Automating Abstract Logics

Gregory Malecha Jesper Bengtson Adam Chlipala
gmalecha@cs.harvard.edu

Harvard SEAS ITU Denmark MIT CSAIL

July 18, 2014

Automating **Everything**

Gregory Malecha Jesper Bengtson Adam Chlipala
gmalecha@cs.harvard.edu

Harvard SEAS ITU Denmark MIT CSAIL

July 18, 2014

Reasoning in Coq

1. Math
2. Meta-theory
3. **Program verification**

Reasoning in Coq

1. Math
2. Meta-theory
3. **Program verification**

Formalize
a language
(Java[BJB12],
x86[JBK13],
C[App11],
Bedrock[Chl11])

Difficulties

1. Binding, validation, etc.

Reasoning in Coq

1. Math
2. Meta-theory
3. **Program verification**

Difficulties

1. Binding, validation, etc.
2. Enriched logics
   - State, step-indexing...

Formalize a language → Develop a logic

(Java[BJB12], x86[JBK13], C[App11], Bedrock[Chl11])

(Charge![BJB12], MSL[App11], ...)

Reasoning in Coq

1. Math
2. Meta-theory
3. **Program verification**

Difficulties

1. Binding, validation, etc.
2. Enriched logics
   - State, step-indexing...
3. Customization/extension

Formalize a language (Java[BJB12], x86[JBK13], C[App11], Bedrock[Chl11]) ⟶ Develop a logic (Charge![BJB12], MSL[App11], ...) ⟶ Verify programs (database[MMSW10], sha-256[App14], ...)

Reasoning in Coq

1. Math
2. Meta-theory
3. **Program verification**

Difficulties

1. Binding, validation, etc.
2. **Enriched logics**
   - State, step-indexing...
3. **Customization/extension**

Formalize a language	$\longrightarrow$	Develop a logic	$\longrightarrow$	Verify programs
			iterate	
(Java[BJB12], x86[JBK13], C[App11], Bedrock[Chl11])		(Charge![BJB12], MSL[App11], ...)		(database[MMSW10], sha-256[App14], ...)

Reasoning in Coq

1. Math
2. Meta-theory
3. **Program verification**

Difficulties

1. Binding, validation, etc.
2. Enriched logics
   - State, step-indexing...
3. Customization/extension

trust		trust
Formalize a language →	Develop a logic →	Verify programs
	iterate
(Java[BJB12], x86[JBK13], C[App11], Bedrock[Chl11])	(Charge![BJB12], MSL[App11], ...)	(database[MMSW10], sha-256[App14], ...)

Preview: Automation Requires Composition!

Ltac

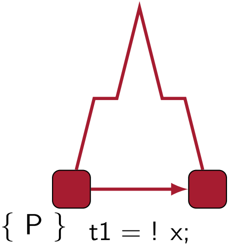
```
Ltac arith := auto with arith.
Ltac entail :=
  ... (** giant **)....
Ltac sym_eval := repeat
 first [apply read;[ solve [entail] | ]
       |apply write;[ solve [entail] | ]
       | ...  ].
```

{ P }

Preview: Automation Requires Composition!

Ltac

```
Ltac arith := auto with arith.
Ltac entail :=
  ... (** giant **)....
Ltac sym_eval := repeat
 first [apply read;[ solve [entail] | ]
       |apply write;[ solve [entail] | ]
       | ...  ].
```



Arith

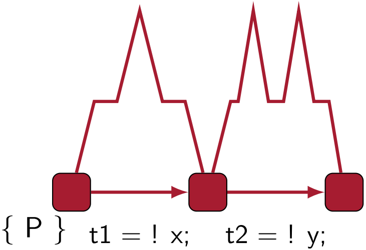
Entailment

SymEval

Preview: Automation Requires Composition!

Ltac

```
Ltac arith := auto with arith.
Ltac entail :=
  ... (** giant **)....
Ltac sym_eval := repeat
 first [apply read;[ solve [entail] | ]
      |apply write;[ solve [entail] | ]
      |  ...  ].
```



Arith

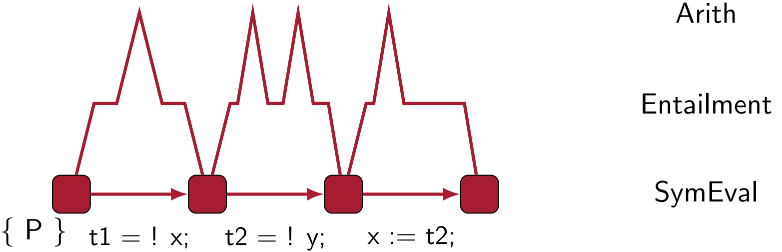
Entailment

SymEval

Preview: Automation Requires Composition!

Ltac

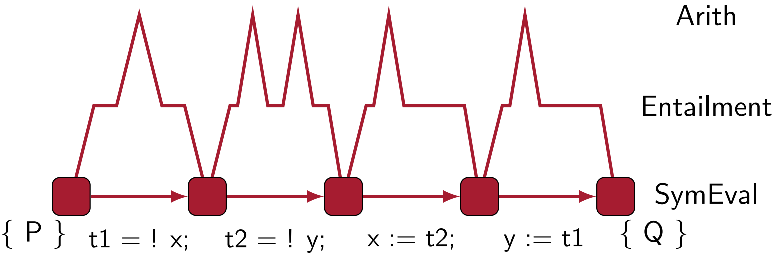
```
Ltac arith := auto with arith.
Ltac entail :=
  ... (** giant **)....
Ltac sym_eval := repeat
 first [apply read;[ solve [entail] | ]
       |apply write;[ solve [entail] | ]
       |  ...  ].
```



Preview: Automation Requires Composition!

Ltac

```
Ltac arith := auto with arith.
Ltac entail :=
  ... (** giant **)....
Ltac sym_eval := repeat
 first [apply read;[ solve [entail] | ]
       |apply write;[ solve [entail] | ]
       | ...  ].
```



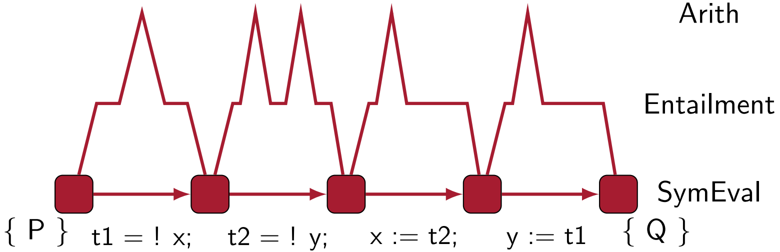
Preview: Automation Requires Composition!

