Reflecting Modular Automation

Gregory Malecha
gmalecha@cs.harvard.edu

Harvard SEAS

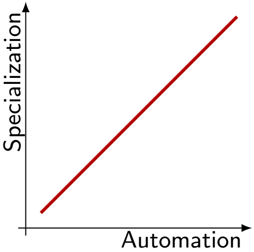
January 18, 2013

The Evolution of Verification

- Verification of single programs used to be a heroic task (Hoare'71)
- Now, the state of the art is about the verifiers/verification methods
  - SMT-based verification (Boogie'06)
  - Dependent types (Coq'04)
  - Program Logics (Shao'07, etc.)
- Goals: easy & expressive verification

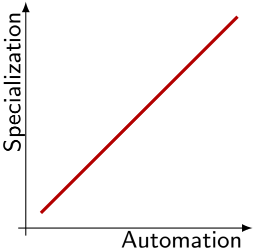
Certainty

- Specializated automation (essential for large verification)
- Already in some proof assistants
  - Sledgehammer – Isabelle



Certainty

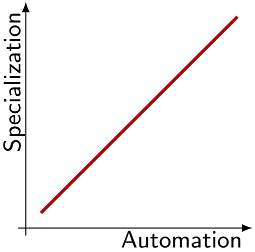
- Specializated automation (essential for large verification)
- Already in some proof assistants
    - Sledgehammer – Isabelle



- Reason in an expressive logic
- Automation for these logics are complex and buggy
- ...checkers can still be simple

Certainty

- Specializated automation (essential for large verification)
- Already in some proof assistants
  - Sledgehammer – Isabelle



- Reason in an expressive logic
- Automation for these logics are complex and buggy
- ...checkers can still be simple
- Want ways to extend simple checkers with custom knowledge
  - ACL2 (Moore)
  - VeriML (Stampolis'12)
  - Computational reflection (Boutin'97)

Focus of this talk

Current Automation

- **Built-in Tactics**: `auto`/`eauto`/`autorewrite`
  - ✓ Fast search
  - ✓ Limited extension points
  - ✗ Can't reason formally about these built-in tactics
  - ✗ Solve or do nothing! Doesn't work well with manual proofs
- **Custom Ltac**: finer granularity control
  - ✓ Easily extensible (higher-order ltac)
  - ✓ Many successes: YNot (Chlipala'09), Bedrock (Chlipala'11), compilers (Chlipala'10)
  - ✗ Limited programming features, dynamic type checking, slow
  - ✗ Can't reason formally about it, "black magic"
- **Reflective Proofs**

Technique for this talk

Reflective Proofs aren't new!

that's >15 years ago!

- but reflective proofs aren't new! (Boutin'97)

Reflective Proofs aren't new!

that's >15 years ago!

- but reflective proofs aren't new! (Boutin'97)

- This talk is about new ways to use them to
  1. Integrate with Coq's unification
  2. Integrate with Ltac
  3. Reason about open domains

Reflective Proofs aren't new!

that's >15 years ago!

- but reflective proofs aren't new! (Boutin'97)
- This talk is about new ways to use them to
  1. Integrate with Coq's unification
  2. Integrate with Ltac
  3. Reason about open domains
- ...and for me to learn tips from all of you!

Outline

1. Reflective Proofs
2. Automation in Bedrock
3. Techniques for Composable Reflective Procedures
   - Term Representation
   - Composition
   - Proof Term Engineering
4. Moving Forward

Outline

1. Reflective Proofs
2. Automation in Bedrock
3. Techniques for Composable Reflective Procedures
   - Term Representation
   - Composition
   - Proof Term Engineering
4. Moving Forward

Simple Proofs

- Formalizing "evenness" with **inductive definitions**

Evenness

$$\frac{}{\text{Even } 0}\ \mathrm{E0}$$

$$\frac{\text{Even } n}{\text{Even } (n+2)}\ \mathrm{E_N}$$

A Proof Even 4

$$\cfrac{\cfrac{\cfrac{}{\text{Even } 0}\ \mathrm{E0}}{\text{Even } 2}\ \mathrm{E_N}}{\text{Even } 4}\ \mathrm{E_N}$$

Limitations of Inductive Proofs

- What if we wanted to prove Even 2048?

Limitations of Inductive Proofs

- What if we wanted to prove Even 2048? **YIKES**

A Proof Even 2048

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{Even } 0}\,\mathrm{E0}}{\text{Even } 2}\,\mathrm{EN}}{\vdots}\,\mathrm{EN}}{\text{Even } 2042}\,\mathrm{EN}}{\text{Even } 2044}\,\mathrm{EN}}{\text{Even } 2046}\,\mathrm{EN}}{\text{Even } 2048}\,\mathrm{EN}$$

Limitations of Inductive Proofs

13s to build/check

A Proof Even 2048

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\text{Even } 0}\ \mathrm{E0}}{\text{Even } 2}\ \mathrm{EN}}{\vdots}\ \mathrm{EN}}{\text{Even } 2042}\ \mathrm{EN}}{\text{Even } 2044}\ \mathrm{EN}}{\text{Even } 2046}\ \mathrm{EN}}{\text{Even } 2048}\ \mathrm{EN}$$

- What if we wanted to prove Even 2048? **YIKES**
- Easy proof! What's wrong?
  - ✗ Too big
  - ✗ Too long to generate
  - ✗ Too long to check

Limitations of Inductive Proofs

13s to build/check

A Proof Even 2048

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{Even } 0}\,\text{E0}}{\text{Even } 2}\,\text{EN}}{\vdots}\,\text{EN}}{\text{Even } 2042}\,\text{EN}}{\text{Even } 2044}\,\text{EN}}{\text{Even } 2046}\,\text{EN}}{\text{Even } 2048}\,\text{EN}$$

- What if we wanted to prove Even 2048? **YIKES**
- Easy proof! What's wrong?
  - ✗ Too big
  - ✗ Too long to generate
  - ✗ Too long to check

Problem: Lack of abstraction...

- Reduce everything to the base inductive types.

An Alternative Proof Style: Reflective Proofs

How to solve the problem?

- Extend the core logic with an understanding of Evenness?

An Alternative Proof Style: Reflective Proofs

How to solve the problem?

- Extend the core logic with an understanding of Evenness?

Checking Evenness

```
let is_even n =
  if n = 0 then true
  else if n = 1 then false
  else is_even (n - 2)
```

An Alternative Proof Style: Reflective Proofs

How to solve the problem?

- Extend the core logic with an understanding of Evenness?

Checking Evenness

```
let is_even n =
  if n = 0 then true
  else if n = 1 then false
  else is_even (n - 2)
```

Write a function to check evenness

An Alternative Proof Style: Reflective Proofs

How to solve the problem?

- Extend the core logic with an understanding of Evenness?

Not foundational :'(

- Unless we **prove** that the extension is sound.

