt{A} \wedge \texttt{B})$$

```
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 : ⟦ tyProp ⟧ = Prop.
Var pfTr : cast_pfP ⟦sTr⟧_tyProp = True.
Var pfAnd : cast_pfP ⟦sAnd⟧_... = ∧.

Thm rtauto_sound
: ∀ g, rtauto g = true →
  cast_pfP ⟦ g ⟧_tyProp.
Proof. ... Qed.
```

Composable Automation

- Explicit casts

```
Ha : cast_pfP A
Hb : cast_pfP B
====================
cast_pfP (A ∧ B)
```

- Composable only when proofs match up exactly

```
Ha : cast_pfP A
====================
cast_pfQ A
```

✓ Very flexible

✗ Verbose

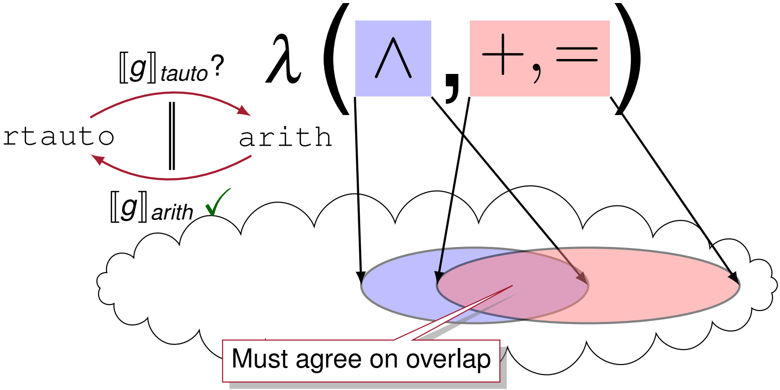
```
Var tyProp : typ.
Var sTr sAnd : sym.

Def rtauto (g : expr) : bool :=
  match g with
  | X_sTr ⇒ true
  | X_sAnd @ l @ r ⇒
    rtauto l && rtauto r
  | _ ⇒ false
  end.

Var pfP : ⟦ tyProp ⟧ = Prop.
Var pfTr : cast_pfP ⟦sTr⟧_tyProp = True.
Var pfAnd : cast_pfP ⟦sAnd⟧_... = ∧.

Thm rtauto_sound
: ∀ g, rtauto g = true →
  cast_pfP ⟦ g ⟧_tyProp.
Proof. ... Qed.
```

Composing Reflective Automation



Two ways to achieve this

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

Composition with Environments

```
Let tyProp := T0. (* typ *)
Let sTr := X0.
Let sAnd := X1.
```

Use numbers

τ_0	τ_1	τ_2	...

```
Def rtauto (g : expr) : bool :=
  match g with
  | X_sTr ⇒ true
  | X_sAnd @ l @ r ⇒
    rtauto l && rtauto r
  | _ ⇒ false
  end.

Var ts : list Type.
Var fs : list ...
```

and environments

```
Thm rtauto_sound
: ∀ g, rtauto g = true →
  ts/fs⟦ g ⟧tyProp.
Proof. ... Qed.
```

passed to ⟦ ⟧

Composition with Environments

```
Let tyProp := T_0. (* typ *)
Let sTr := X_0.
Let sAnd := X_1.

Def rtauto (g : expr) : bool :=
  match g with
  | X_sTr ⇒ true
  | X_sAnd @ l @ r ⇒
    rtauto l && rtauto r
  | _ ⇒ false
  end.
```

```
Var ts : list Type.
Var fs : list ...

Thm rtauto_sound
: ∀ g, rtauto g = true →
  ts/fs⟦ g ⟧tyProp.
Proof. ... Qed.
```

Var ts :

τ_0	τ_1	τ_2	...

```
Thm rtauto_sound
: ∀ g, rtauto g = true →
  ts/fs⟦ g ⟧tyProp.
Proof. ... Qed.
```

Composition with Environments

```
Let tyProp := T₀. (* typ *)
Let sTr := X₀.
Let sAnd := X₁.

Def rtauto (g : expr) : bool :=
  match g with
  | X_sTr ⇒ true
  | X_sAnd @ l @ r ⇒
    rtauto l && rtauto r
  | _ ⇒ false
  end.
```

```
Var ts : list Type.
Var fs : list ...

Thm rtauto_sound
: ∀ g, rtauto g = true →
  ts/fs⟦ g ⟧tyProp.
Proof. ... Qed.
```

