Compositional Computational Reflection

Gregory Malecha Adam Chlipala Thomas Braibant
gmalecha@cs.harvard.edu

Harvard SEAS MIT CSAIL Inria

July 17, 2014

Program Verification in Bedrock [Chl11]

Imperative Program

```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
[∀ ls, PRE[V] sll ls (V "x")
      POST[R] ⌈ R = V "n" + length ls ⌉
            ∗ sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ← ∗"x"
  };;
  Return "n"
```

Hints / Theorems

```
Def sll : list W → W → HProp := ...

Thm nil_fwd : ∀ ls (p : W), p = 0
→ sll ls p ⊢ ⌈ ls = nil ⌉.
Proof. .. Qed.

Thm cons_fwd : ∀ ls (p : W), p ≠ 0
→ sll ls p ⊢
   ∃ x, ∃ ls', ⌈ ls = x :: ls' ⌉ ∗
   ∃ p', p ↦ (x, p') ∗ sll ls' p'.
Proof. .. Qed.
```

```
Thm sllMOk : moduleOk sllM.
Proof. vcgen; abstract (sep hints; finish). Qed.
```

Program Verification in Bedrock [Chl11]

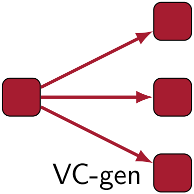
Imperative Program

```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
[∀ ls, PRE[V] sll ls (V "x")
      POST[R] ⌈ R = V "n" + length ls ⌉
            * sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ← *"x"
  };;
  Return "n"
```

Hints / Theorems

```
Def sll : list W → W → HProp := ...

Thm nil_fwd : ∀ ls (p : W), p = 0
→ sll ls p ⊢ ⌈ ls = nil ⌉.
Proof. .. Qed.

Thm cons_fwd : ∀ ls (p : W), p ≠ 0
→ sll ls p ⊢
   ∃ x, ∃ ls', ⌈ ls = x :: ls' ⌉ *
   ∃ p', p ↦ (x, p') * sll ls' p'.
Proof. .. Qed.
```

```
Thm sllMOk : moduleOk sllM.
Proof. vcgen; abstract (sep hints; finish). Qed.
```



Program Verification in BEDROCK [Chl11]

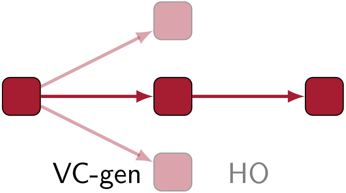
Imperative Program

```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
[∀ ls, PRE[V] sll ls (V "x")
      POST[R] ⌈ R = V "n" + length ls ⌉
             * sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ← *"x"
  };;
  Return "n"
```

Hints / Theorems

```
Def sll : list W → W → HProp := ...

Thm nil_fwd : ∀ ls (p : W), p = 0
→ sll ls p ⊢ ⌈ ls = nil ⌉.
Proof. .. Qed.

Thm cons_fwd : ∀ ls (p : W), p ≠ 0
→ sll ls p ⊢
   ∃ x, ∃ ls', ⌈ ls = x :: ls' ⌉ *
   ∃ p', p ↦ (x, p') * sll ls' p'.
Proof. .. Qed.
```

```
Thm sllMOk : moduleOk sllM.
Proof. vcgen; abstract (sep hints; finish). Qed.
```



VC-gen

HO

Program Verification in BEDROCK [Chl11]

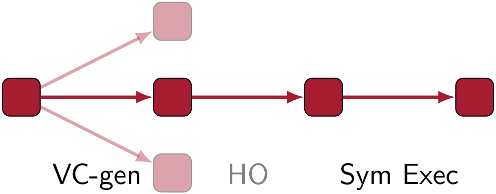
Imperative Program

```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
[∀ ls, PRE[V] sll ls (V "x")
      POST[R] ⌈ R = V "n" + length ls ⌉
                 * sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ← *"x"
  };;
  Return "n"
```

Hints / Theorems

```
Def sll : list W → W → HProp := ...

Thm nil_fwd : ∀ ls (p : W), p = 0
→ sll ls p ⊢ ⌈ ls = nil ⌉.
Proof. .. Qed.

Thm cons_fwd : ∀ ls (p : W), p ≠ 0
→ sll ls p ⊢
   ∃ x, ∃ ls', ⌈ ls = x :: ls' ⌉ *
   ∃ p', p ↦ (x, p') * sll ls' p'.
Proof. .. Qed.
```

```
Thm sllMOk : moduleOk sllM.
Proof. vcgen; abstract (sep hints; finish). Qed.
```



Program Verification in BEDROCK [Chl11]

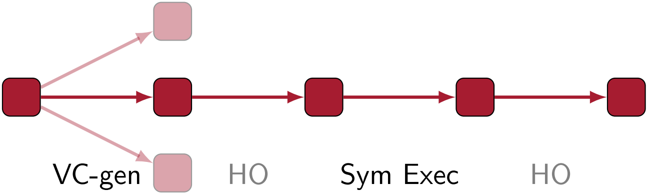
Imperative Program

```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
[∀ ls, PRE[V] sll ls (V "x")
      POST[R] ⌈ R = V "n" + length ls ⌉
              ∗ sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ← ∗"x"
  };;
  Return "n"
```