Checking Evenness

```
let is_even n =
  if n = 0 then true
  else if n = 1 then false
  else is_even (n − 2)
```

Justifying the Checker

```
Theorem is_even_sound : ∀ n,
  is_even n = true → Even n.
Proof. ... Qed.
```

Prove the procedure is sound

Reflective Proofs

Inductive Proof

$$\cfrac{\cfrac{\vdots}{\cfrac{\cfrac{\cfrac{\text{Even } 2042}{\text{Even } 2044}\,\textsc{En}}{\text{Even } 2046}\,\textsc{En}}{\text{Even } 2048}\,\textsc{En}}}{}$$

$$\cfrac{\cfrac{\cfrac{}{\text{Even } 0}\,\text{E0}}{\text{Even } 2}\,\textsc{En}}{\vdots}\,\textsc{En} \qquad \cfrac{\vdots}{\text{Even } 2042}\,\textsc{En}$$

Reflective Proof

$$\cfrac{}{\text{Even } 2048}$$

Reflective Proofs

Inductive Proof

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\text{Even } 0}\ \textsc{E0}}{\text{Even } 2}\ \textsc{En}}{\vdots}\ \textsc{En}}{\text{Even } 2042}\ \textsc{En}}{\text{Even } 2044}\ \textsc{En}}{\text{Even } 2046}\ \textsc{En}}{\text{Even } 2048}\ \textsc{En}$$

Reflective Proof

$$\frac{\text{is\_even } 2048 = \text{true}}{\text{Even } 2048}\ \text{is\_even\_sound}$$

Apply the lemma

Reflective Proofs

Inductive Proof

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\text{Even } 0}\,\text{E0}}{\text{Even } 2}\,\text{EN}}{\vdots}\,\text{EN}}{\text{Even } 2042}\,\text{EN}}{\text{Even } 2044}\,\text{EN}}{\text{Even } 2046}\,\text{EN}}{\text{Even } 2048}\,\text{EN}}{}
$$

Reflective Proof

Reduction

$$
\cfrac{\cfrac{\text{true} = \text{true}}{\text{is\_even } 2048 = \text{true}}\,\beta\text{-}\textsc{Conv}}{\text{Even } 2048}\,\text{is\_even\_sound}
$$

Reflective Proofs

Inductive Proof

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\text{Even } 0}\ \textsc{E0}}{\text{Even } 2}\ \textsc{En}}{\vdots}\ \textsc{En}}{\text{Even } 2042}\ \textsc{En}}{\text{Even } 2044}\ \textsc{En}}{\text{Even } 2046}\ \textsc{En}}{\text{Even } 2048}\ \textsc{En}
$$

Reflective Proof

Reflexivity

$$
\cfrac{\cfrac{\cfrac{}{\text{true} = \text{true}}\ \textsc{Refl}}{\text{is\_even } 2048 = \text{true}}\ \beta\text{-}\textsc{Conv}}{\text{Even } 2048}\ \text{is\_even\_sound}
$$

Reflective Proofs

13s to build/check

<1s to build/check

Inductive Proof

$$\dfrac{\dfrac{\vdots}{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{Even } 0}\,\textsc{E0}}{\text{Even } 2}\,\textsc{En}}{}\,\textsc{En}}{}}}{}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\text{Even } 2042}\,\textsc{En}}{\text{Even } 2044}\,\textsc{En}}{\text{Even } 2046}\,\textsc{En}}{\text{Even } 2048}\,\textsc{En}$$

Reflective Proof

$$\dfrac{\dfrac{\dfrac{}{\text{true} = \text{true}}\,\textsc{Refl}}{\text{is\_even } 2048 = \text{true}}\,\beta\text{-}\textsc{Conv}}{\text{Even } 2048}\,\text{is\_even\_sound}$$

1025 rules vs. 3 rules

Outline

1. Reflective Proofs
2. Automation in Bedrock
3. Techniques for Composable Reflective Procedures
   - Term Representation
   - Composition
   - Proof Term Engineering
4. Moving Forward

Automation for Low-Level Code

Bedrock
is a framework for writing and reasoning about low-level imperative code

- This domain requires a great deal of extensibility
    - Abstractions are defined in the logic, not in the language
        - Abstract predicates for new data structures
    - Domain reasoning for bitvectors, equality, and arithmatic

```
Definition hints : TacPackage.
  prepare (bst_fwd, nil_fwd, cons_fwd) (bst_bwd, nil_bwd, cons_bwd).
Defined.

Definition bstM := bmodule "bst" {{
  bfunction "lookup"("s", "k", "tmp") [lookupS]
    "s" ← * "s";;
    [Ex s, Ex t,
      PRE[V] bst' s t (V "s") * mallocHeap
      POST[R] ⌈ (V "k" ∈ s) \is R ⌉ * bst' s t (V "s") * mallocHeap]
    While ("s" ≠ 0) {
      "tmp" ← "s" + 4;;
      "tmp" ← * "tmp";;
      If ("k" = "tmp") {
        (* Key matches! *)
        Return 1
      } else {
        If ("k" < "tmp") {
          (* Searching for a lower key *)
          "s" ← * "s"
        } else {
          (* Searching for a higher key *)
          "s" ← "s" + 8;;
          "s" ← * "s"
        }
      }
    };;
    Return 0
  end }}.

Theorem bstMOk : moduleOk bstM.
Proof. vcgen; abstract (sep hints; auto). Qed.
```

Imperative code

Hoare logic-like specifications

Separation logic

```
Definition hints : TacPackage.
  prepare (bst_fwd, nil_fwd, cons_fwd) (bst_bwd, nil_bwd, cons_bwd).
Defined.

Definition bstM := bmodule "bst" {{
  bfunction "lookup"("s", "k", "tmp") [l
    "s" ←∗ "s";;
    [Ex s, Ex t,
     PRE[V] bst' s t (V "s") ∗ mallocHeap
     POST[R] ⌈ (V "k" ∈ s) \is R ⌉ ∗ bst' s t (V "s") ∗ mallocHeap]
    While ("s" ≠ 0) {
      "tmp" ← "s" + 4;;
      "tmp" ←∗ "tmp";;
      If ("k" = "tmp") {
        (* Key matches! *)
        Return 1
      } else {
        If ("k" < "tmp") {
          (* Searching for a lower key *)
          "s" ←∗ "s"
        } else {
          (* Searching for a higher key *)
          "s" ← "s" + 8;;
          "s" ←∗ "s"
        }
      }
    };;
    Return 0
  end }}.

Theorem bstMOk : moduleOk bstM.
Proof. vcgen; abstract (sep hints; auto). Qed.
```

```
Definition hints : TacPackage.
  prepare (bst_fwd, nil_fwd, cons_fwd) (bst_bwd, nil_bwd, cons_bwd).
Defined.

Definition bstM := bmodule "bst" {{
  bfunction "lookup"("s", "k", "tmp") [lookupS]
    "s" ← ∗ "s";;
    [Ex s, Ex t,
      PRE[V] bst' s t (V "s") ∗ mallocHeap
      POST[R] ⌈ (V "k" ∈ s) \is R ⌉ ∗ bst' s t (V "s") ∗ mallocHeap]
    While ("s" ≠ 0) {
      "tmp" ← "s" + 4;;
      "tmp" ← ∗ "tmp";;
      If ("k" = "tmp") {
        (* Key matches! *)
        Return 1
      } else {
        If ("k" < "tmp") {
          (* Searching for a lower key *)
          "s" ← ∗ "s"
        } else {
          (* Searching for a higher key *)
          "s" ← "s" + 8;;
          "s" ← ∗ "s"
        }
      }
  end }

Theorem bstMOk : moduleOk bstM.
Proof. vcgen; abstract (sep hints; auto). Qed.
```

Verification

```
Definition hints : TacPackage.
  prepare (bst_fwd, nil_fwd, cons_fwd) (bst_bwd, nil_bwd, cons_bwd).
Defined.

Definition bstM :=
  bfunction "lookup"("s", "k", "tmp") [lookupS]
    "s" ←∗ "s";;
    [Ex s, Ex t,
      PRE[V] bst' s t (V "s") ∗ mallocHeap
      POST[R] ⌈ (V "k" ∈ s) \is R ⌉ ∗ bst' s t (V "s") ∗ mallocHeap]
    While ("s" ≠ 0) {
      "tmp" ← "s" + 4;;
      "tmp" ←∗ "tmp";;
      If ("k" = "tmp") {
        (∗ Key matches! ∗)
        Return 1
      } else {
        If ("k" < "tmp") {
          (∗ Searching for a lower key ∗)
          "s" ←∗ "s"
        } else {
          (∗ Searching for a higher key ∗)
          "s" ← "s" + 8;;
          "s" ←∗ "s"
        }
      }
    };;
    Return 0
  end }}.

Theorem bstMOk : moduleOk bstM.
Proof. vcgen; abstract (sep hints; auto). Qed.
```