Var ts :

τ_0	τ_1	τ_2	...

```
Thm rtauto_sound
: ∀ g, rtauto g = true →
  ts/fs⟦ g ⟧tyProp.
Proof. ... Qed.
```

$[\![T_0]\!] \neq \mathbb{P}$ ✗

Composition with Environments

```
Let tyProp := T₀. (* typ *)
Let sTr := X₀.
Let sAnd := X₁.

Def rtauto (g : expr) : bool :=
  match g with
  | X_sTr ⇒ true
  | X_sAnd @ l @ r ⇒
    rtauto l && rtauto r
  | _ ⇒ false
  end.
```

```
Var ts : list Type.
Var fs : list ...

Thm rtauto_sound
: ∀ g, rtauto g = true →
  ts/fs⟦ g ⟧tyProp.
Proof. ... Qed.
```

Var ts :	$\mathbb{P}$	$\mathbb{B}$	$\mathbb{N}$

```
Thm rtauto_sound
: ∀ g, rtauto g = true →
  ts/fs⟦ g ⟧tyProp.
Proof. ... Qed.
```

$[\![\underline{T}_0]\!] \equiv \mathbb{P}$ ✓

Composition with Environments

Let tyProp := $\underline{\mathrm{T}}_0$. (* typ *)
Let sTr := $\underline{\mathrm{X}}_0$.
Let sAnd := $\underline{\mathrm{X}}_1$.

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

Var ts : list Type.
Var fs : list ...

Thm rtauto_sound
: ∀ g, rtauto g = true →
$^{ts \oplus c}_{fs}[\![\, g \,]\!]_{\mathrm{tyProp}}$.
Proof. ... Qed.

Var ts :	τ_0	τ_1	τ_2	...
		$\oplus$		
Let c :=	$\mathbb{P}$	?	?	...
		$\equiv$		
	$\mathbb{P}$	τ_1	τ_2	...

Thm rtauto_sound
: ∀ g, rtauto g = true →
$^{ts \oplus c}_{fs}[\![\, g \,]\!]_{\mathrm{tyProp}}$.
Proof. ... Qed.

$[\![\underline{\mathrm{T}}_0]\!] \equiv \mathbb{P}$ ✓

Generic Reflective Automation

- Some tasks are very easy to automate

Proof

$$\frac{}{\vdash x \in (\{x, y\} \cup \{z\})}$$

Generic Reflective Automation

- Some tasks are very easy to automate

Proof

$$\dfrac{\dfrac{}{\vdash x \in \{x, y\}}}{\vdash x \in (\{x, y\} \cup \{z\})}\ \textsc{Lem2}$$

Generic Reflective Automation

- Some tasks are very easy to automate

Syntax Proof

$$\forall\, a\, B\, C,\quad a \in B \to a \in (B \cup C)$$

$$\frac{\overline{\vdash x \in \{x, y\}}}{\vdash x \in (\{x, y\} \cup \{z\})}\ \textsc{Lem2}$$

Generic Reflective Automation

- Some tasks are very easy to automate

Syntax Proof

Unification

$\forall\, a\, B\, C,$
$\quad ?a \in\, ?B \rightarrow$
$\quad ?a \in (?B \cup\, ?C)$

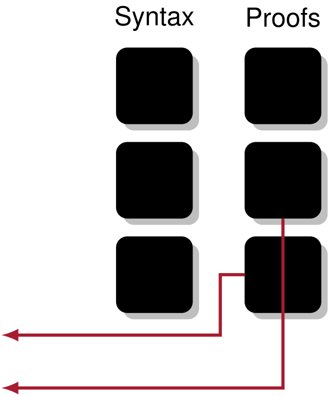
What expressions make the conclusion match the goal?

$$\dfrac{\dfrac{}{\vdash x \in \{x, y\}}}{\vdash x \in (\{x, y\} \cup \{z\})}\ \textsc{Lem2}$$

Generic Reflective Automation

- Some tasks are very easy to automate

$$\dfrac{\dfrac{\dfrac{}{\vdash x = x}\;\textsc{Lem3}}{\vdash x \in \{x, y\}}\;\textsc{Lem2}}{\vdash x \in (\{x, y\} \cup \{z\})}$$