Hints / Theorems

```
Def sll : list W → W → HProp := ...

Thm nil_fwd : ∀ ls (p : W), p = 0
→ sll ls p ⊢ ⌈ ls = nil ⌉.
Proof. .. Qed.

Thm cons_fwd : ∀ ls (p : W), p ≠ 0
→ sll ls p ⊢
   ∃ x, ∃ ls', ⌈ ls = x :: ls' ⌉ ∗
   ∃ p', p ↦ (x, p') ∗ sll ls' p'.
Proof. .. Qed.
```

```
Thm sllMOk : moduleOk sllM.
Proof. vcgen; abstract (sep hints; finish). Qed.
```



Program Verification in Bedrock [Chl11]

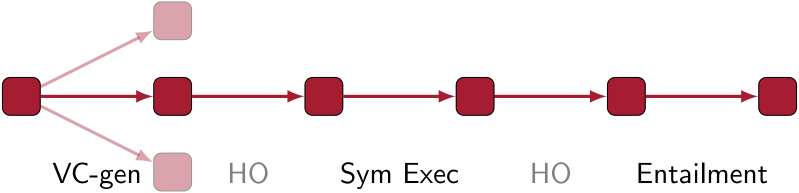
Imperative Program

```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
[∀ ls, PRE[V] sll ls (V "x")
       POST[R] ⌈ R = V "n" + length ls ⌉
              * sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ← *"x"
  };;
  Return "n"
```

Hints / Theorems

```
Def sll : list W → W → HProp := ...

Thm nil_fwd : ∀ ls (p : W), p = 0
→ sll ls p ⊢ ⌈ ls = nil ⌉.
Proof. .. Qed.

Thm cons_fwd : ∀ ls (p : W), p ≠ 0
→ sll ls p ⊢
   ∃ x, ∃ ls', ⌈ ls = x :: ls' ⌉ *
   ∃ p', p ↦ (x, p') * sll ls' p'.
Proof. .. Qed.
```

```
Thm sllMOk : moduleOk sllM.
Proof. vcgen; abstract (sep hints; finish). Qed.
```



Program Verification in BEDROCK [Chl11]

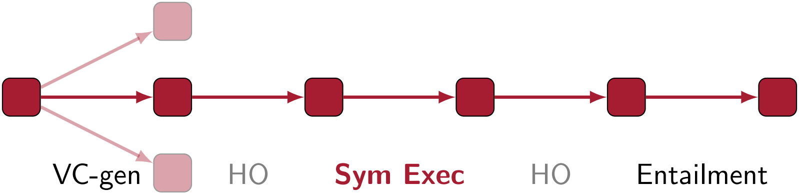
Imperative Program

```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
[∀ ls, PRE[V] sll ls (V "x")
      POST[R] ⌈ R = V "n" + length ls ⌉
             * sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ← *"x"
  };;
  Return "n"
```

Hints / Theorems

```
Def sll : list W → W → HProp := ...

Thm nil_fwd : ∀ ls (p : W), p = 0
→ sll ls p ⊢ ⌈ ls = nil ⌉.
Proof. .. Qed.

Thm cons_fwd : ∀ ls (p : W), p ≠ 0
→ sll ls p ⊢
   ∃ x, ∃ ls', ⌈ ls = x :: ls' ⌉ *
   ∃ p', p ↦ (x, p') * sll ls' p'.
Proof. .. Qed.
```

```
Thm sllMOk : moduleOk sllM.
Proof. vcgen; abstract (sep hints; finish). Qed.
```



Ltac-based Symbolic Execution

Coq's tactic language

```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
  [∀ ls,
   PRE[V] sll ls (V "x")
   POST[R] ⌈ R = V "n" + length ls ⌉
          ∗ sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ← ∗ "x"
  };;
  Return "n"
```

Ltac Automation

```
Ltac sym_eval :=
  repeat first
    [ eapply step_read ; side_condition
    | ...
    | autorewrite with lemmas ].
```

$$\frac{}{\{P\}c_1;c_2;c_3;c_4\{R\}}$$

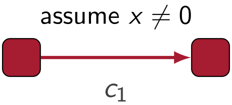
Ltac-based Symbolic Execution

```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
  [∀ ls,
   PRE[V] sll ls (V "x")
   POST[R] ⌈ R = V "n" + length ls ⌉
         ∗ sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ← ∗ "x"
  };;
  Return "n"
```

Ltac Automation

```
Ltac sym_eval :=
  repeat first
    [ eapply step_read ; side_condition
    | ...
    | autorewrite with lemmas ].
```

$$\frac{\overline{\{P'\}c_2; c_3; c_4\{R\} \quad \ldots}}{\{P\}c_1; c_2; c_3; c_4\{R\}}$$



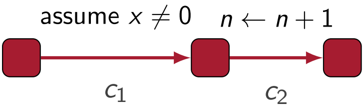
Ltac-based Symbolic Execution

```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
  [∀ ls,
   PRE[V] sll ls (V "x")
   POST[R] ⌈ R = V "n" + length ls ⌉
         ∗ sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ← ∗ "x"
  };;
  Return "n"
```

Ltac Automation

```
Ltac sym_eval :=
  repeat first
    [ eapply step_read ; side_condition
    | ...
    | autorewrite with lemmas ].
```

$$\cfrac{\cfrac{\{P''\}c_3; c_4\{R\} \quad \ldots}{\{P'\}c_2; c_3; c_4\{R\} \quad \ldots}}{\{P\}c_1; c_2; c_3; c_4\{R\}}$$



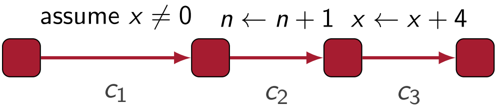
Ltac-based Symbolic Execution

```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
  [∀ ls,
   PRE[V] sll ls (V "x")
   POST[R] ⌈ R = V "n" + length ls ⌉
         * sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ← * "x"
  };;
  Return "n"
```