User-extensible hints

Automation

High-level Bedrock Automation

Verification-condition generation

$$\{p \mapsto 1 * q \mapsto 2\} \quad \mathsf{t = {*}p; {*}p = {*}q; {*}q = t} \quad \{q \mapsto 1 * p \mapsto 2\}$$

High-level BEDROCK Automation

Verification-condition generation

Symbolic Execution

$\{p \mapsto 1 * q \mapsto 2\}$ t = *p;*p = *q;*q = t $\{q \mapsto 1 * p \mapsto 2\}$

High-level BEDROCK Automation

Verification-condition generation

Symbolic Execution

$$\{p \mapsto 1 * q \mapsto 2\} \quad \mathsf{t = *p;*p = *q;*q = t} \quad \{q \mapsto 1 * p \mapsto 2\}$$

$$\mathbf{t = 1} \vdash \quad \{p \mapsto 1 * q \mapsto 2\} \quad \mathsf{*p = *q;*q = t} \quad \{q \mapsto 1 * p \mapsto 2\}$$

High-level BEDROCK Automation

Verification-condition generation

Symbolic Execution

$$\{p \mapsto 1 * q \mapsto 2\} \quad \mathsf{t = *p;*p = *q;*q = t} \quad \{q \mapsto 1 * p \mapsto 2\}$$

$$\mathbf{t = 1} \vdash \{p \mapsto 1 * q \mapsto 2\} \quad \mathsf{*p = *q;*q = t} \quad \{q \mapsto 1 * p \mapsto 2\}$$

$$\mathsf{t} = 1 \vdash \{p \mapsto \mathbf{2} * q \mapsto 2\} \quad \mathsf{*q = t} \quad \{q \mapsto 1 * p \mapsto 2\}$$

High-level BEDROCK Automation

Verification-condition generation

Symbolic Execution

	$\{p \mapsto 1 * q \mapsto 2\}$	t = *p;*p = *q;*q = t	$\{q \mapsto 1 * p \mapsto 2\}$
$\mathbf{t = 1} \vdash$	$\{p \mapsto 1 * q \mapsto 2\}$	*p = *q;*q = t	$\{q \mapsto 1 * p \mapsto 2\}$
$t = 1 \vdash$	$\{p \mapsto \mathbf{2} * q \mapsto 2\}$	*q = t	$\{q \mapsto 1 * p \mapsto 2\}$
$t = 1 \vdash$	$\{p \mapsto 2 * q \mapsto \mathbf{t}\}$	skip	$\{q \mapsto 1 * p \mapsto 2\}$

High-level Bedrock Automation

Verification-condition generation

Symbolic Execution

$$\{p \mapsto 1 * q \mapsto 2\} \quad \texttt{t = *p;*p = *q;*q = t} \quad \{q \mapsto 1 * p \mapsto 2\}$$

$$\mathbf{t = 1} \vdash \{p \mapsto 1 * q \mapsto 2\} \quad \texttt{*p = *q;*q = t} \quad \{q \mapsto 1 * p \mapsto 2\}$$

$$t = 1 \vdash \{p \mapsto \mathbf{2} * q \mapsto 2\} \quad \texttt{*q = t} \quad \{q \mapsto 1 * p \mapsto 2\}$$

$$t = 1 \vdash \{p \mapsto 2 * q \mapsto \mathbf{t}\} \quad \texttt{skip} \quad \{q \mapsto 1 * p \mapsto 2\}$$

Higher-order reasoning

High-level BEDROCK Automation

Verification-condition generation

Symbolic Execution

	$\{p \mapsto 1 * q \mapsto 2\}$	t = *p;*p = *q;*q = t	$\{q \mapsto 1 * p \mapsto 2\}$
$\mathbf{t = 1} \vdash$	$\{p \mapsto 1 * q \mapsto 2\}$	*p = *q;*q = t	$\{q \mapsto 1 * p \mapsto 2\}$
$t = 1 \vdash$	$\{p \mapsto \mathbf{2} * q \mapsto 2\}$	*q = t	$\{q \mapsto 1 * p \mapsto 2\}$
$t = 1 \vdash$	$\{p \mapsto 2 * q \mapsto \mathbf{t}\}$	skip	$\{q \mapsto 1 * p \mapsto 2\}$

Higher-order reasoning

Entailment Checking

$$t = 1 \vdash \quad p \mapsto 2 * q \mapsto t \quad \Longrightarrow \quad q \mapsto 1 * p \mapsto 2$$

High-level Bedrock Automation

Verification-condition generation

Symbolic Execution

$$
\begin{array}{rlrl}
 & \{p \mapsto 1 * q \mapsto 2\} & \mathsf{t} = {}^*\mathsf{p};{}^*\mathsf{p} = {}^*\mathsf{q};{}^*\mathsf{q} = \mathsf{t} & \{q \mapsto 1 * p \mapsto 2\} \\
\mathbf{t = 1} \vdash & \{p \mapsto 1 * q \mapsto 2\} & {}^*\mathsf{p} = {}^*\mathsf{q};{}^*\mathsf{q} = \mathsf{t} & \{q \mapsto 1 * p \mapsto 2\} \\
\mathsf{t} = 1 \vdash & \{p \mapsto \mathbf{2} * q \mapsto 2\} & {}^*\mathsf{q} = \mathsf{t} & \{q \mapsto 1 * p \mapsto 2\} \\
\mathsf{t} = 1 \vdash & \{p \mapsto 2 * q \mapsto \mathbf{t}\} & \mathsf{skip} & \{q \mapsto 1 * p \mapsto 2\}
\end{array}
$$

Higher-order reasoning

Entailment Checking

$$
\begin{array}{rrcl}
\mathsf{t} = 1 \vdash & p \mapsto 2 * q \mapsto t & \Longrightarrow & q \mapsto 1 * p \mapsto 2 \\
\mathsf{t} = 1 \vdash & q \mapsto t & \Longrightarrow & q \mapsto 1
\end{array}
$$

Using domain provers

High-level BEDROCK Automation

Verification-condition generation

Symbolic Execution

$$
\begin{array}{rlrl}
 & \{p \mapsto 1 * q \mapsto 2\} & \mathsf{t = *p; *p = *q; *q = t} & \{q \mapsto 1 * p \mapsto 2\} \\
\mathbf{t = 1} \vdash & \{p \mapsto 1 * q \mapsto 2\} & \mathsf{*p = *q; *q = t} & \{q \mapsto 1 * p \mapsto 2\} \\
\mathsf{t} = 1 \vdash & \{p \mapsto \mathbf{2} * q \mapsto 2\} & \mathsf{*q = t} & \{q \mapsto 1 * p \mapsto 2\} \\
\mathsf{t} = 1 \vdash & \{p \mapsto 2 * q \mapsto \mathbf{t}\} & \mathsf{skip} & \{q \mapsto 1 * p \mapsto 2\}
\end{array}
$$

Higher-order reasoning

Entailment Checking

$$
\begin{array}{rrcl}
\mathsf{t} = 1 \vdash & p \mapsto 2 * q \mapsto t & \Longrightarrow & q \mapsto 1 * p \mapsto 2 \\
\mathsf{t} = 1 \vdash & q \mapsto t & \Longrightarrow & q \mapsto 1 \\
\mathsf{t} = 1 \vdash & \emptyset & \Longrightarrow & \emptyset
\end{array}
$$