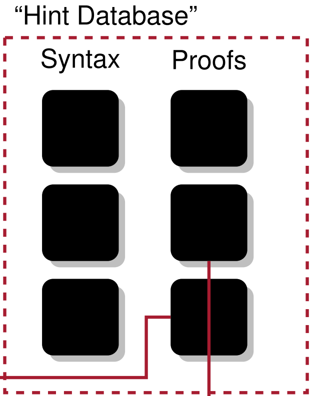


Generic Reflective Automation

- Some tasks are very easy to automate

Generic procedures make it easy to quickly build simple automation

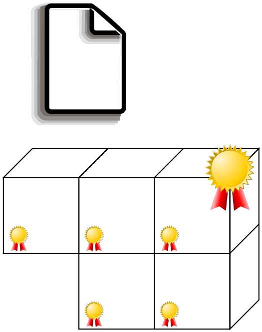
`auto`
`autorewrite`



$$\dfrac{\dfrac{}{\vdash x = x}\;\textsc{Lem3}}{\dfrac{\vdash x \in \{x, y\}}{\vdash x \in (\{x, y\} \cup \{z\})}\;\textsc{Lem2}}$$

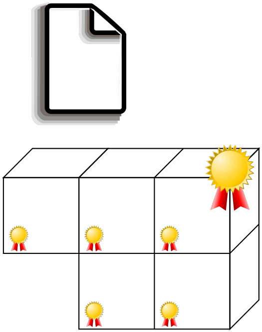
Thesis

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



Thesis

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



BEDROCK: Composablity, 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: Composablity, 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.

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

$< 20x$

"Overhead of verification"

BEDROCK: Macro Performance

- Does open computational reflection make verification faster? **Yes**

BEDROCK: Macro Performance

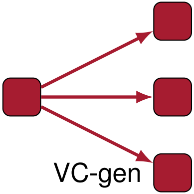
- Does open computational reflection make verification faster? **Yes**
- Does it make verification fast? **Reasonably**

BEDROCK: Macro Performance

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

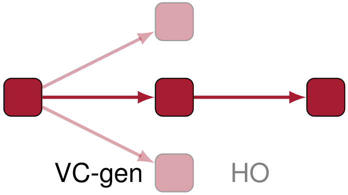
BEDROCK: Macro Performance

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



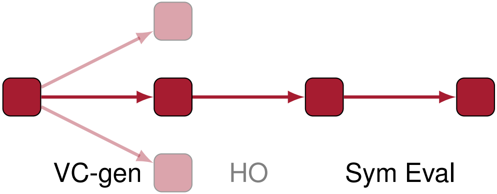
BEDROCK: Macro Performance

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



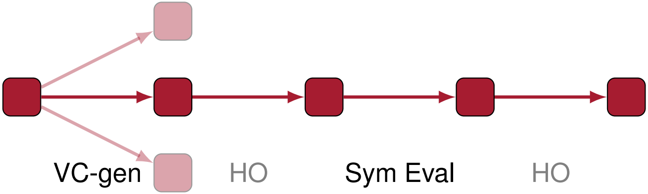
BEDROCK: Macro Performance

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



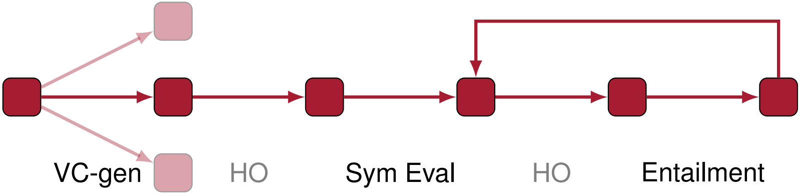
BEDROCK: Macro Performance

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



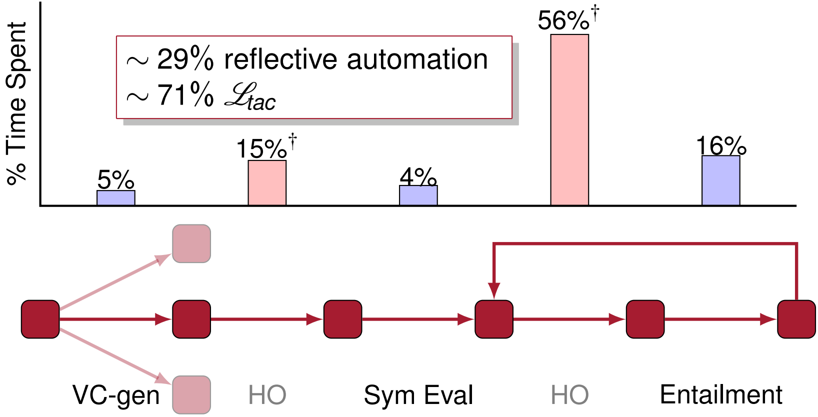
BEDROCK: Macro Performance

- 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**



† The division of the 71% is for illustrative purposes only, the results simply states that 71% of the total time is spent in $\mathscr{L}tac$.