Ltac Automation

```
Ltac sym_eval :=
  repeat first
    [ eapply step_read ; side_condition
    | ...
    | autorewrite with lemmas ].
```

$$\cfrac{\cfrac{\cfrac{\overline{\{P'''\}c_4\{R\} \quad \ldots}}{\{P''\}c_3; c_4\{R\} \quad \ldots}}{\{P'\}c_2; c_3; c_4\{R\} \quad \ldots}}{\{P\}c_1; c_2; c_3; c_4\{R\}}$$



Ltac-based Symbolic Execution

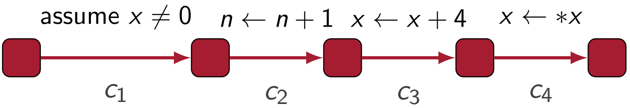
```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
  [∀ ls,
   PRE[V] sll ls (V "x")
   POST[R] ⌈ R = V "n" + length ls ⌉
         ∗ sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ← ∗ "x"
  };;
  Return "n"
```

Ltac Automation

```
Ltac sym_eval :=
  repeat first
    [ eapply step_read ; side_condition
    | ...
    | autorewrite with lemmas ].
```

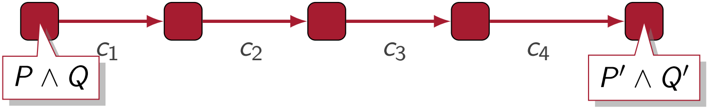
≫**5x** the problem size!

$$\cfrac{\cfrac{\cfrac{\cfrac{P''' \vdash R}{\{P'''\}c_4\{R\} \quad \ldots}}{\{P''\}c_3; c_4\{R\} \quad \ldots}}{\{P'\}c_2; c_3; c_4\{R\} \quad \ldots}}{\{P\}c_1; c_2; c_3; c_4\{R\}}$$



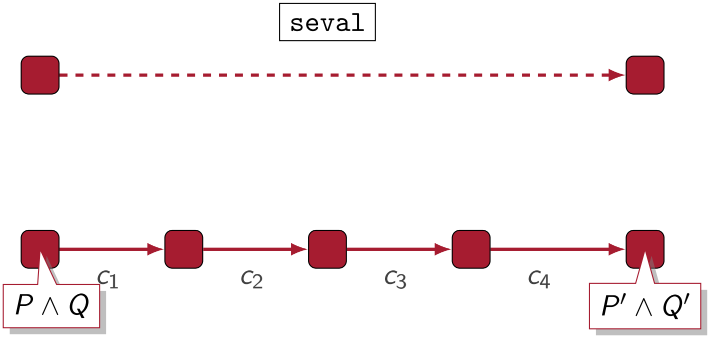
Computational Reflection [Bou97]

- Write a function & prove it sound.



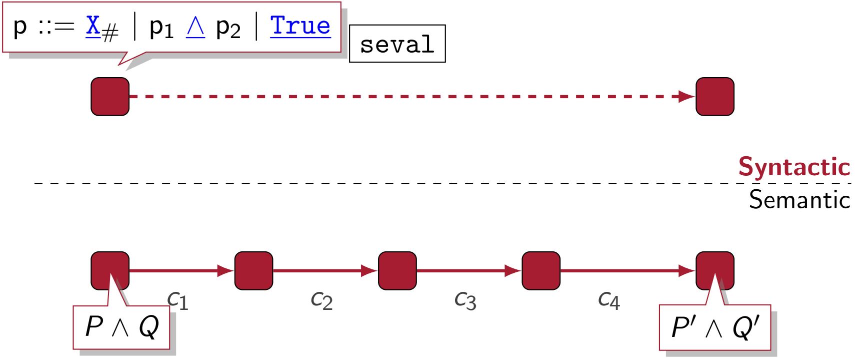
Computational Reflection [Bou97]

- Write a function & prove it sound.



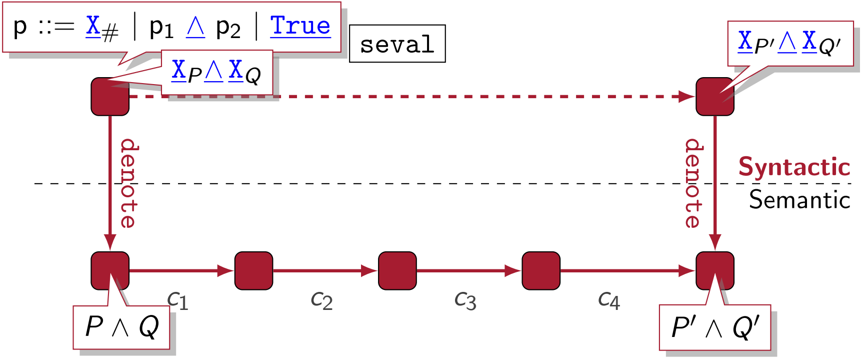
Computational Reflection [Bou97]

- Write a function & prove it sound.