Ltac

```
Ltac arith := auto with arith.
Ltac entail :=
  ... (** giant **)....
Ltac sym_eval := repeat
 first [apply read;[ solve [entail] | ]
       |apply write;[ solve [entail] | ]
       | ...  ].
```

RTac

```
Definition arith := AUTO arith_hints.
Definition entail :=
  rtac_extern (fun us vs s goal ⇒
            entailer us vs s goal arith).
Definition sym_eval := REPEAT 10
 FIRST [APPLY read_syn entail
       ; APPLY write_syn entail
       ;  ..  ].
```



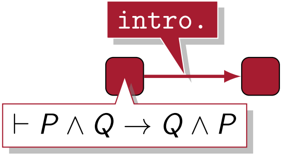
Tactic-based Logic Automation

$$\frac{\vdots}{\vdash P \wedge Q \rightarrow Q \wedge P}$$

$\vdash P \wedge Q \rightarrow Q \wedge P$

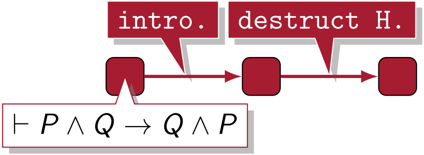
Tactic-based Logic Automation

$$\cfrac{\cfrac{\vdots}{P \wedge Q \vdash Q \wedge P}}{\vdash P \wedge Q \to Q \wedge P}\ \textsc{Intro}$$



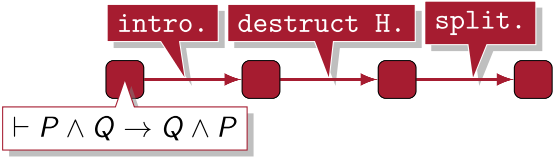
Tactic-based Logic Automation

$$
\dfrac{\dfrac{\dfrac{\vdots}{P, Q \vdash Q \land P}}{P \land Q \vdash Q \land P}\ \textsc{Destruct}}{\vdash P \land Q \to Q \land P}\ \textsc{Intro}
$$



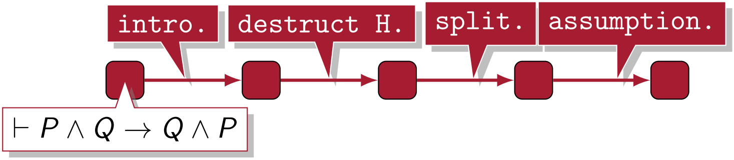
Tactic-based Logic Automation

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{P, Q \vdash P} \qquad \dfrac{\vdots}{P, Q \vdash Q}}{P, Q \vdash Q \wedge P}\ \textsc{Split}}{P \wedge Q \vdash Q \wedge P}\ \textsc{Destruct}}{\vdash P \wedge Q \to Q \wedge P}\ \textsc{Intro}$$



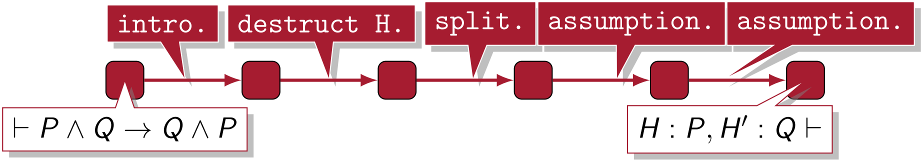
Tactic-based Logic Automation

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{P, Q \vdash P} \qquad \dfrac{}{P, Q \vdash Q}\ \textsc{Assume}}{P, Q \vdash Q \wedge P}\ \textsc{Split}}{P \wedge Q \vdash Q \wedge P}\ \textsc{Destruct}}{\vdash P \wedge Q \to Q \wedge P}\ \textsc{Intro}$$



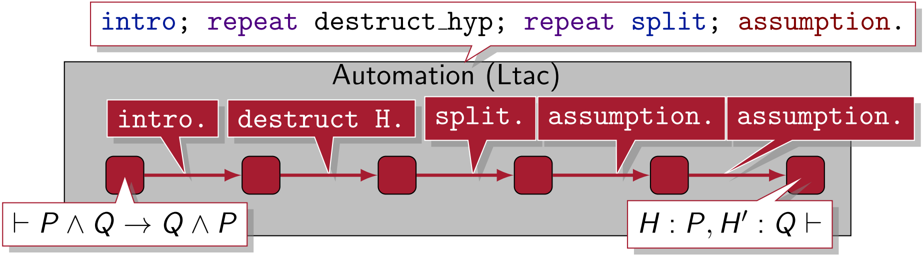
Tactic-based Logic Automation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P, Q \vdash P} \qquad \dfrac{}{P, Q \vdash Q}\ \textsc{Assume}}{P, Q \vdash Q \wedge P}\ \textsc{Split}}{P \wedge Q \vdash Q \wedge P}\ \textsc{Destruct}}{\vdash P \wedge Q \to Q \wedge P}\ \textsc{Intro}$$

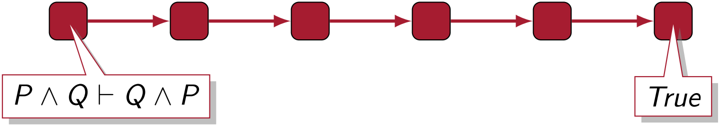


Tactic-based Logic Automation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P, Q \vdash P} \qquad \dfrac{}{P, Q \vdash Q}\ \textsc{Assume}}{P, Q \vdash Q \wedge P}\ \textsc{Split}}{P \wedge Q \vdash Q \wedge P}\ \textsc{Destruct}}{\vdash P \wedge Q \to Q \wedge P}\ \textsc{Intro}$$