BEDROCK: Customizability & Performance

- Customizability is essential for good performance.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\{\ldots\}-}}{\{\ldots\}c_3}}{\{\ldots\}c_2\,;c_3}}{\{x \mapsto (l, n) * \text{llist}\; ls'\; n\}c_2\,;c_3}}{\{x \neq 0 \land \text{llist}\; ls\; x\}c_2\,;c_3}}{\{\text{llist}\; ls\; x\}c_1\,;c_2\,;c_3}$$

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;
}
```

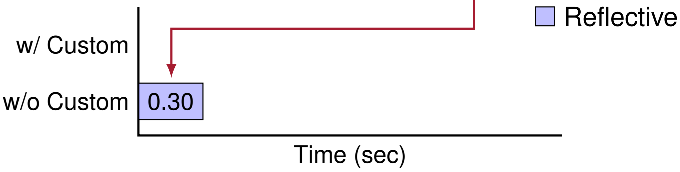
Bedrock: Customizability & Performance

- Customizability is essential for good performance.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\dots\}-}}{\{\dots\}c_3}}{\{\dots\}c_2;c_3}}{\{x \mapsto (l,n) * \text{llist } ls'\ n\}c_2;c_3}}{\{x \neq 0 \wedge \text{llist } ls\ x\}c_2;c_3}}{\{\text{llist } ls\ x\}c_1;c_2;c_3}
$$

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;
}
```



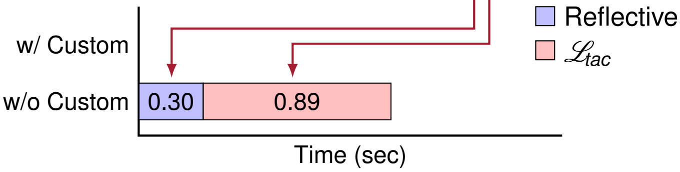
BEDROCK: Customizability & Performance

- Customizability is essential for good performance.

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\{\ldots\} -}}{\{\ldots\} c_3}}{\{\ldots\} c_2 ; c_3}}{\{x \mapsto (l, n) * \text{llist } ls' \; n\} c_2 ; c_3}}{\{x \neq 0 \wedge \text{llist } ls \; x\} c_2 ; c_3}}{\{\text{llist } ls \; x\} c_1 ; c_2 ; c_3}$$

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;
}
```



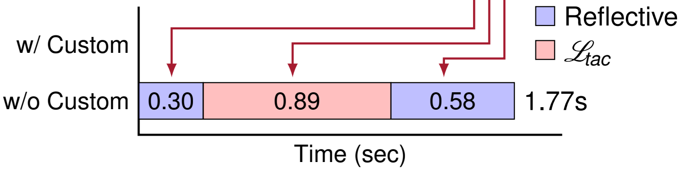
BEDROCK: Customizability & Performance

- Customizability is essential for good performance.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\ldots\}-}}{\{\ldots\}c_3}}{\{\ldots\}c_2;c_3}}{\{x \mapsto (l,n) * \text{llist}\ ls'\ n\}c_2;c_3}}{\{x \neq 0 \wedge \text{llist}\ ls\ x\}c_2;c_3}$$
$$\{\text{llist}\ ls\ x\}c_1;c_2;c_3$$

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;
}
```