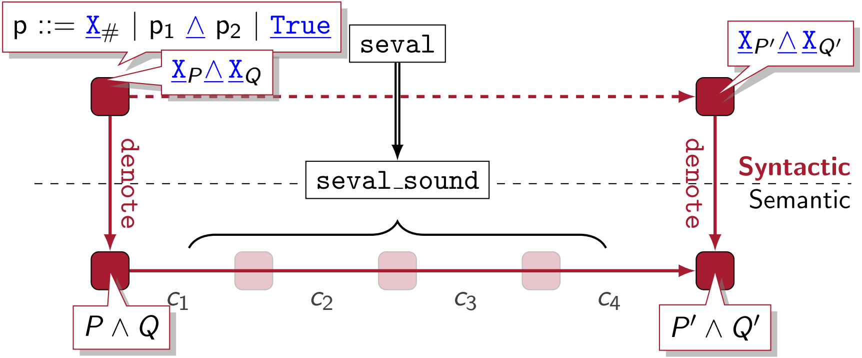
Computational Reflection [Bou97]

- Write a function & prove it sound.



Computational Reflection [Bou97]

- Write a function & prove it sound.

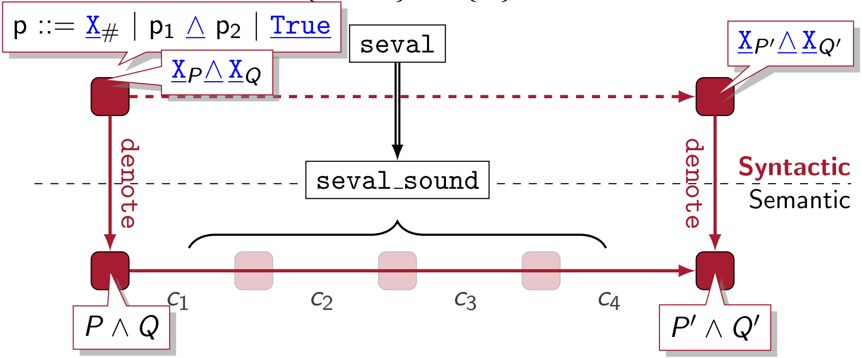


Computational Reflection [Bou97]

- Write a function & prove it sound.

Returned to the user

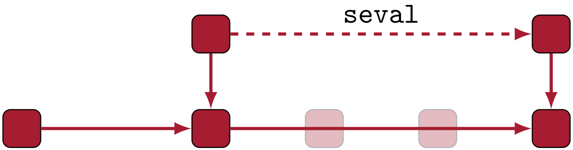
$$\dfrac{\dfrac{\dfrac{\dfrac{}{\underline{X}_{P'}\triangle\underline{X}_{Q'} = \underline{X}_{P'}\triangle\underline{X}_{Q'}}\textsc{Refl}}{\texttt{seval}(\underline{X}_P \triangle \underline{X}_Q)(c_1;..) = \underline{X}_{P'}\triangle\underline{X}_{Q'}}\textsc{Conv} \qquad \dfrac{\dfrac{\vdots}{P' \wedge Q' \vdash R}}{[\![\underline{X}_{P'}\triangle\underline{X}_{Q'}]\!] \vdash R}}{\{[\![\underline{X}_P\triangle\underline{X}_Q]\!]\}c_1;..\{R\}}\textsc{Sound}}{\{P\wedge Q\}c_1;..\{R\}}\textsc{Conv}$$



Reflective Symbolic Execution

```
Ind prop := True | p ∧ q

Def ⟦ p ⟧prop :=
  match p with
  | P ∧ Q ⇒ ⟦P⟧prop ∧ ⟦Q⟧prop
  | ...  ⇒ ...
```

```
Fix seval (p : prop) (c : list cmd) :=
  match c with
    | nil ⇒ p
    | Read x y :: c ⇒
      seval (eval_read p x y) c
    | ...  ⇒ ...
  end

Thm seval_sound : ∀ p c q,
  seval p c = q → {⟦q⟧} c {⟦q⟧}.
Proof. .. Qed.
```

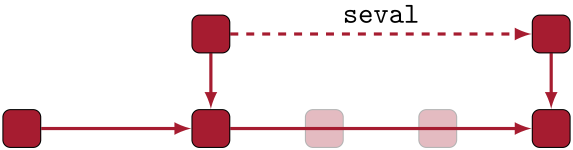


Reflective Symbolic Execution

```
Ind prop := True | p ∧ q | e1 ↦ e2

Def [[ p ]]prop :=
  match p with
  | P ∧ Q ⇒ [[P]]pr
  | ...   ⇒ ...
```

Side conditions?

```
Fix seval (p : prop) (c : list cmd) :=
  match c with
    | nil ⇒ p
    | Read x y :: c ⇒
      seval (eval_read p x y) c
    | ... ⇒ ...
  end

Thm seval_sound : ∀ p c q,
  seval p c = q → {[[q]]} c {[[q]]}.
Proof. .. Qed.
```

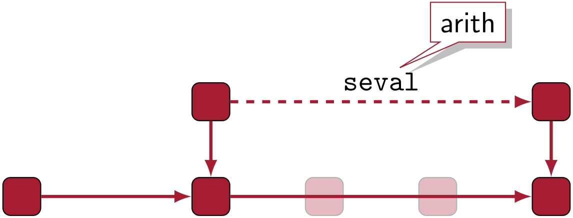


Reflective Symbolic Execution

```
Ind arith := …  | e1 + e2 | e1 − e2

Ind prop := True | p ∧ q | e1↦e2

Def ⟦ p ⟧prop :=
  match p with
  | P ∧ Q ⇒ ⟦P⟧prop ∧ …
  | …  ⇒ …
```

```
Fix seval (p : prop) (c : list cmd) :=
  match c with
    | nil ⇒ p
    | Read x y :: c ⇒
      seval (eval_read p x y) c
    | …  ⇒ …
  end

Thm seval_sound : ∀ p c q,
  seval p c = q → {⟦q⟧} c {⟦q⟧}.
Proof. .. Qed.
```

Side conditions?