High-level BEDROCK Automation

Verification-condition generation

Symbolic Execution

	$\{p \mapsto 1 * q \mapsto 2\}$	$\mathsf{t = *p;*p = *q;*q = t}$	$\{q \mapsto 1 * p \mapsto 2\}$
$\mathbf{t = 1} \vdash$	$\{p \mapsto 1 * q \mapsto 2\}$	$\mathsf{*p = *q;*q = t}$	$\{q \mapsto 1 * p \mapsto 2\}$
$\mathsf{t} = 1 \vdash$	$\{p \mapsto \mathbf{2} * q \mapsto 2\}$	$\mathsf{*q = t}$	$\{q \mapsto 1 * p \mapsto 2\}$
$\mathsf{t} = 1 \vdash$	$\{p \mapsto 2 * q \mapsto \mathbf{t}\}$	skip	$\{q \mapsto 1 * p \mapsto 2\}$

Higher-order reasoning

Entailment Checking

$\mathsf{t} = 1 \vdash$	$p \mapsto 2 * q \mapsto t$	$\Longrightarrow$	$q \mapsto 1 * p \mapsto 2$
$\mathsf{t} = 1 \vdash$	$q \mapsto t$	$\Longrightarrow$	$q \mapsto 1$
$\mathsf{t} = 1 \vdash$	$\emptyset$	$\Longrightarrow$	$\emptyset$

Pure conclusions

High-level BEDROCK Automation

Verification-condition generation

Symbolic Execution

Reflective

	$\{p \mapsto 1 * q \mapsto 2\}$	t = *p;*p = *q;*q = t	$\{q \mapsto 1 * p \mapsto 2\}$
t = 1 $\vdash$	$\{p \mapsto 1 * q \mapsto 2\}$	*p = *q;*q = t	$\{q \mapsto 1 * p \mapsto 2\}$
t = 1 $\vdash$	$\{p \mapsto \mathbf{2} * q \mapsto 2\}$	*q = t	$\{q \mapsto 1 * p \mapsto 2\}$
t = 1 $\vdash$	$\{p \mapsto 2 * q \mapsto \mathbf{t}\}$	skip	$\{q \mapsto 1 * p \mapsto 2\}$

Higher-order reasoning

Entailment Checking

Reflective

t = 1 $\vdash$	$p \mapsto 2 * q \mapsto t$	$\Longrightarrow$	$q \mapsto 1 * p \mapsto 2$
t = 1 $\vdash$	$q \mapsto t$	$\Longrightarrow$	$q \mapsto 1$
t = 1 $\vdash$	$\emptyset$	$\Longrightarrow$	$\emptyset$

Pure conclusions

The Need for Extensibility

Consecutive cells

Access the second

$\{p \mapsto (x, y)\}$ a = *(p+4);... $\{a = y * \dots\}$

The Need for Extensibility

- All of these programs are essentially the same...

$$
\begin{array}{rlll}
 & \{p \mapsto (x,y)\} & \texttt{a = *(p+4);}\ldots & \{a = y * \ldots\} \\
 & \{p \mapsto (x,y)\} & \texttt{a = *(4+p);}\ldots & \{a = y * \ldots\} \\
q = p + 4 \vdash & \{p \mapsto (x,y)\} & \texttt{a = *q;}\ldots & \{a = y * \ldots\} \\
q = p + 1 * 4 \vdash & \{p \mapsto (x,y)\} & \texttt{a = *q;}\ldots & \{a = y * \ldots\}
\end{array}
$$

Symbolic values

The Need for Extensibility

- All of these programs are essentially the same...

$$
\begin{array}{rlll}
 & \{p \mapsto (x,y)\} & \texttt{a = *(p+4);...} & \{a = y * \ldots\} \\
 & \{p \mapsto (x,y)\} & \texttt{a = *(4+p);...} & \{a = y * \ldots\} \\
q = p + 4 \vdash & \{p \mapsto (x,y)\} & \texttt{a = *q;...} & \{a = y * \ldots\} \\
q = p + 1 * 4 \vdash & \{p \mapsto (x,y)\} & \texttt{a = *q;...} & \{a = y * \ldots\}
\end{array}
$$

- We need to reason *symbolically* about variables

$$
\begin{array}{cccc}
 & & q = p + 4 & q = 4 + p \\
\vdash p{+}4 = p{+}4 & \vdash p{+}4 = 4{+}p & \vdash p + 4 = q & \vdash p + 4 = q
\end{array}
$$

Must be able to reason about **arbitrary** domains.

Example: lists, words, ...

Extensible Automation of Side-conditions

- Implement reflective provers that can prove these side-conditions & verify them.
- **Example** Check if the fact is know from a hypothesis.

Extensible Automation of Side-conditions

- Implement reflective provers that can prove these side-conditions & verify them.
- **Example** Check if the fact is know from a hypothesis.

Check Hypotheses

```
let provable (facts: props)
   (p : prop) =
  contains? facts p
```

Look for p in facts

Check Reflexivit

```
let provable (facts: props)
   (p : prop) =
  match p with
    | Equal l r ⇒ equal? l r
    | _ ⇒ false
  end
```

Syntactic equality

Extensible Automation of Side-conditions

- Implement reflective provers that can prove these side-conditions & verify them.
- **Example** Check if the fact is know from a hypothesis.
- Other provers
  - Machine words
  - Lists

Check Hypotheses

```
let provable (facts: props)
   (p : prop) =
  contains? facts p
```

Check Reflexivity

```
let provable (facts: props)
   (p : prop) =
  match p with
   | Equal l r ⇒ equal? l r
   | _ ⇒ false
  end
```

Extensible Automation of Side-conditions

- Implement reflective provers that can prove these side-conditions & verify them.
- **Example** Check if the fact is know from a hypothesis.
- Other provers
  - Machine words
  - Lists
- **Composable Proofs!**
  - Sound provers can be used by sound procedures

Check Hypotheses

```
let provable (facts: props)
   (p : prop) =
  contains? facts p
```

Check Reflexivity

```
let provable (facts: props)
   (p : prop) =
  match p with
    | Equal l r ⇒ equal? l r
    | _ ⇒ false
  end
```

Extensible Automation of Side-conditions

- Implement reflective provers that can prove these side-conditions & verify them.
- **Example** Check if the fact is know from a hypothesis.
- Other provers
  - Machine words
  - Lists
- **Composable Proofs!**
  - Sound provers can be used by sound procedures
  - Provers about different domains compose

Check Hypotheses

```
let provable ts (facts: props ts)
   (p : prop ts) =
  contains? facts p
```

Check Reflexivity

```
let provable ts (facts: props ts)
   (p : prop ts) =
  match p with
    | Equal l r ⇒ equal? l r
    | _ ⇒ false
  end
```

Extensible Abstract Predicate Refinement

- Bedrock enables user-defined data abstractions to be automated!

1. Heap implications are given as **hints** to the verifier.

List Unfold

$\texttt{Theorem list\_fwd} : \forall\ \texttt{p xs},$
$\quad \texttt{p} \neq 0 \rightarrow$
$\quad \texttt{llist p xs} \Longrightarrow$
$\quad \exists\ \texttt{x xs' q},\ \texttt{xs} = \texttt{x} :: \texttt{xs'} \ast$
$\qquad \texttt{p} \mapsto (\texttt{x},\texttt{q}) \ast \texttt{llist q xs'}$

Extensible Abstract Predicate Refinement

- Bedrock enables user-defined data abstractions to be automated!