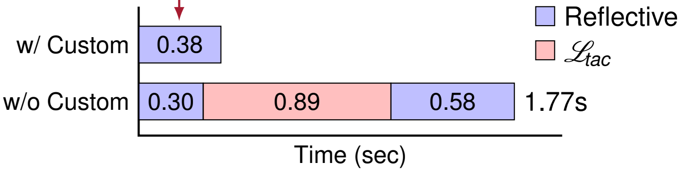
Bedrock: Customizability & Performance

- Customizability is essential for good performance.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\ldots\}\,-}}{\{\ldots\}\,c_3}}{\{\ldots\}\,c_2\,;\,c_3}}{\{x \mapsto (l, n) * \text{llist}\ ls'\ n\}\,c_2\,;\,c_3}}{\dfrac{\{x \neq 0 \wedge \text{llist}\ ls\ x\}\,c_2\,;\,c_3}{\{\text{llist}\ ls\ x\}\,c_1\,;\,c_2\,;\,c_3}}$$

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;
}
```



BEDROCK: Customizability & Performance

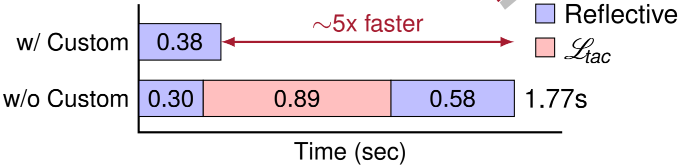
- Customizability is essential for good performance.

$$\{\ldots\}\,-$$
$$\{\ldots\}\,c_3$$
$$\{\ldots\}\,c_2\,;\,c_3$$
$$\{x \mapsto (l, n) * \text{llist}\ ls'\ n\}\,c_2\,;\,c_3$$
$$\{x \neq 0 \wedge \text{llist}\ ls\ x\}\,c_2\,;\,c_3$$
$$\{\text{llist}\ ls\ x\}\,c_1\,;\,c_2\,;\,c_3$$

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;
```

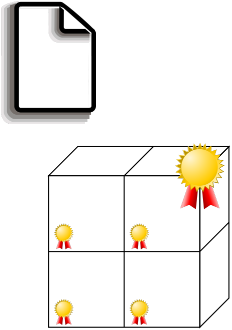
5x overall speedup



- Cost for entering "reflected" world

Thesis

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	Lists
Arith	Sets
Complex ⟷	Simple

A Whole Range of Reflective Procedures

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

```
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

A Whole Range of Reflective Procedures

- Build a language/library for writing/composing reflective procedures
- Capture backtracking proof search (similar to $\mathscr{L}_{tac}$)

```
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

$\mathcal{L}_{tac}$ Automation

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

Program Verification using Combinators

$\mathscr{L}_{tac}$ Automation

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

$\mathscr{R}_{tac}$ Automation†

```
Def verify := repeat₁₀ first
  [ eapply step_read_syn ;[| side_condition ]
  | ...
  | rtauto ].

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

† Stylized $\mathscr{R}_{tac}$ syntax.

Program Verification using Combinators

$\mathcal{L}_{tac}$ Automation

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

Functions

$\mathcal{R}_{tac}$ Automation†

```
Def verify := repeat₁₀ first
  [ eapply step_read_syn ;[| side_condition ]
  | ...
  | rtauto ].

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

† Stylized $\mathcal{R}_{tac}$ syntax.

Program Verification using Combinators

$\mathcal{L}_{tac}$ Automation

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

Functions

$\mathcal{R}_{tac}$ Automation†

```
Def verify := repeat10 first
  [ eapply step_read_syn ;[|side_condition]
  | ...
  | rtauto ].

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

† Stylized $\mathcal{R}_{tac}$ syntax.

Program Verification using Combinators

$\mathscr{L}_{tac}$ Automation

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

$\mathscr{R}_{tac}$ Automation†

```
Def verify := repeat10 first
  [ eapply step_read_syn ;[| side_condition ]
  | ...
  | rtauto ].

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

Functions

Proof checked once and for all

Soundness derived composably

† Stylized $\mathscr{R}_{tac}$ syntax.