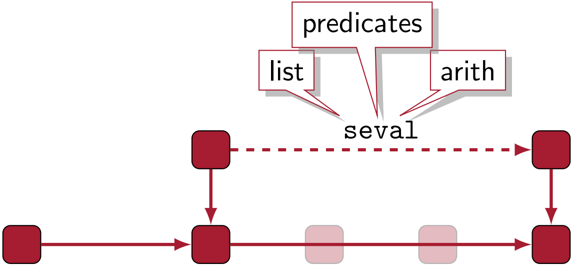


Reflective Symbolic Execution

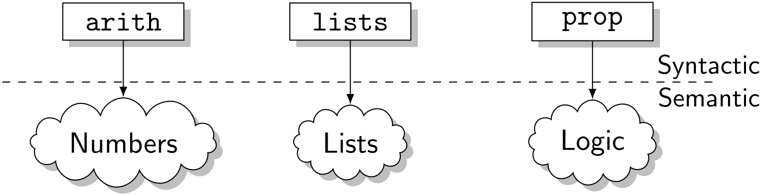
```
Ind arith := ...  | e1 + e2 | e1 − e2
Ind lists := ...  | e1 :: e2 | nil

Ind prop := True | p △ q  | e1↦e2
       | llist e1 e2

Def ⟦ p ⟧prop :=
  match p with
  | P △ Q ⇒ ⟦P⟧pro
  | ...  ⇒ ...
```

Side conditions?

```
Fix seval (p : prop) (c : list cmd) :=
  match c with
    | nil ⇒ p
    | Read x y ::  c ⇒
      seval (eval_read p x y) c
    | ...  ⇒ ...
  end

Thm seval_sound : ∀ p c q,
  seval p c = q → {⟦q⟧} c {⟦q⟧}.
Proof. ..  Qed.
```



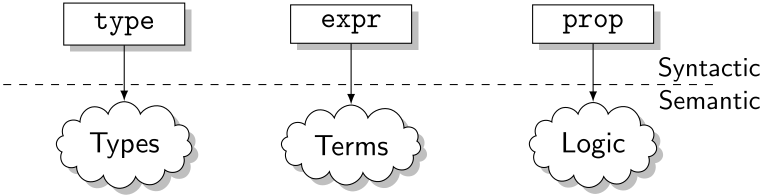
Compositional Syntax



Compositional Syntax

- **Simple** core language

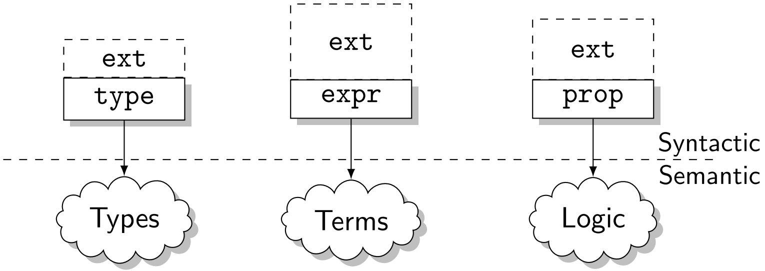
```
Ind typ := Typ (key : K).

Ind expr :=
  Call (key : K) (args : list expr)
| Var (idx : ℕ)
Ind prop := p ∧ q | True | ∃t p
```



Compositional Syntax

- **Simple** core language
- **Extensible** via environments

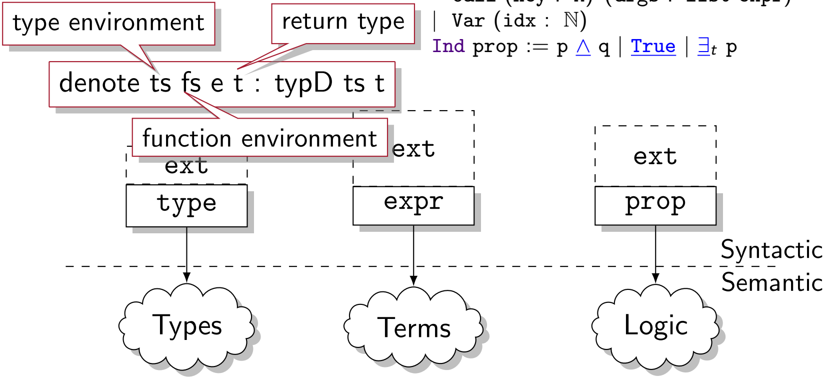
```
Ind typ := Typ (key : K).

Ind expr :=
  Call (key : K) (args : list expr)
| Var (idx : ℕ)
Ind prop := p ∧ q | True | ∃t p
```



Compositional Syntax

- **Simple** core language
- **Extensible** via environments

```
Ind typ := Typ (key : K).

Ind expr :=
  Call (key : K) (args : list expr)
| Var (idx : ℕ)
Ind prop := p ∧ q | True | ∃_t p
```



Reasoning with Environments

Specialized Syntax

```
Def prove_zero e : bool :=
  match e with
  | Plus l r ⇒ ....

Thm prove_zero_sound : ∀ e,
  prove_arith e = true →
  arithD e = 0.
```

Generic Syntax

```
Def prove_zero e : bool :=
  match e with
  | App ? [ l ; r ] ⇒ ....

Thm prove_zero_sound : ∀ ts fs e,


  prove_arith e = true →
  denote ts fs e T? = 0.
```

Reasoning with Environments

Specialized Syntax

```
Def prove_zero e : bool :=
  match e with
  | Plus l r ⇒ ....

Thm prove_zero_sound : ∀ e,
  prove_arith e = true →
  arithD e = 0.
```

τ_1	τ_2	$\tau \ldots$

Generic Syntax

```
Def prove_zero e : bool :=
  match e with
  | App ? [ l ; r ] ⇒ ....

Thm prove_zero_sound : ∀ ts fs e,


  prove_arith e = true →
  denote ts fs e T? = 0.
```

Where is ℕ?