1. Heap implications are given as **hints** to the verifier.
   - "Does this hint apply?" **unification** problem!

List Unfold

$\texttt{Theorem list\_fwd} : \forall\ \texttt{p xs},$
$\quad \texttt{p} \neq 0 \rightarrow$
$\quad \texttt{llist p xs} \Longrightarrow$
$\quad \exists\ \texttt{x xs' q},\ \texttt{xs} = \texttt{x :: xs'} *$
$\qquad \texttt{p} \mapsto (\texttt{x,q}) * \texttt{llist q xs'}$

Extensible Abstract Predicate Refinement

- Bedrock enables user-defined data abstractions to be automated!

1. Heap implications are given as **hints** to the verifier.
   - "Does this hint apply?" **unification** problem!
2. Side-conditions discharged by provers.

List Unfold

```
Theorem list_fwd : ∀ p xs,
  p ≠ 0 →
  llist p xs ⟹
  ∃ x xs' q, xs = x :: xs' *
     p ↦ (x,q) * llist q xs'
```

Extensible Abstract Predicate Refinement

- Bedrock enables user-defined data abstractions to be automated!

1. Heap implications are given as **hints** to the verifier.
   - "Does this hint apply?" **unification** problem!
2. Side-conditions discharged by provers.
3. Potentially introduces **new** variables!

List Unfold

```
Theorem list_fwd : ∀ p xs,
  p ≠ 0 →
  llist p xs ⟹
  ∃ x xs' q, xs = x :: xs' *
     p ↦ (x,q) * llist q xs'
```

Extensible Abstract Predicate Refinement

- Bedrock enables user-defined data abstractions to be automated!

1. Heap implications are given as **hints** to the verifier.
   - "Does this hint apply?" **unification** problem!
2. Side-conditions discharged by provers.
3. Potentially introduces **new** variables!

List Unfold

```
Theorem list_fwd : ∀ p xs,
  p ≠ 0 →
  llist p xs ⟹
  ∃ x xs' q, xs = x :: xs' *
     p ↦ (x,q) * llist q xs'
```

All predicates are interpreted like this

Nothing "baked-in"

Applications

- Bedrock code
  - Data types
  - Thread scheduler
  - Garbage collector

Applications

- Bedrock code
  - Data types
  - Thread scheduler
  - Garbage collector
- Compilation
  - Low-level program is a function of an input program
  - Standard tactics (and Ltac) do higher-order reasoning, automation takes care of symbolic execution, entailment checking, etc.

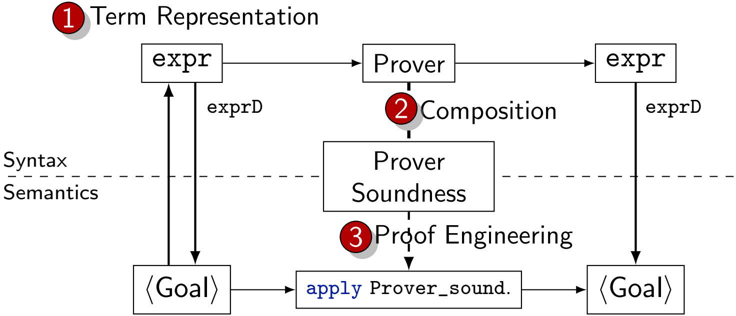
Applications

- Bedrock code
  - Data types
  - Thread scheduler
  - Garbage collector
- Compilation
  - Low-level program is a function of an input program
  - Standard tactics (and Ltac) do higher-order reasoning, automation takes care of symbolic execution, entailment checking, etc.
- Other verification systems
  - MIT-Yale-Princeton team – Cyber-physical systems (HACMS)
  - Princeton will be using our entailment checker for reasoning about C

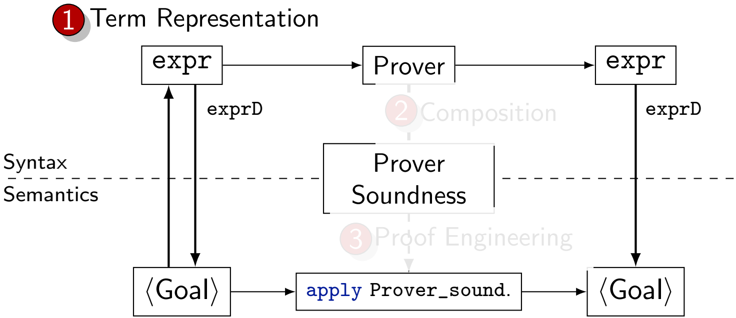
Outline

1. Reflective Proofs
2. Automation in Bedrock
3. Techniques for Composable Reflective Procedures
   - Term Representation
   - Composition
   - Proof Term Engineering
4. Moving Forward

Techniques for Composing Automation



Techniques for Composing Automation



Representing Terms

```
Variable ts : list Type.
Inductive expr : Type :=
| Const : ∀ t, tvarD ts t → expr
| Var : nat → expr
| UVar : nat → expr
| Func : nat → list expr → expr
| Equal : tvar → expr → expr → expr.
```

Representing Terms

Type environment

```
Variable ts : list Type
Inductive expr : Type :=
| Const : ∀ t, tvarD ts t → expr
| Var : nat → expr
| UVar : nat → expr
| Func : nat → list expr → expr
| Equal : tvar → expr → expr → expr.
```

Type denotation function

- Supports constants of unknown any type

Representing Terms

```
Variable ts : list Type.
Inductive expr : Type :=
| Const : ∀ t, tvarD ts t → expr
| Var : nat → expr
| UVar : nat → expr
| Func : nat → list expr → expr
| Equal : tvar → expr → expr → expr.
```

- Supports constants of unknown any type

```
Variable fs : functions ts.
Variable menv venv : env ts.
Fixpoint exprD (e : expr) (t : tvar) :
    option (tvarD ts t) :=
  match e with
    | Const t' c ⇒
      cast_or_fail t t' c
    | Var x ⇒ lookupAs venv t x
    | UVar x ⇒ lookupAs menv t x
    | Func f xs ⇒
      match nth_error fs f with
        | None ⇒ None
        | Some f ⇒
          cast_or_fail t (Range f)
            (applyD exprD _ xs _ (Impl f))
      end
    | Equal t l r ⇒ ...
  end.
```

Representing Terms

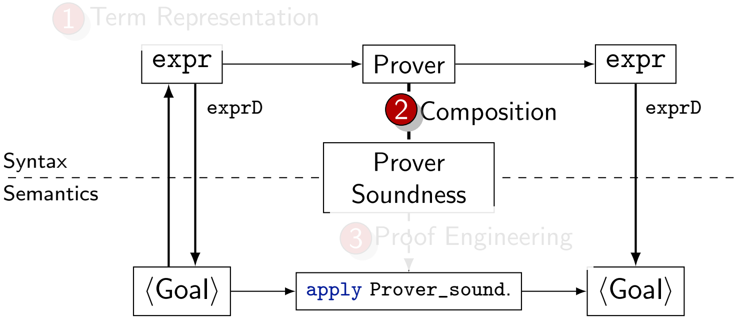
```
Variable ts : list Type.
Inductive expr : Type :=
| Const : ∀ t, tvarD ts t → expr
| Var : nat → expr
| UVar : nat → expr
| Func : nat → list expr → expr
| Equal : tvar → expr → expr → expr.
```