Program Verification using Combinators

$\mathcal{L}_{tac}$ Automation

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

Functions

$\mathcal{R}_{tac}$ Automation†

```
Def verify := repeat₁₀ first
  [ eapply step_read_syn ;[| side_condition ]
  | ...
  | rtauto ].

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

Builds the generic proof

Proof checked once and for all

Soundness derived composably

† Stylized $\mathcal{R}_{tac}$ syntax.

Verifying $\mathscr{R}_{tac}$: Soundly Assembling Proofs

$$\frac{\mathsf{llist}\,x\,ls \vdash \exists l\,ls\,n, \ldots \qquad \{\exists l\,ls\,n, x \mapsto (l, n) * \mathsf{llist}\,n\,ls\}\,c_2; c_3\,\{?Q\}}{\{\mathsf{llist}\,x\,ls\}\,c_1; c_2; c_3\,\{?Q\}}$$

Verifying $\mathscr{R}_{tac}$: Soundly Assembling Proofs

Parallel obligations

$$\frac{\text{llist}\, x\, ls \vdash \exists l\, ls\, n, \ldots \qquad \{\exists l\, ls\, n, x \mapsto (l, n) * \text{llist}\, n\, ls\}\, c_2; c_3\, \{?Q\}}{\{\text{llist}\, x\, ls\}\, c_1; c_2; c_3\, \{?Q\}}$$

“Free” unification variables

Verifying $\mathscr{R}_{tac}$: Soundly Assembling Proofs

$$\dfrac{\llbracket\text{llist}\, x\, ls \vdash \exists l\, ls\, n, \ldots \rrbracket \quad \dfrac{\forall l\, ls\, n, \{x \mapsto (l, n) * \text{llist}\, n\, ls\}\, c_2; c_3\, \{?Q\}}{\{\exists l\, ls\, n, x \mapsto (l, n) * \text{llist}\, n\, ls\}\, c_2; c_3\, \{?Q\}} \quad \{\exists l\, ls\, n, x \mapsto (l, n) * \text{llist}\, n\, ls\}\, c_2; c_3\, \{?Q\}}{\{\text{llist}\, x\, ls\}\, c_1; c_2; c_3\, \{?Q\}}$$

Local Reasoning

Combine matching proofs

Verifying $\mathscr{R}_{tac}$: Soundly Assembling Proofs

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

$$\underline{\mathsf{False} \rightarrow 1 = 2} \;\; \wedge \mathscr{R}_{tac}\text{-inv}$$

Reason under binders

$$\frac{\forall l\, ls\, n, \{x \mapsto (l, n) * \text{llist}\, n\, ls\} c_2; c_3 \{?Q\}}{\{\exists l\, ls\, n, x \mapsto (l, n) * \text{llist}\, n\, ls\} c_2; c_3 \{?Q\}}$$

Local Reasoning

Combine matching proofs

$$\frac{\text{llist}\, x\, ls \vdash \exists l\, ls\, n, \ldots \qquad \{\exists l\, ls\, n, x \mapsto (l, n) * \text{llist}\, n\, ls\} c_2; c_3 \{?Q\}}{\{\text{llist}\, x\, ls\} c_1; c_2; c_3 \{?Q\}}$$

Verifying $\mathscr{R}_{tac}$: Soundly Assembling Proofs

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

$$\left(\underline{\text{False} \to 1 = 2}\right) \land \mathscr{R}_{tac}\text{-inv}$$

Reason under binders

$$\frac{\forall l\, ls\, n, \{x \mapsto (l, n) * \text{llist}\, n\, ls\} c_2; c_3 \{?Q\}}{\{\exists l\, ls\, n, x \mapsto (l, n) * \text{llist}\, n\, ls\} c_2; c_3 \{?Q\}}$$

Local Reasoning

Combine matching proofs

$$\frac{\text{llist}\, x\, ls \vdash \exists l\, ls\, n, \ldots \qquad \{\exists l\, ls\, n, x \mapsto (l, n) * \text{llist}\, n\, ls\} c_2; c_3 \{?Q\}}{\{\text{llist}\, x\, ls\} c_1; c_2; c_3 \{?Q\}}$$

Verifying $\mathscr{R}_{tac}$: Soundly Assembling Proofs

$$\dfrac{?Q = P \qquad\qquad P \vdash ?Q}{\{P\} - \{?Q\}}$$

Global Reasoning

Instantiate unification variable