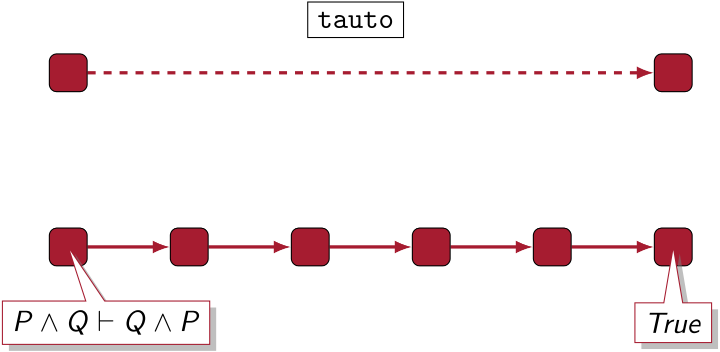


Proof by Computational Reflection [Bou97]



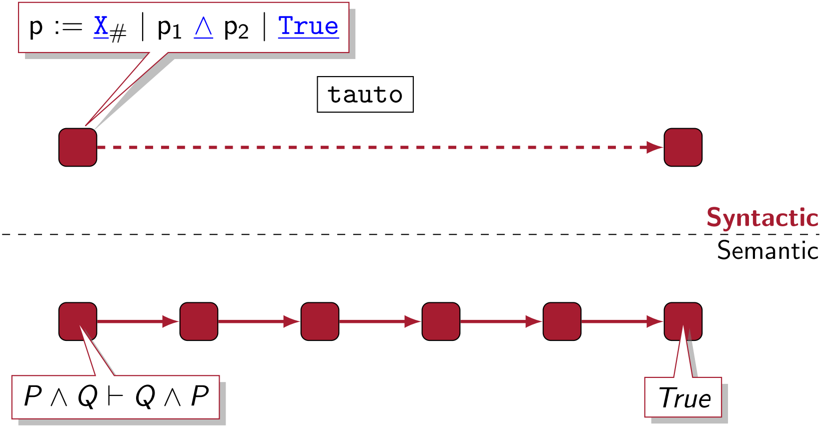
Proof by Computational Reflection [Bou97]

- Write a **procedure**



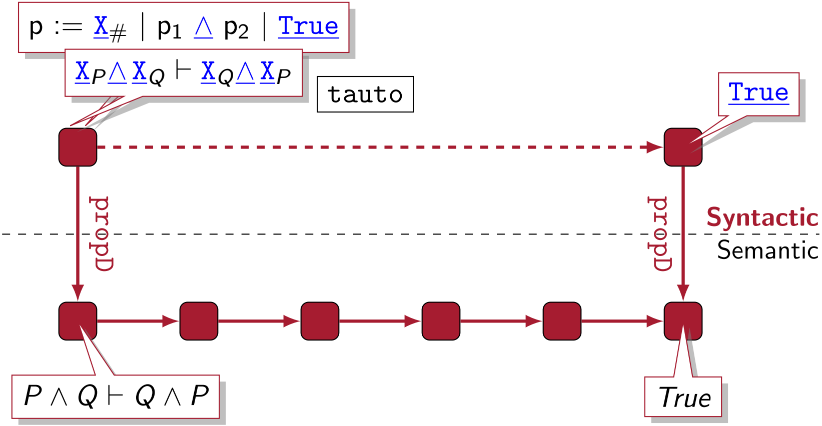
Proof by Computational Reflection [Bou97]

- Write a **procedure**
  - Pick the abstraction



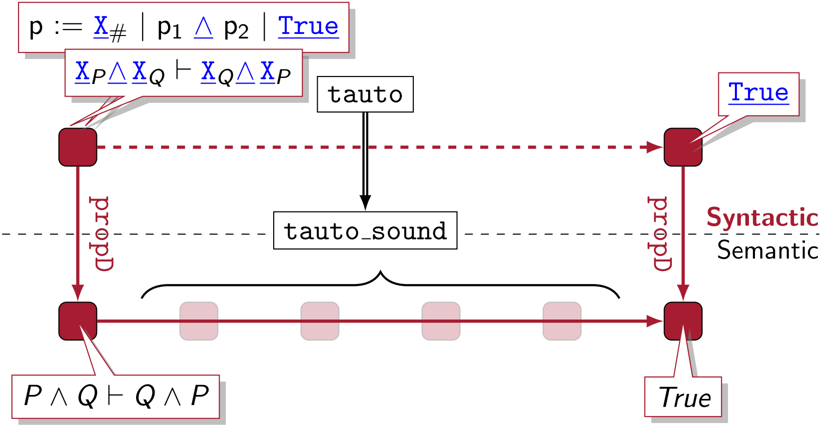
Proof by Computational Reflection [Bou97]

- Write a **procedure**
  - Pick the abstraction



Proof by Computational Reflection [Bou97]

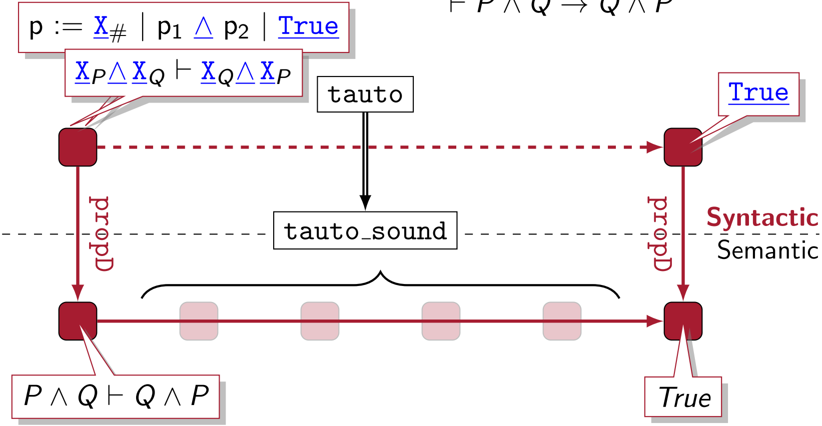
- Write a **procedure**
  - Pick the abstraction



Proof by Computational Reflection [Bou97]

- Write a **procedure**
  - Pick the abstraction
  - Small/efficient proofs

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathit{true} = \mathit{true}}\ \textsc{Refl}}{\vdash \mathtt{tauto}(\mathtt{X}_P \mathbin{\underline{\wedge}} \mathtt{X}_Q \mathbin{\underline{\to}} \mathtt{X}_Q \mathbin{\underline{\wedge}} \mathtt{X}_P) = \mathit{true}}\ \textsc{Conv}}{\vdash \mathtt{propD}(\mathtt{X}_P \mathbin{\underline{\wedge}} \mathtt{X}_Q \mathbin{\underline{\to}} \mathtt{X}_Q \mathbin{\underline{\wedge}} \mathtt{X}_P)}\ \textsc{Sound}}{\vdash P \wedge Q \to Q \wedge P}\ \textsc{Conv}
$$



A Simple Tautology Solver

(1) Define syntax.