- Supports constants of unknown any type
- Allows ill-formed terms

```
Variable fs : functions ts.
Variable menv venv : env ts.
Fixpoint exprD (e : expr) (t : tvar) :
    option (tvarD ts t) :=
  match e with
    | Const t' c ⇒
      cast_or_fail t t' c
    | Var x ⇒ lookupAs venv t x
    | UVar x ⇒ lookupAs menv t x
    | Func f xs ⇒
      match nth_error fs f with
        | None ⇒ None
        | Some f ⇒
          cast_or_fail t (Range f)
            (applyD exprD _ xs _ (Impl f))
      end
    | Equal t l r ⇒ ...
  end.
```

Partial function

Techniques for Composing Automation



Reflective Procedures

Constant Fold +

```
Variable ts : list Type.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cfp args with
        | 0 , [Const 0 l, Const 0 r] ⇒
          Const 0 (l + r)
        | _ , args ⇒ Func f args
      end
  end.
```

Reflective Procedures

Constant Fold +

```
Variable ts : list Type.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cfp args with
        | 0 , [Const 0 l, Const 0 r] ⇒
          Const 0 (l + r)
        | _ , args ⇒ Func f args
      e
  end.
```

Bad Type! tvarD ts 0

Reflective Procedures

Consta Easy fix...

```
Let ts := nat :: nil.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cfp args with
        | 0 , [Const 0 l, Const 0 r] ⇒
          Const 0 (l + r)
        | _ , args ⇒ Func f args
      end
  end.
```

Reflective Procedures

Consta Easy fix...

```
Let ts := nat :: nil.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cfp args with
        | 0 , [Const 0 l, Const 0 r] ⇒
          Const 0 (l + r)
        | _ , args ⇒ Func f args
      end
  end.
```

Constant Fold <

```
Let ts := nat :: bool :: nil.
Fixpoint cflt (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cflt args with
        | 1 , [Const 0 l, Const 0 r] ⇒
          Const 1 (ltb l r)
        | _ , args ⇒ Func f args
      end
  end.
```

...but it doesn't compose any more!

Reflective Procedures

Consta[nt Fold +]

Easy fix...

```
Let ts := nat :: nil.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cfp args with
        | 0 , [Const 0 l, Const 0 r] ⇒
          Const 0 (l + r)
        | _ , args ⇒ Func f args
      end
  end.
```

Constant Fold <

```
Let ts := nat :: bool :: nil.
Fixpoint cflt (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cflt args with
        | 1 , [Const 0 l, Const 0 r] ⇒
          Const 1 (ltb l r)
        | _ , args ⇒ Func f args
      end
  end.
```

...but it doesn't compose any more!

Composition

```
Definition compose T (f g : T → T) : T → T := fun x ⇒ f (g x).
```

Achieving Composition

- Need to state "all environments with nat at 0"

Propositional

```
Variable ts : list Type.
Hypothesis natAt0 : tvarD 0 ts = nat.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
  | Const _ _ | Var _ | UVar _ ⇒ e
  | Func f args ⇒
    match f , map red args with
    | 0 , [Const 0 l, Const 0 r] ⇒
      match natAt0 in _ = t
        return t → t → expr ts with
      | refl_equal ⇒ fun l r ⇒
        Const 0 (match sym_eq natAt0
                    in _ = t return t with
                  | refl_equal ⇒ (l + r)
                  end)
      end l r
    | _ , args ⇒ Func f args
    end
  end.
```

Achieving Composition

- Need to state … nat at 0"

Update ts' to satisfy the requirement

Update

```
Variable ts' : list Type.
Let ts := repr (Some nat :: nil) ts'.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
  | Const _ _ | Var _ | UVar _ ⇒ e
  | Func f args ⇒
    match f , map red args with
    | 0 , [Const 0 l, Const 0 r] ⇒
      match natAt0 in _ = t
        return t → t → expr ts with
      | refl_equal ⇒ fun l r ⇒
        Const 0 (match sym_eq natAt0
                   in _ = t return t with
                 | refl_equal ⇒ (l + r)
                 end)
      end l r
    | _ , args ⇒ Func f args
    end
  end.
```

Achieving Composition

- Need to state "all environments with nat at 0"

Update

```
Variable ts' : list Type.
Let ts := repr (Some nat :: nil) ts'.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
  | Const _ _ | Var _ | UVar _ ⇒ e
  | Func f args ⇒
    match f , map red args with
    | 0 , [Const 0 l, Const 0 r] ⇒
      match natAt0 in _ = t
        return t → t → expr ts with
      | refl_equal ⇒ fun l r ⇒
        Const 0 (match sym_eq natAt0
                   in _ = t return t with
                 | refl_equal ⇒ (l + r)
                 end)
      end l r
    | _ , args ⇒ Func f args
    end
  end.
```

Environment Constraints

```
Variable T : Type.
Variable default : T.

Fixpoint repr (rep : list (option T)) :
    list T → list T :=
  match rep with
    | nil ⇒ fun x ⇒ x
    | None :: rep ⇒ fun x ⇒
      hd default x :: repr rep (tl x)
    | Some v :: rep ⇒ fun x ⇒
      v :: repr rep (tl x)
  end.
```

Achieving Composition

- Need to state "all environments with nat at 0"

Update

```
Variable ts' : list Type.
Let ts := repr (Some nat :: nil) ts'.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
  | Const _ _ | Var _ | UVar _ ⇒ e
  | Func f args ⇒
    match f , map red args with
    | 0 , [Const 0 l, Const 0 r] ⇒
      match natAt0 in _ = t
        return t → t → expr ts with
      | refl_equal ⇒ fun l r ⇒
        Const 0 (match sym_eq natAt0
                    in _ = t return t with
                  | refl_equal ⇒ (l + r)
                  end)
      end l r
    | _ , args ⇒ Func f args
    end
  end.
```

Environment Constraints

```
Variable T : Type.
Variable default : T.

Fixpoint repr (rep : list (option T)) :
    list T → list T :=
  match rep with
    | nil ⇒ fun x ⇒ x
    | None :: rep ⇒ fun x ⇒
      hd default x :: repr rep (tl x)
    | Some v :: rep ⇒ fun x ⇒
      v :: repr rep (tl x)
  end.
```

Properties

$$\mathrm{repr}\ l\ (\mathrm{repr}\ r\ ls) = \mathrm{repr}\ r\ (\mathrm{repr}\ l\ ls)$$
$$\mathrm{repr}\ l\ (\mathrm{repr}\ l\ ls) = \mathrm{repr}\ l\ ls$$

Achieving Composition

- Need to state "all environments with nat at 0"

Update

```
Variable ts' : list Type.
Let ts := repr (Some nat :: nil) ts'.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
  | Const _ _ | Var _ | UVar _ ⇒ e
  | Func f args ⇒
    match f , map red args with
    | 0 , [Const 0 l, Const 0 r] ⇒
      match natAt0 in _ = t
        return t → t → expr ts with
        | refl_equal ⇒ fun l r ⇒
        Const 0 (match sym_eq natAt0
                   in _ = t return t with
                   | refl_equal ⇒ (l + r)
                   end)
      end l r
    | _ , args ⇒ Func f args
    end
  end.
```

Environment Constraints

```
Variable T : Type.
Variable default : T.

Fixpoint re                                ) :
    list T –
  match rep
    | nil ⇒ fun x ⇒ .
    | None :: rep ⇒ fun x ⇒
      hd default x :: repr rep (tl x)
    | Some v :: rep ⇒ fun x ⇒
      v :: repr rep (tl x)
  end.
```

Implementation avoids getting stuck on variables.

Hold computationally!