$$\dfrac{\forall l\, ls\, n, \{x \mapsto (l, n) * \text{llist}\, n\, ls\} c_2; c_3 \{?Q\}}{\{\exists l\, ls\, n, x \mapsto (l, n) * \text{llist}\, n\, ls\} c_2; c_3 \{?Q\}}$$

Local Reasoning

Combine matching proofs

$$\dfrac{\text{llist}\, x\, ls \vdash \exists l\, ls\, n, \ldots \qquad \{\exists l\, ls\, n, x \mapsto (l, n) * \text{llist}\, n\, ls\} c_2; c_3 \{?Q\}}{\{\text{llist}\, x\, ls\} c_1; c_2; c_3 \{?Q\}}$$

Verifying $\mathscr{R}_{tac}$: Soundly Assembling Proofs

Propagate through the entire proof!

Ensure that the choice is valid in the stronger context

$$\frac{?Q = P \qquad P \vdash ?Q}{?\mathbf{Q} = \mathbf{P} \rightarrow \{P\} - \{?Q\}}$$

Global Reasoning

Instantiate unification variable

$$\frac{?\mathbf{Q} = \mathbf{P} \rightarrow \forall l\, ls\, n, \{x \mapsto (l, n) * \text{llist}\, n\, ls\} c_2; c_3 \{?Q\}}{?\mathbf{Q} = \mathbf{P} \rightarrow \{\exists l\, ls\, n, x \mapsto (l, n) * \text{llist}\, n\, ls\} c_2; c_3 \{?Q\}}$$

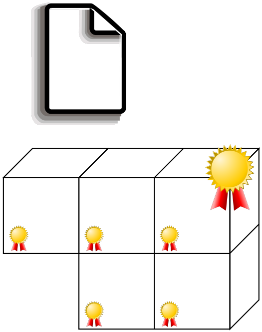
Local Reasoning

Combine matching proofs

$$\frac{\text{llist}\, x\, ls \vdash \exists l\, ls\, n, \ldots \qquad ?\mathbf{Q} = \mathbf{P} \rightarrow \{\exists l\, ls\, n, x \mapsto (l, n) * \text{llist}\, n\, ls\} c_2; c_3 \{?Q\}}{?\mathbf{Q} = \mathbf{P} \rightarrow \{\text{llist}\, x\, ls\} c_1; c_2; c_3 \{?Q\}}$$

Thesis

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

$\boxed{\lambda}$

The Lambda Cube

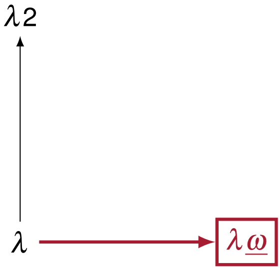
Enriching the Framework

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

The Lambda Cube

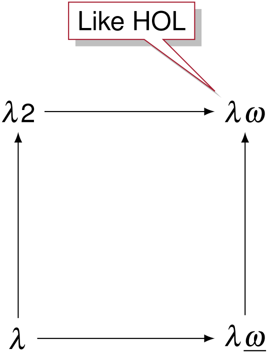
Enriching the Framework

- **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

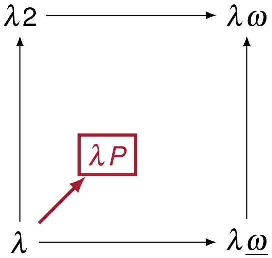
Enriching the Framework



The Lambda Cube

- **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.

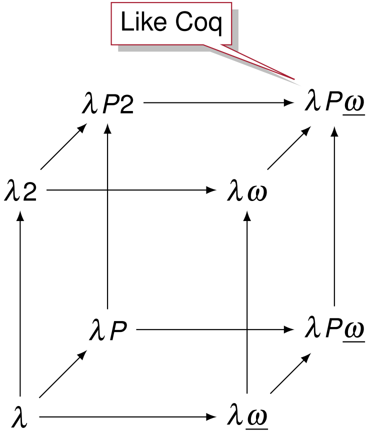
Enriching the Framework



The Lambda Cube

- **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]

Enriching the Framework



The Lambda Cube

- **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]

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

Elizabeth Malecha

Mom & Dad

MD 309, PLV@MIT, Antonis Stampoulis, Uri Braun

References I

Samuel Boutin.
Using reflection to build efficient and certified decision procedures.
In *Proc. TACS*, 1997.

James Chapman.
Type theory should eat itself.
Electron. Notes Theor. Comput. Sci., 228:21–36, January 2009.

Adam Chlipala.
Mostly-automated verification of low-level programs in computational separation logic.
In *Proc. PLDI*, pages 234–245. ACM, 2011.

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.
In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer Berlin Heidelberg, 2007.

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers.
Meta-theory a la carte.
SIGPLAN Not., 48(1):207–218, January 2013.

Conor McBride.
Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation.
In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, pages 1–12, New York, NY, USA, 2010. ACM.

References II

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.
Journal of Functional Programming, 18:423–436, 7 2008.