$\mathtt{Ind\ prop} = \underline{\mathtt{True}} \mid \mathtt{p}\ \underline{\wedge}\ \mathtt{q} \mid \underline{\mathtt{X}}_{\#}$

A Simple Tautology Solver

(1) Define syntax.

```
Ind prop = True | p ∧ q | X#
```

(2) Define meaning.

```
Fix propD (ps : env Prop) (p : prop) : Prop
:= match p with
     True ⇒ True
   | p ∧ q ⇒
    propD ps p ∧ propD ps q
   | Xp ⇒ lookup ps p
```

A Simple Tautology Solver

(1) Define syntax.

```
Ind prop = True | p ∧ q | X#
```

(2) Define meaning.

```
Fix propD (ps : env Prop) (p : prop) : Prop
:= match p with
     True ⇒ True
   | p ∧ q ⇒
    propD ps p ∧ propD ps q
   | Xp ⇒ lookup ps p
```

```
Def prove hyps goal : bool
:= match goal with
     True ⇒ true
   | p∧q ⇒
     prove hyps p && prove hyps q
   | _ ⇒
         find_assumption hyps goal
```

(3) Write a procedure.

A Simple Tautology Solver

(1) Define syntax.

```
Ind prop = True | p ∧ q | X_#
```

(2) Define meaning.

```
Fix propD (ps : env Prop) (p : prop) : Prop
:= match p with
     True ⇒ True
   | p ∧ q ⇒
    propD ps p ∧ propD ps q
   | X_p ⇒ lookup ps p
```

```
Def prove hyps goal : bool
:= match goal with
     True ⇒ true
   | p∧q ⇒
     prove hyps p && prove hyps q
   | _ ⇒
       find_assumption hyps goal
```

(3) Write a procedure.

```
Theorem prove_sound : ∀ ps hs goal,
  prove hs goal = true →
  All (propD ps) hs ⊢
  propD ps goal.
Proof. ...  Qed.
```

(4) Prove the procedure.

A Generic Tautology Solver

```
Ind prop = True | p △ q | X#
```

```
Def prove hyps goal : bool
:= match goal with
     True ⇒ true
   | p△q ⇒
     prove hyps p && prove hyps q
   | _ ⇒
         find_assumption hyps goal
```

Abstract Prop

```
Fix propD (ps : env L) (p : prop) : L
:= match p with
     True ⇒ True
   | p △ q ⇒
    propD ps p ∧ propD ps q
   | Xp ⇒ lookup ps p
```

```
Theorem prove_sound : ∀ ps hs goal,
  prove hs goal = true →
  All (propD ps) hs ⊢
  propD ps goal.
Proof. ... Qed.
```

Similar proof

A Generic Tautology Solver

- Extensions
  - $\triangleright$ – Modalities
  - $*$, $-\!\!*$ – Connectives
  - $\mapsto$, llist – Predicates

```
Ind prop = True | p ∧ q | X#
```

```
Def prove hyps goal : bool
:= match goal with
     True ⇒ true
   | p∧q ⇒
     prove hyps p && prove hyps q
   | _ ⇒
        find_assumption hyps goal
     || solve_by_extern hyps goal
```

```
Fix propD (ps : env L) (p : prop) : L
:= match p with
     True ⇒ True
   | p ∧ q ⇒
     propD ps p ∧ propD ps q
   | Xp ⇒ lookup ps p
```

```
Theorem prove_sound : ∀ ps hs goal,
  prove hs goal = true →
  All (propD ps) hs ⊢
  propD ps goal.
Proof. ...  Qed.
```

An Extensible Tautology Solver?

$$\frac{}{x \in y \vdash x \in z \cup y}\text{Add-Union} \qquad \frac{}{a :: b = c :: d \vdash a = c}\text{Cons-Inj}$$

- Not everything is a tautology

```
Ind prop = True | p ∧ q | X#
```

```
Def prove hyps goal : bool
:= match goal with
     True ⇒ true
   | p∧q ⇒
     prove hyps p && prove hyps q
   | _ ⇒
         find_assumption hyps goal
     || solve_by_extern hyps goal
```

```
Fix propD (ps : env L) (p : prop) : L
:= match p with
     True ⇒ True
   | p ∧ q ⇒
     propD ps p ∧ propD ps q
   | Xp ⇒ lookup ps p
```

```
Theorem prove_sound : ∀ ps hs goal,
  prove hs goal = true →
  All (propD ps) hs ⊢
  propD ps goal.
Proof. ... Qed.
```

An Extensible Tautology Solver?

$$\frac{}{x \in y \vdash x \in z \cup y}\text{Add-Union} \qquad \frac{}{a :: b = c :: d \vdash a = c}\text{Cons-Inj}$$