Reasoning with Environments

Specialized Syntax

```
Def prove_zero e : bool :=
  match e with
  | Plus l r ⇒ ....

Thm prove_zero_sound : ∀ e,
  prove_arith e = true →
  arithD e = 0.
```

Generic Syntax

```
Def prove_zero e : bool :=
  match e with
  | App ? [ l ; r ] ⇒ ....

Thm prove_zero_sound : ∀ ts fs e,


  prove_arith e = true →
  denote ts fs e T₁ = 0.
```

Where is $\mathbb{N}$?

τ_1	τ_2	$\tau \ldots$

Arith			
Arith	$\mathbb{Z}$	$\mathbb{N}$	$\mathbb{R}$

Reasoning with Environments

Specialized Syntax

```
Def prove_zero e : bool :=
  match e with
  | Plus l r ⇒ ....

Thm prove_zero_sound : ∀ e,
  prove_arith e = true →
  arithD e = 0.
```

Generic Syntax

```
Def prove_zero e : bool :=
  match e with
  | App ? [ l ; r ] ⇒ ....

Thm prove_zero_sound : ∀ ts fs e,


  prove_arith e = true →
  denote ts fs e T1 = 0.
```

	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}\ldots$
Arith	?	$\mathbb{N}$	?

Reasoning with Environments

Specialized Syntax

```
Def prove_zero e : bool :=
  match e with
  | Plus l r ⇒ ....

Thm prove_zero_sound : ∀ e,
  prove_arith e = true →
  arithD e = 0.
```

Generic Syntax

```
Def prove_zero e : bool :=
  match e with
  | App ? [ l ; r ] ⇒ ....

Thm prove_zero_sound : ∀ ts fs e,
  tc_arith ⊨ ts →

  prove_arith e = true →
  denote ts fs e T1 = 0.
```

	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}\ldots$
Arith	?	$\mathbb{N}$	?

Reasoning with Environments

Specialized Syntax

```
Def prove_zero e : bool :=
  match e with
  | Plus l r ⇒ ....

Thm prove_zero_sound : ∀ e,
  prove_arith e = true →
  arithD e = 0.
```

Generic Syntax

```
Def prove_zero e : bool :=
  match e with
  | App ? [ l ; r ] ⇒ ....

Thm prove_zero_sound : ∀ ts fs e,
  let ts := ts ⊕ tc_arith in

  prove_arith e = true →
  denote ts fs e T1 = 0.
```

$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}\ldots$

$\oplus$

Arith

?	$\mathbb{N}$	?

$\equiv$

$\mathcal{T}_1$	$\mathbb{N}$	$\mathcal{T}\ldots$

Reasoning with Environments

Specialized Syntax

```
Def prove_zero e : bool :=
  match e with
  | Plus l r ⇒ ....

Thm prove_zero_sound : ∀ e,
  prove_arith e = true →
  arithD e = 0.
```

Generic Syntax

```
Def prove_zero e : bool :=
  match e with
  | App 1 [ l ; r ] ⇒ ....

Thm prove_zero_sound : ∀ ts fs e,
  let ts := ts ⊕ tc_arith in
  let fs := fs ⊕ fc_arith in
  prove_arith e = true →
  denote ts fs e T1 = 0.
```

	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}\ldots$
		$\oplus$	
Arith	?	$\mathbb{N}$	?
		$\equiv$	
	$\mathcal{T}_1$	$\mathbb{N}$	$\mathcal{T}\ldots$

F_1	F_2	$F\ldots$
	$\oplus$	
?	$T_1 \to T_1 \to T_1, +$	?
	$\equiv$	
F_1	$T_1 \to T_1 \to T_1, +$	$F\ldots$

Semantic Composition

```
Thm arith_zero_sound : ∀ ts' fs',
let ts := ts' ⊕ tc_arith in
let fs := fs' ⊕ fc_arith in
∀ e,
  arith_zero hs goal = true →
  denote ts fs e T0 = 0.
Proof. ...  Qed.
```

```
Thm list_nil_sound : ∀ ts' fs',
let ts := ts' ⊕ tc_list in
let fs := fs' ⊕ fc_list in
∀ e,
  list_nil e = true →
  denote ts fs e T0 = nil.
Proof. ...  Qed.
```

List	list $\mathbb{N}$	$\mathbb{N}$	?
Arith	?	$\mathbb{N}$	?

Semantic Composition

```
Thm arith_zero_sound : ∀ ts' fs',
let ts := ts' ⊕ tc_arith in
let fs := fs' ⊕ fc_arith in
∀ e,
  arith_zero hs goal = true →
  denote ts fs e T_0 = 0.
Proof. ... Qed.
```

```
Thm list_nil_sound : ∀ ts' fs',
let ts := ts' ⊕ tc_list in
let fs := fs' ⊕ fc_list in
∀ e,
  list_nil e = true →
  denote ts fs e T_0 = nil.
Proof. ... Qed.
```

List	list $\mathbb{N}$	$\mathbb{N}$	?