Properties

$$\text{repr } l \,(\text{repr } r\ ls) = \text{repr } r \,(\text{repr } l\ ls)$$
$$\text{repr } l \,(\text{repr } l\ ls) = \text{repr } l\ ls$$

Achieving Composition

- Need to state "all environments with nat at 0"

Update

```
Variable ts' : list Type.
Let ts := repr (Some nat ::
Fixpoint cfp (e : expr ts) :
  match e with
  | Const _ _ | Var _ | UVa
  | Func f args ⇒
    match f , map red args w
    | 0 , [Const 0 l, Const 0 r] ⇒
      match natAt0 in _ = t
        return t → t → expr ts with
      | refl_equal ⇒ fun l r ⇒
        Const 0 (match sym_eq natAt0
                   in _ = t return t with
                 | refl_equal ⇒ (l + r)
                 end)
      end l r
    | _ , args ⇒ Func f args
    end
  end.
```

Requires that the lookup computation depends only on the indexed position.

Environment Constraints

```
Variable T : Type.
         default : T.
```

Implementation avoids getting stuck on variables.

```
        None :: rep ⇒ fun x ⇒
        hd default x :: repr rep (tl x)
      | Some v :: rep ⇒ fun x ⇒
        v :: repr rep (tl x)
  end.
```

Hold computationally!

Properties

$$\mathrm{repr}\ l\ (\mathrm{repr}\ r\ ls) = \mathrm{repr}\ r\ (\mathrm{repr}\ l\ ls)$$

$$\mathrm{repr}\ l\ (\mathrm{repr}\ l\ ls) = \mathrm{repr}\ l\ ls$$

Composition with repr

```
Definition rep_plus := [Some nat].
Definition cfp : ∀ ts',
  let ts := repr rep_plus ts' in
  expr ts → expr ts.

fun ts' ⇒ cfp (repr rep_lt ts')
: ∀ ts',
  let ts:=repr rep_plus (repr rep_lt ts')
  in expr ts → expr ts.
```

```
Definition rep_lt:=[Some nat,Some bool].
Definition cflt : ∀ ts',
  let ts := repr rep_lt ts' in
  expr ts → expr ts.

fun ts' ⇒ cflt (repr rep_plus ts')
: ∀ ts',
  let ts:=repr rep_lt (repr rep_plus ts')
  in expr ts → expr ts.
```

Composition with `repr`

```
Definition rep_plus := [Some nat].
Definition cfp : ∀ ts',
  let ts := repr rep_plus ts' in
  expr ts → expr ts.

fun ts' ⇒ cfp (repr rep_lt ts')
: ∀ ts',
  let ts:=repr rep_plus (repr rep_lt ts')
  in expr ts → expr ts.
```

```
Definition rep_lt:=[Some nat,Some bool].
Definition cflt : ∀ ts',
  let ts := repr rep_lt ts' in
  expr ts → expr ts.

fun ts' ⇒ cflt (repr rep_plus ts')
: ∀ ts',
  let ts:=repr rep_lt (repr rep_plus ts')
  in expr ts → expr ts.
```

Composition with `repr`

```
Definition rep_plus := [Some nat].
Definition cfp : ∀ ts',
  let ts := repr rep_plus ts' in
  expr ts → expr ts.

fun ts' ⇒ cfp (repr rep_lt ts')
: ∀ ts',
  let ts:=repr rep_plus (repr rep_lt ts')
  in expr ts → expr ts.
```

```
Definition rep_lt:=[Some nat,Some bool].
Definition cflt : ∀ ts',
  let ts := repr rep_lt ts' in
  expr ts → expr ts.

fun ts' ⇒ cflt (repr rep_plus ts')
: ∀ ts',
  let ts:=repr rep_lt (repr rep_plus ts')
  in expr ts → expr ts.
```

$$\text{repr } l \,(\text{repr } r \; ls) = \text{repr } r \,(\text{repr } l \; ls)$$

Composition with `repr`

```
Definition rep_plus := [Some nat].
Definition cfp : ∀ ts',
  let ts := repr rep_plus ts' in
  expr ts → expr ts.

fun ts' ⇒ cfp (repr rep_lt ts')
: ∀ ts',
  let ts:=repr rep_plus (repr rep_lt ts')
  in expr ts → expr ts.
```

```
Definition rep_lt:=[Some nat,Some bool].
Definition cflt : ∀ ts',
  let ts := repr rep_lt ts' in
  expr ts → expr ts.

fun ts' ⇒ cflt (repr rep_plus ts')
: ∀ ts',
  let ts:=repr rep_lt (repr rep_plus ts')
  in expr ts → expr ts.
```

$$\text{repr } l \, (\text{repr } r \; ls) = \text{repr } r \, (\text{repr } l \; ls)$$

```
fun ts ⇒ @compose (expr (repr (rep_combine rep_plus rep_lt)  ts))
                  (cfp (repr rep_lt ts)) (cflt (repr rep_plus ts))
: ∀ ts, repr (repr (rep_combine rep_plus rep_lt) ts)
```

Composition with repr

```
Definition rep_plus := [Some nat].
Definition cfp : ∀ ts',
  let ts := repr rep_plus ts' in
  expr ts → expr ts.

fun ts' ⇒ cfp (repr rep_lt ts')
: ∀ ts',
  let ts:=repr rep_plus (repr rep_lt ts')
  in expr ts → expr ts.
```

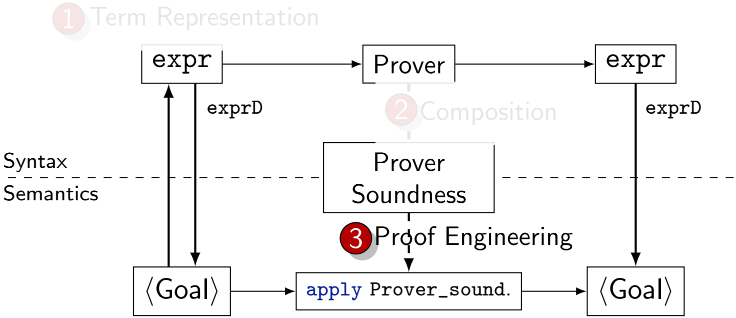
```
Definition rep_lt:=[Some nat,Some bool].
Definition cflt : ∀ ts',
  let ts := repr rep_lt ts' in
  expr ts → expr ts.

fun ts' ⇒ cflt (repr rep_plus ts')
: ∀ ts',
  let ts:=repr rep_lt (repr rep_plus ts')
  in expr ts → expr ts.
```

```
fun ts ⇒ @compose (expr (repr (rep_combine α β) ts))
                  (cfp (repr β ts)) (cflt (repr α ts))
: ∀ ts, repr (repr (rep_combine α β) ts)
```

Well-typed if α and β are **computationally** compatible.

Techniques for Composing Automation



Proof Terms: Traditional Soundness Lemmas

- Performance is significantly affected by the statement of the lemma and the way it is applied

Traditional Soundness

```
Theorem verifyOk : ∀ x y,
  P x →
  verify x y = true →
  denote x y.
```

$$\dfrac{\dfrac{\ldots}{\texttt{P x y}} \qquad \dfrac{\dfrac{}{\texttt{true = true}}\ \textsc{Refl}}{\texttt{verify x y = true}}\ \textsc{Conv}}{\texttt{denote x y}}\ \texttt{verifyOk}$$

Proof Terms: Soundness Lemmas for Simplification

- Procedures that simplify goals are more complex

Traditional Soundness

```
Theorem verifyOk : ∀ x y,
  P x y →
  verify x y = true →
  denote x y.
```

Simplification Soundness

```
Theorem simplifyOk : ∀ x,
  P x → ∀ y,
    simpilfy x = y →
    denote y →
    denote x.
```

$$\dfrac{\dfrac{\ldots}{\texttt{P x y}} \qquad \dfrac{\overline{\texttt{true} = \texttt{true}}}{\texttt{verify x y} = \texttt{true}}}{\texttt{denote x y}}$$