- Not everything is a tautology

```
Ind prop = True | p ∧ q | X#
```

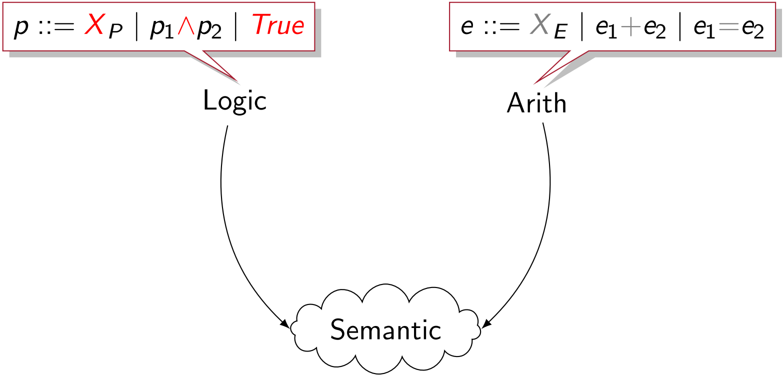
```
Def prove hyps goal : bool
:= match goal with
     True ⇒ true
   | p∧q ⇒
     prove hyps p && prove hyps q
   | _ ⇒
         find_assumption hyps goal
     || solve_by_extern hyps goal
```

```
Fix propD (ps : env L) (p : prop) : L
:= match p with
     True ⇒ True
   | p ∧ q ⇒
    propD ps p ∧ propD ps q
   | Xp ⇒ lookup ps p
```

```
Theorem prove_sound : ∀ ps hs goal,
  prove hs goal = true →
  All (propD ps) hs ⊢
  propD ps goal.
Proof. ... Qed.
```

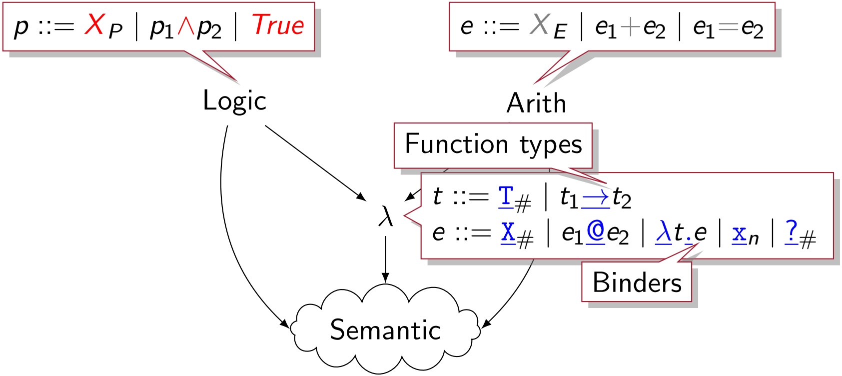
Enriching the Syntax

- STLC + base **types** & terms



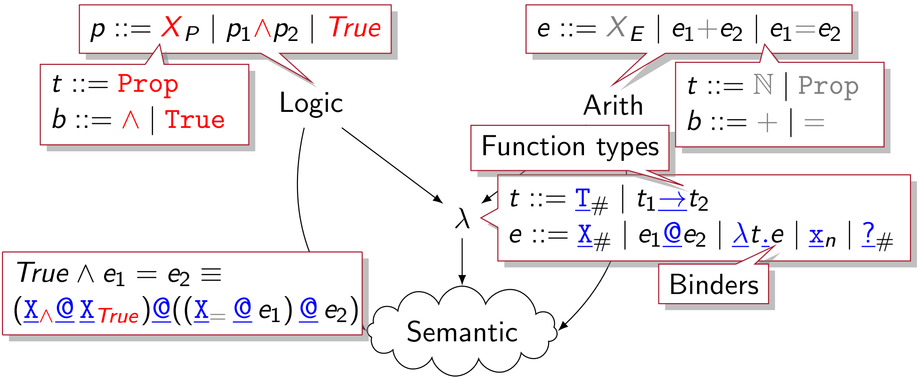
Enriching the Syntax

- STLC + base **types** & terms



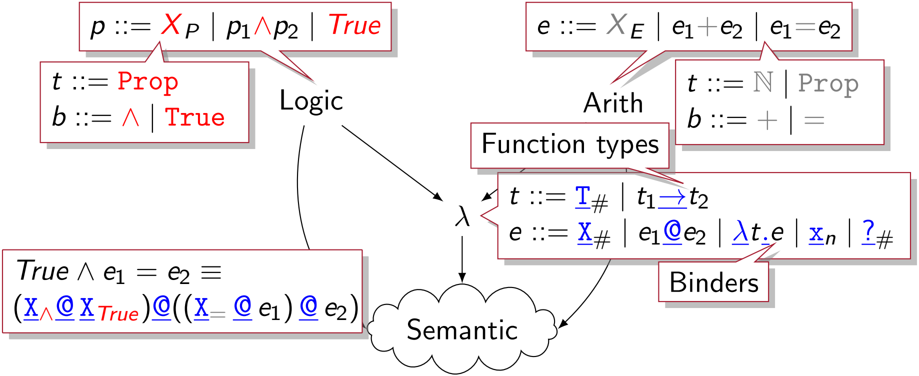
Enriching the Syntax

- STLC + base **types** & terms



Enriching the Syntax, Compositionally

- STLC + base **types** & terms
  - Compositional (see [MCB14])



An Extensible Tautology Solver

$$\frac{}{x \in y \vdash x \in z \cup y}\ \text{Add-Union} \qquad \frac{}{a :: b = c :: d \vdash a = c}\ \text{Cons-Inj}$$

- Not everything is a tautology

```
Ind typ = t₁ → t₂ | T#
Ind expr = e₁ @ e₂ | λ t . e | X# | ...
```

```
Fix exprD (ps : env L) (p : expr) : L
:= match p with
     Xp ⇒ lookup ps p
   | p ∧ q ⇒
     exprD ps p ∧ exprD ps q
   | ... ⇒ ...
```

```
Def prove hyps goal : bool
:= match goal with
     X_True ⇒ true
   | X∧@p@q ⇒
     prove hyps p && prove hyps q
   | _ ⇒
       find_assumption hyps goal
     || solve_by_extern hyps goal
```

```
Theorem prove_sound : ∀ ps hs goal,
  prove hs goal = true →
  All (propD ps) hs ⊢
  propD ps goal.
Proof. ... Qed.
```

An Extensible Tautology Solver

$$\frac{}{x \in y \vdash x \in z \cup y}\ \text{Add-Union} \qquad \frac{}{a :: b = c :: d \vdash a = c}\ \text{Cons-Inj}$$