$\oplus$

Arith	?	$\mathbb{N}$	?

$\equiv$

list $\mathbb{N}$	$\mathbb{N}$	?

Semantic Composition

$(\mathsf{ts} \oplus \mathsf{tc}_{list}) \oplus \mathsf{tc}_{arith}$

```
Thm arith_zero_sound : ∀ ts' fs',
let ts := ts' ⊕ tc_arith in
let fs := fs' ⊕ fc_arith in
∀ e,
  arith_zero hs goal = true →
  denote ts fs e T_0 = 0.
Proof. ...  Qed.
```

$(\mathsf{ts} \oplus \mathsf{tc}_{arith}) \oplus \mathsf{tc}_{list}$

```
Thm list_nil_sound : ∀ ts' fs',
let ts := ts' ⊕ tc_list in
let fs := fs' ⊕ fc_list in
∀ e,
  list_nil e = true →
  denote ts fs e T_0 = nil.
Proof. ...  Qed.
```

Arith	?	$\mathbb{N}$	?
		$\oplus$	
List	list $\mathbb{N}$	$\mathbb{N}$	?
		$\equiv$	
	list $\mathbb{N}$	$\mathbb{N}$	?

Semantic Composition

$(\mathsf{ts} \oplus \mathsf{tc}_{list}) \oplus \mathsf{tc}_{arith}$

```
Thm arith_zero_sound : ∀ ts' fs',
let ts := ts' ⊕ tc_arith in
let fs := fs' ⊕ fc_arith in
∀ e,
  arith_zero hs goal = true →
  denote ts fs e T0 = 0.
Proof. ...  Qed.
```

$(\mathsf{ts} \oplus \mathsf{tc}_{arith}) \oplus \mathsf{tc}_{list}$

```
Thm list_nil_sound : ∀ ts' fs',
let ts := ts' ⊕ tc_list in
let fs := fs' ⊕ fc_list in
∀ e,
  list_nil e = true →
  denote ts fs e T0 = nil.
Proof. ...  Qed.
```

List	list $\mathbb{N}$	$\mathbb{N}$	?
		$\oplus$	
Arith	?	$\mathbb{N}$	?
		$\equiv$	
	list $\mathbb{N}$	$\mathbb{N}$	?

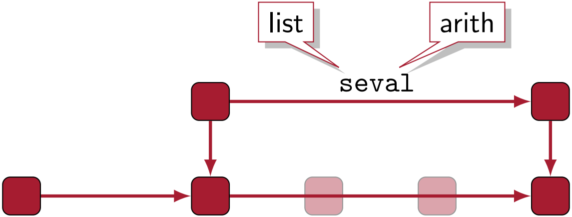
Symmetric composition

Canonical environments

No casts!

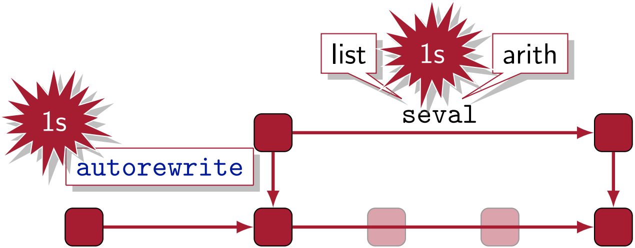
Compositional Symbolic Execution

- Compose provers with compatible constraints
- Parameterize `seval` by provers for side-conditions



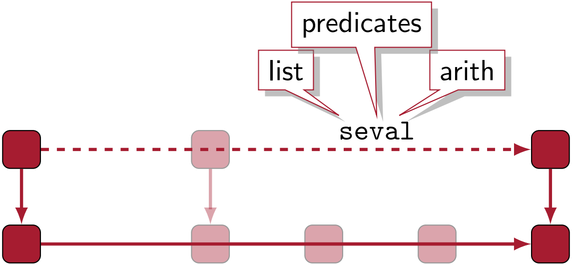
Compositional Symbolic Execution

- Compose provers with compatible constraints
- Parameterize `seval` by provers for side-conditions



Compositional Symbolic Execution

- Compose provers with compatible constraints
- Parameterize `seval` by provers for side-conditions



Generic (Reflective) Extension

- Abstraction enables generic, **reusable** procedures.
  → Avoid boiler-plate automation & proofs!

- `autorewrite` – rewrite with a collection of lemmas

```
Def sll : list W → W → HProp := ...

Thm nil_fwd : ∀ ls (p : W), p = 0
→ sll ls p ⟹ ⌈ ls = nil ⌉.
Proof. .. Qed.

Thm cons_fwd : ∀ ls (p : W), p ≠ 0
→ sll ls p ⟹ ∃ x, ∃ ls', ⌈ ls = x :: ls' ⌉ *
                 ∃ p', p ↦ (x, p') * sll ls' p'.
Proof. .. Qed.

Thm sllMOk : moduleOk sllM.
Proof. vcgen; abstract (sep hints; finish). Qed.
```

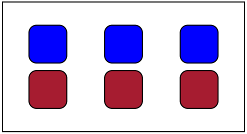
Generic (Reflective) Extension

- Abstraction enables generic, **reusable** procedures.
  → Avoid boiler-plate automation & proofs!