- Not everything is a tautology

```
Ind typ = t₁ → t₂ | T#
Ind expr = e₁ @ e₂ | λ t . e | X# | …
```

```
Def prove hyps goal : bool
:= match goal with
     X_True ⇒ true
   | X∧@p@q ⇒
     prove hyps p && prove hyps q
   | _ ⇒
       find_assumption hyps goal
     || solve_by_extern hyps goal
```

```
Fix exprD (ps : env L) (p : expr) : L
:= match p with
     X_p ⇒ lookup ps p
   | p ∧ q ⇒
     exprD ps p ∧ exprD ps q
   | … ⇒ …
```

```
Theorem prove_sound : ∀ ps hs goal,
  prove hs goal = true →
  All (propD ps) hs ⊢
  propD ps goal.
Proof. … Qed.
```

Polymorphic over (contrained) extensions[MCB14]

An Extensible Tautology Solver

$$\frac{}{x \in y \vdash x \in z \cup y}\ \text{Add-Union} \qquad \frac{}{a :: b = c :: d \vdash a = c}\ \text{Cons-Inj}$$

- Not everything is a tautology

```
Ind typ = t1 → t2 | T#
Ind expr = e1 @ e2 | λ t . e | X# | ...
```

```
Def prove hyps goal : bool
:= match goal with
     X_True ⇒ true
   | X∧@p@q ⇒
     prove hyps p && prove hyps q
   | _ ⇒
        find_assumption hyps goal
     || solve_by_extern hyps goal
```

Too much effort to write!

```
Fix exprD (ps : env L) (p : expr) : L
:= match p with
     X_p ⇒ lookup ps p
   | p ∧ q ⇒
     exprD ps p ∧ exprD ps q
   | ... ⇒ ...
```

```
Theorem prove_sound : ∀ ps hs goal,
  prove hs goal = true →
  All (propD ps) hs ⊢
  propD ps goal.
Proof. ... Qed.
```

Assembling Custom Automation

Use these reflectively

```
Lem list_eq : ∀ x y xs ys,
  x = y → xs = ys → x :: xs = y :: ys.
Lem list_len : ∀ xs ys,
  |xs ∪ ys| = |xs| + |ys|.
Lem add_in : ∀ x y z,
  x ∈ z → x ∈ (y ∪ z).
```

Assembling Custom Automation

Use these reflectively

```
Lem list_eq : ∀ x y xs ys,
  x = y → xs = ys → x :: xs = y :: ys.
Lem list_len : ∀ xs ys,
  |xs ∪ ys| = |xs| + |ys|.
Lem add_in : ∀ x y z,
  x ∈ z → x ∈ (y ∪ z).
```

```
Def prove (g : expr) : bool :=
 match g with
 | (@ (@ X∈ e) (@ (@ X∪ s1) s2)) ⇒
   prove (@ (@ X∈ e) s1)
 | (X=[ℕ] @ (X:: @ x @ xs) (X:: @ y @ ys) ⇒
   prove (X=ℕ @ x @ y) &&
   prove (X=[ℕ] @ xs @ ys)
 | ...  ⇒ ...

Thm prove_sound : ∀ ts fs ...
```

Assembling Custom Automation

Build reflective procedures automatically from lemmas.

Reflective lemma "application"

1. Unification & substitutions

```
Lem list_eq : ∀ x y xs ys,
  x = y → xs = ys → x :: xs = y :: ys.
Lem list_len : ∀ xs ys,
  |xs ∪ ys| = |xs| + |ys|.
Lem add_in : ∀ x y z,
  x ∈ z → x ∈ (y ∪ z).
```

Bound variables

```
Def prove (g : expr) : bool :=
 match g with
 | (@ (@ X_∈ e) (@ (@ X_∪ s1) s2)) ⇒
   prove (@ (@ X_∈ e) s1)
 | (X_=[ℕ] @ (X_:: @ x @ xs) (X_:: @ y @ ys) ⇒
   prove (X_=ℕ @ x @ y) &&
   prove (X_=[ℕ] @ xs @ ys)
 | ... ⇒ ...

Thm prove_sound : ∀ ts fs ...
```

Lemmas, External Hints & Hint Databases

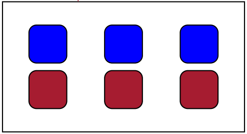
Build reflective procedures automatically from lemmas.

Reflective lemma "application"

1. Unification & substitutions
2. Semantic reasoning

```
Lem list_eq : ∀ x y xs ys,
  x = y → xs = ys → x :: xs = y :: ys.
Lem list_len : ∀ xs ys,
  |xs ∪ ys| = |xs| + |ys|.
Lem add_in : ∀ x y z,
  x ∈ z → x ∈ (y ∪ z).
```

Built automatically



Lemmas, External Hints & Hint Databases

Build reflective procedures automatically from lemmas.

Reflective lemma "application"

1. Unification & substitutions
2. Semantic reasoning

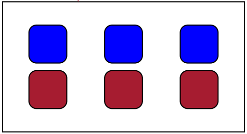
Parameterized automation

- auto
- autorewrite

```
Lem list_eq : ∀ x y xs ys,
  x = y → xs = ys → x :: xs = y :: ys.
Lem list_len : ∀ xs ys,
  |xs ∪ ys| = |xs| + |ys|.
Lem add_in : ∀ x y z,
  x ∈ z → x ∈ (y ∪ z).
```

`auto_prove`

`auto_sound`



Generic, reflective proof search!

(Semantic) Unification

- Applying lemmas generically requires unification

$$\vdash (\mathtt{f}\ \square_1\ \mathtt{y}) \sim (\mathtt{f}\ \mathtt{x}\ \square_2) \hookrightarrow \{\square_1 \mapsto \mathtt{x}\ ,\ \square_2 \mapsto \mathtt{y}\}$$

(Semantic) Unification = Equality (e)Prover

$$\texttt{unify} : \texttt{expr} \rightarrow \texttt{expr} \rightarrow \texttt{subst} \rightarrow \texttt{option subst}$$

- Applying lemmas generically requires unification

$$\vdash (\texttt{f}\ \square_1\ \texttt{y}) \sim (\texttt{f}\ \texttt{x}\ \square_2) \hookrightarrow \{\square_1 \mapsto \texttt{x}\ ,\ \square_2 \mapsto \texttt{y}\}$$

(Semantic) Unification = Equality (e)Prover

$$\texttt{unify} : \texttt{expr} \rightarrow \texttt{expr} \rightarrow \texttt{subst} \rightarrow \texttt{option subst}$$

- Applying lemmas generically requires unification

$$\vdash (\texttt{f}\ \square_1\ \texttt{y}) \sim (\texttt{f}\ \texttt{x}\ \square_2) \hookrightarrow \{\square_1 \mapsto \texttt{x}\ ,\ \square_2 \mapsto \texttt{y}\}$$

- **Programmable** – Coq's native unification is fixed

$$\vdash (\texttt{x} + \texttt{y}) \sim (\texttt{y} + \texttt{x}) \hookrightarrow \{\}$$

$$\texttt{x} = \texttt{y} \vdash (\texttt{f}\ \texttt{x}) \sim (\texttt{f}\ \texttt{y}) \hookrightarrow \{\}$$

(Semantic) Unification = Equality (e)Prover

Typed

$$\texttt{unify} : \texttt{expr} \to \texttt{expr} \to \textbf{typ} \to \texttt{subst} \to \texttt{option subst}$$

- Applying lemmas generically requires unification

$$\vdash (\texttt{f}\ \square_1\ \texttt{y}) \sim_\tau (\texttt{f}\ \texttt{x}\ \square_2) \hookrightarrow \{\square_1 \mapsto \texttt{x}\ ,\ \square_2 \mapsto \texttt{y}\}$$

- **Programmable** – Coq's native unification is fixed

$$\vdash (\texttt{x} + \texttt{y}) \sim_{\mathbf{N}} (\texttt{y} + \texttt{x}) \hookrightarrow \{\}$$

$$\texttt{x} = \texttt{y} \vdash (\texttt{f}\ \texttt{x}) \sim_\tau (\texttt{f}\ \texttt{y}) \hookrightarrow \{\}$$

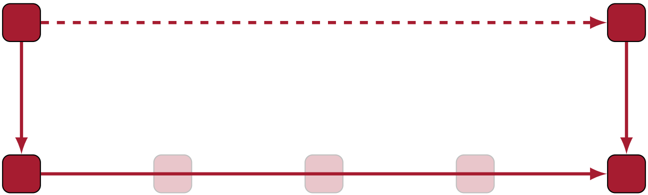
- **Typed** – Coq's native unification is untyped

$$\vdash () \sim_{\texttt{unit}} \texttt{x} \hookrightarrow \{\}$$

(Simple) Reflective Tactics

auto is somewhat limited

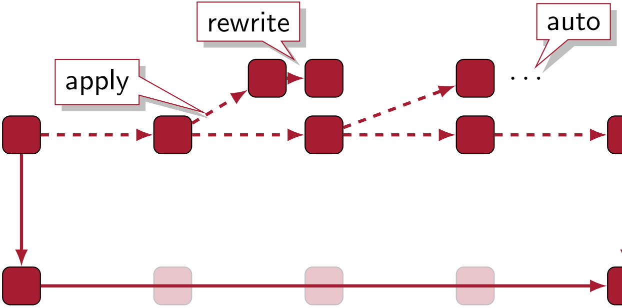
- Backward reasoning from a goal
- Limited ability to customize proof search
- Must solve the goal entirely



(Simple) Reflective Tactics

auto is somewhat limited

- Backward reasoning from a goal
- Limited ability to customize proof search
- Must solve the goal entirely

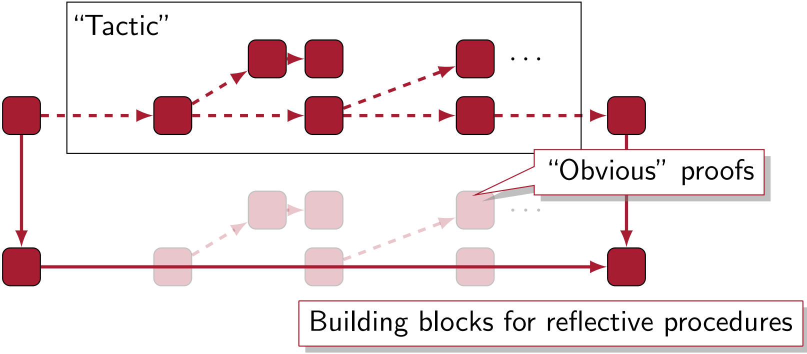


Building blocks for reflective procedures

(Simple) Reflective Tactics

auto is somewhat limited

- Backward reasoning from a goal
- Limited ability to customize proof search
- Must solve the goal entirely



RTac: Reflective Tactics

Ltac

```
repeat first [ apply read ; auto
             | apply write ; auto
             | ...  ]
```

RTac

```
Def the_tac db := REPEAT 10
  (FIRST [ APPLY read_syn (AUTO db)
         | APPLY write_syn (AUTO db)
         | ...  ]) .
```

RTac: Reflective Tactics

Ltac

```
repeat first [ apply read ; auto
             | apply write ; auto
             | ...  ]
```

RTac

```
Def the_tac db := REPEAT 10
  (FIRST [ APPLY read_syn (AUTO db)
         | APPLY write_syn (AUTO db)
         | ...  ]).

Thm the_tac_sound db : hints_sound db
    → rtac_sound the_tac.
Proof. intro.
  apply REPEAT_sound.
  apply FIRST_sound.
  + apply APPLY_sound;
    [exact read |apply AUTO_sound; auto ].
  + apply APPLY_sound;
    [exact write |apply AUTO_sound; auto
     ].
  + ...
Qed.
```

Soundness is a predicate transformer

RTac: Reflective Tactics

Ltac

```
repeat first [ apply read ; auto
             | apply write ; auto
             | ...  ]
```

- Simple proofs
- Reflective (separate proofs)
- Stable across Coq versions
- Shallow embedding → extensible

Easy to use custom procedures

RTac

```
Def the_tac db := REPEAT 10
  (FIRST [ APPLY read_syn (AUTO db)
         | APPLY write_syn (AUTO db)
         | ...  ]) .

Thm the_tac_sound db : hints_sound db
    → rtac_sound the_tac.
Proof. intro.
  apply REPEAT_sound.
  apply FIRST_sound.
  + apply APPLY_sound;
    [exact read |apply AUTO_sound; auto ].
  + apply APPLY_sound;
    [exact write |apply AUTO_sound; auto
     ].
  + ...
Qed.
```

Soundness is a predicate transformer

Require Import MirrorCore[1]

Next time you need **customizable**, **reflective** automation...