- `autorewrite` – rewrite with a collection of lemmas

```
Def sll : list W → W → HProp := ...

Thm nil_fwd : ∀ ls (p : W), p = 0
→ sll ls p ⟹ ⌈ ls = nil ⌉.
Proof. .. Qed.

Thm cons_fwd : ∀ ls (p : W), p ≠ 0
→ sll ls p ⟹ ∃ x, ∃ ls', ⌈ ls = x :: ls' ⌉ ∗
                ∃ p', p ↦ (x, p') ∗ sll ls' p'.
Proof. .. Qed.

Thm sllMOk : moduleOk sllM.
Proof. vcgen; abstract (sep hints; finish). Qed.
```

Constructed automatically



Hint Database

Generic (Reflective) Extension

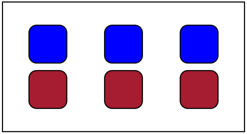
- Abstraction enables generic, **reusable** procedures.
  → Avoid boiler-plate automation & proofs!

- autorewrite – rewrite with a collection of lemmas

```
Def sll : list W → W → HProp := ...

Thm nil_fwd : ∀ ls (p : W), p = 0
→ sll ls p ⟹ ⌈ ls = nil ⌉.
Proof. .. Qed.

Thm cons_fwd : ∀ ls (p : W), p ≠ 0
→ sll ls p ⟹ ∃ x, ∃ ls', ⌈ ls = x :: ls' ⌉ *
                ∃ p', p ↦ (x, p') * sll ls' p'.
Proof. .. Qed.

Thm sllMOk : moduleOk sllM.
Proof. vcgen; abstract (sep hints; finish). Qed.
```

rewrite_all

rewrite_all_sound



Hint Database

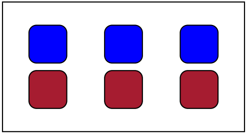
Generic (Reflective) Extension

- Abstraction enables generic, **reusable** procedures.
  → Avoid boiler-plate automation & proofs!
- `autorewrite` – rewrite with a collection of lemmas

```
Def sll : list W → W → HProp := ...

Thm nil_fwd : ∀ ls (p : W), p = 0
→ sll ls p ⟹ ⌈ ls = nil ⌉.
Proof. .. Qed.

Thm cons_fwd : ∀ ls (p : W), p ≠ 0
→ sll ls p ⟹ ∃ x, ∃ ls', ⌈ ls = x :: ls' ⌉ *
              ∃ p', p ↦ (x, p') * sll ls' p'.
Proof. .. Qed.

Thm sllMOk : moduleOk sllM.
Proof. vcgen; abstract (sep hints; finish). Qed.
```

`rewrite_all`

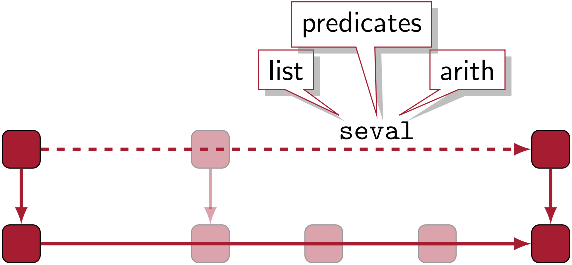
`rewrite_all_sound`



Hint Database

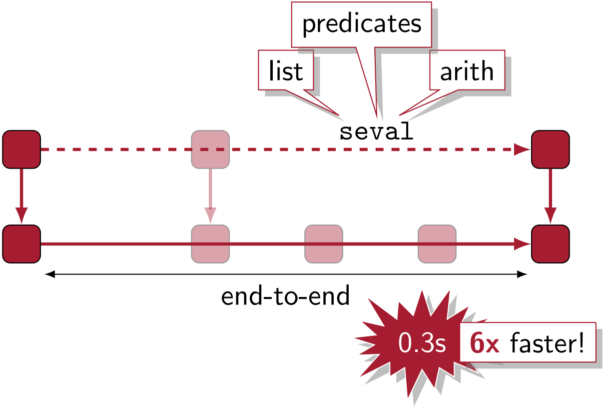
Compositional Symbolic Execution

- Compose provers with compatible constraints
- Parameterize `seval` by provers for side-conditions



Compositional Symbolic Execution

- Compose provers with compatible constraints
- Parameterize `seval` by provers for side-conditions
- Include predicate unfolding hints



Related Work

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

Related Work

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

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

Internalized *judgemental equality*

Recap

```
bfunction "length"("x", "n") [lengthS]
  "n" ← 0;;
  [∀ ls,
   PRE[V] sll ls (V "x")
   POST[R] ⌈ R = V "n" + length ls ⌉ ∗ sll ls (V "x")]
  While ("x" ≠ 0) {
    "n" ← "n" + 1;;
    "x" ← "x" + 4;;
    "x" ←∗ "x"
  };;
  Return "n"
```

```
Def sll : list W → W → HProp := ...

Thm nil_fwd : ∀ ls (p : W), p = 0
→ sll ls p ⟹ ⌈ ls = nil ⌉.
Proof. .. Qed.

Thm cons_fwd : ∀ ls (p : W), p ≠ 0
→ sll ls p ⟹ ∃ x, ∃ ls', ⌈ ls = x :: ls' ⌉ ∗
                ∃ p', p ↦ (x, p') ∗ sll ls' p'.
Proof. .. Qed.
```

seval ⊕ entailment ⊕ rewriting ⊕ lemmas ⊕ provers

```
Thm sllMOk : moduleOk sllM.
Proof. vcgen; abstract (sep hints; finish). Qed.
```

https://github.com/gmalecha/mirror-shard
https://github.com/gmalecha/bedrock-mirror-shard

MirrorCore @ Coq Workshop

References I

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.

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.

References II

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

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.