1

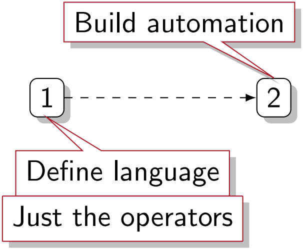
Define language

Just the operators

[1]Refactoring in progress

Require Import MirrorCore[1]

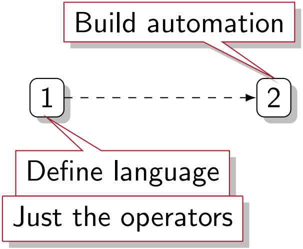
Next time you need **customizable**, **reflective** automation...



[1] Refactoring in progress

Require Import MirrorCore[1]

Next time you need **customizable**, **reflective** automation...



1) Existing generic automation

2) Extend automation (auto)

3) RTac "tactics"

4) Custom procedures

[1] Refactoring in progress

Require Import MirrorCore[1]

Next time you need **customizable**, **reflective** automation...

Build automation

1 - - - - - - - - - - → 2

Define language

Just the operators

1) Existing generic automation

2) Extend automation (auto)

3) RTac "tactics"

4) Custom procedures

Folds Lifting Unification
Substitution ...

Reuse the meta-theory & tactics

[1]Refactoring in progress

Related Work

- "Intensional" Theories (e.g. Coq, Agda)
  1. Simple Types [GW07] – Similar term representation
  2. Modular Meta Theory [DdSOS13]
  3. AAC Tactics, ROmega, field, ring [BP11, GM05, Les11] – reflective procedures
  4. Posteriori Simulation [CCGHRGZ13] – Faster computation
  5. Mtac [ZDK$^+$13] – Coq extension (proof-generating)
  6. SSreflect [GM10] – Coq library (proof-generating)

Related Work

- "Intensional" Theories (e.g. Coq, Agda)
  1. Simple Types [GW07] – Similar term representation
  2. Modular Meta Theory [DdSOS13]
  3. AAC Tactics, ROmega, field, ring [BP11, GM05, Les11] – reflective procedures
  4. Posteriori Simulation [CCGHRGZ13] – Faster computation
  5. Mtac [ZDK+13] – Coq extension (proof-generating)
  6. SSreflect [GM10] – Coq library (proof-generating)

- "Extensional" Theories
  1. VeriML [SS10], NuPrl
  2. LF

Related Work

- "Intensional" Theories (e.g. Coq, Agda)
  1. Simple Types [GW07] – Similar term representation
  2. Modular Meta Theory [DdSOS13]
  3. AAC Tactics, ROmega, field, ring [BP11, GM05, Les11] – reflective procedures
  4. Posteriori Simulation [CCGHRGZ13] – Faster computation
  5. Mtac [ZDK$^+$13] – Coq extension (proof-generating)
  6. SSreflect [GM10] – Coq library (proof-generating)
- "Extensional" Theories
  1. VeriML [SS10], NuPrl
  2. LF
- Non-dependent Theories
  - Isabelle, HOL, ...

MirrorCore

- Generic logic automation (i.e. for lifted logics)
- Extensible syntax (e.g. user-defined types and functions)
- Extensible automation (e.g. auto, autorewrite)
- Simple tactic language

`https://github.com/gmalecha/mirror-core`
`https://github.com/jesper-bengtson/MirrorCharge`

$$\begin{tikzcd} \text{Formalize a language} \arrow[r] & \text{Develop a logic} \arrow[r] \arrow[l, bend right, "\text{trust}"'] & \text{Verify programs} \arrow[l, bend right, "\text{trust}"'] \end{tikzcd}$$

References I

Andrew W. Appel.
Verified software toolchain.
In *Proc. ESOP*, volume 6602 of *LNCS*, pages 1–17. Springer-Verlag, 2011.

Andrew W. Appel.
Verification of a cryptographic primitive: Sha-256, May 2014.

Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal.
Charge! – a framework for higher-order separation logic in Coq.
In *Interactive Theorem Proving*, pages 315–331, 2012.

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

Thomas Braibant and Damien Pous.
Tactics for reasoning modulo AC in Coq.
In *Proc. CPP*, 2011.

Guillaume Claret, Lourdes Carmen Gonzlez Huesca, Yann Rgis-Gianas, and Beta Ziliani.
Lightweight proof by reflection using a posteriori simulation of effectful computation.
In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 67–83. Springer Berlin Heidelberg, 2013.

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

References II

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

B. Grégoire and A. Mahboubi.
Proving equalities in a commutative ring done right in Coq.
In *Proc. TPHOLs*, 2005.

Georges Gonthier and Assia Mahboubi.
An introduction to small scale reflection in Coq.
Journal of Formalized Reasoning, 3(2):95–152, 2010.

Franois Garillot and Benjamin Werner.
Simple types in type theory: Deep and shallow encodings.
In *Theorem Proving in Higher Order Logics*, volume 4732 of *LNCS*, pages 368–382. Springer Berlin Heidelberg, 2007.

Jonas B. Jensen, Nick Benton, and Andrew Kennedy.
High-level separation logic for low-level code.
In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 301–314, New York, NY, USA, 2013. ACM.

Stéphane Lescuyer.
Formalisation et développement d'une tactique réflexive pour la démonstration automatique en Coq.
Thèse de doctorat, Université Paris-Sud, January 2011.

Gregory Malecha, Adam Chlipala, and Thomas Braibant.
Compositional computational reflection.
In *Interactive Theorem Proving*, 2014.
To Appear.

References III

Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky.
Toward a verified relational database management system.
In *POPL'10*, January 2010.

Antonis Stampoulis and Zhong Shao.
VeriML: typed computation of logical terms inside a language with effects.
In *Proc. ICFP*, pages 333–344. ACM, 2010.

Beta Ziliani, Derek Dreyer, Neel Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis.
Mtac: A monad for typed tactic programming in Coq.
In *Proc. ICFP*, 2013.