Proof Terms: Soundness Lemmas for Simplification

- Procedures that simplify goals are more complex

Traditional Soundness

```
Theorem verifyOk : ∀ x y,
  P x y →
  verify x y = true →
  denote x y.
```

Simplification Soundness

```
Theorem simplifyOk : ∀ x,
  P x → ∀ y,
    simplify x = y →
    denote y →
    denote x.
```

$$\dfrac{\dfrac{\ldots}{\texttt{P x y}} \qquad \dfrac{\dfrac{}{\texttt{true = true}}}{\texttt{verify x y = true}}}{\texttt{denote x y}}$$

$$\dfrac{\dfrac{\ldots}{\texttt{P x}} \qquad \dfrac{\dfrac{}{\texttt{... = ?1}}}{\texttt{verify x = ?1}} \qquad \dfrac{\vdots}{\texttt{denote ?1}}}{\texttt{denote x}}$$

Proof Terms: Soundness Lemmas for Simplification

- Procedures that simplify goals are more complex

Traditional Soundness

```
Theorem verifyOk : ∀ x y,
  P x y →
  verify x y = true →
  denote x y.
```

Simplification Soundness

```
Theorem simplifyOk : ∀ x,
  P x → ∀ y,
    simplify x = y →
    denote y →
    denote x.
```

$$\dfrac{\dfrac{\ldots}{\texttt{P x y}} \qquad \dfrac{\dfrac{}{\texttt{true} = \texttt{true}}}{\texttt{verify x y} = \texttt{true}}}{\texttt{denote x y}}$$

Presented to the user

$$\dfrac{\dfrac{\ldots}{\texttt{P x}} \qquad \dfrac{\dfrac{}{\ldots = ?1}}{\texttt{verify x} = ?1} \qquad \dfrac{\vdots}{\texttt{denote } ?1}}{\texttt{denote x}}$$

?1 is unknown until we solve this

Proof Terms: Soundness Lemmas for Simplification

- Procedures that simplify goals are more complex

Traditional Soundness

```
Theorem verifyOk : ∀ x y,
  P x y →
  verify x y = true →
  denote x y.
```

Sim… Result embedded in proof!

```
Theorem simplifyOk : ∀ x,
  P x → ∀ y,
    simplify x = y →
    denote y →
    denote x.
```

$$\dfrac{\dfrac{\ldots}{\texttt{P x y}} \qquad \dfrac{\dfrac{}{\texttt{true} = \texttt{true}}}{\texttt{verify x y} = \texttt{true}}}{\texttt{denote x y}}$$

$$\dfrac{\dfrac{\ldots}{\texttt{P x}} \qquad \dfrac{\dfrac{}{\ldots = ?1}}{\texttt{verify x} = ?1} \qquad \dfrac{\vdots}{\texttt{denote } ?1}}{\texttt{denote x}}$$

Proof Terms: Simplifying Soundness Lemmas (Alt)

- Avoiding large terms in proofs can be important
  - The indicies of our dependent representation make it large!

Alternate Soundness

```
Theorem simplifyOk' : ∀ x,
  P x →
  (let y := simpilfy x in
   denote y) →
  denote x.
```

Simplification Soundness

```
Theorem simplifyOk : ∀ x,
  P x → ∀ y,
    simpilfy x = y →
    denote y →
    denote x.
```

$$\dfrac{\dfrac{\ldots}{\texttt{P x}} \quad \dfrac{\overline{\ldots = ?1}}{\texttt{verify x} = ?1} \quad \dfrac{\vdots}{\texttt{denote ?1}}}{\texttt{denote x}}$$

Proof Terms: Simplifying Soundness Lemmas (Alt)

- Avoiding large terms in proofs can be important
  - The indicies of our dependent representation make it large!

Alternate Soundness

```
Theorem simplifyOk' : ∀ x,
  P x →
  (let y := simpilfy x in
   denote y) →
  denote x.
```

let-bind result

Simplification Soundness

```
Theorem simplifyOk : ∀ x,
  P x → ∀ y,
    simpilfy x = y →
    denote y →
    denote x.
```

$$\dfrac{\dfrac{\ldots}{\texttt{P x}} \qquad \dfrac{\dfrac{}{\ldots = ?1}}{\texttt{verify x} = ?1} \qquad \dfrac{\vdots}{\texttt{denote } ?1}}{\texttt{denote x}}$$

Proof Terms: Simplifying Soundness Lemmas (Alt)

- Avoiding large terms in proofs can be important
  - The indicies of our dependent representation make it large!

Alternate Soundness

```
Theorem simplifyOk' : ∀ x,
  P x →
  (let y := simpilfy x in
   denote y) →
  denote x.
```

Simplification Soundness

```
Theorem simplifyOk : ∀ x,
  P x → ∀ y,
    simpilfy x = y →
    denote y →
    denote x.
```

Reduce before continuing

$$\frac{\dfrac{\ldots}{\texttt{P x}} \qquad \dfrac{\vdots}{\texttt{let y := simplify x in denote y}}}{\texttt{denote x}}$$

$$\frac{\dfrac{\ldots}{\texttt{P x}} \qquad \dfrac{\dfrac{}{\ldots = ?1}}{\texttt{verify x} = ?1} \qquad \dfrac{\vdots}{\texttt{denote } ?1}}{\texttt{denote x}}$$

Proof Terms: Simplifying Soundness Lemmas (Alt)

- Avoiding large terms in proofs can be important
  - The indicies of our dependent representation make it large!

Alternate Soundness

```
Theorem simplifyOk' : ∀ x,
  P x →
  (let y := simpilfy x in
   denote y) →
  denote x.
```

Simplification Soundness

```
Theorem simplifyOk : ∀ x,
  P x → ∀ y,
    simpilfy x = y →
    denote y →
    denote x.
```

Reduce before continuing

$$\dfrac{\dfrac{\ldots}{\texttt{P x}} \qquad \dfrac{\vdots}{\texttt{let y := simplify x in denote y}}}{\texttt{denote x}}$$

$$\dfrac{\dfrac{\ldots}{\texttt{P x}} \qquad \dfrac{\overline{\ldots = ?1}}{\texttt{verify x} = ?1} \qquad \dfrac{\vdots}{\texttt{denote } ?1}}{\texttt{denote x}}$$

- Important to keep some symbols opaque; otherwise the resulting term has no abstraction!

Outline

1. Reflective Proofs
2. Automation in BEDROCK
3. Techniques for Composable Reflective Procedures
   - Term Representation
   - Composition
   - Proof Term Engineering
4. Moving Forward

What's Next?

- How much automation can we achieve?
  - Decidability will always be a problem
  - Ltac integration makes it easy to code heuristics!

What's Next?

- How much automation can we achieve?
    - Decidability will always be a problem
    - Ltac integration makes it easy to code heuristics!
- Performance implications
    - log factor for functional data structures
    - Ahead-of-time compilation/optimization?

What's Next?

- How much automation can we achieve?
    - Decidability will always be a problem
    - Ltac integration makes it easy to code heuristics!
- Performance implications
    - log factor for functional data structures
    - Ahead-of-time compilation/optimization?
- How can we lower the burden of writing this code?
    - Inverting a complex denotation function into fast reification is cumbersome (we wrote a plugin)
    - Would a single syntax be sufficient?

Credits

None of this work would be possible without my collaborators & advisors:

Thomas Braibant
Adam Chlipala
Greg Morrisett

Proof Terms: Reduction

- Reflective procedures need **fast** evaluation!
  - Procedures are often large & complex
- Four evaluation strategies in Coq
  - `hnf` – head-normal-form
  - `lazy`/`simpl` – call-by-need evaluation (with selective reduction)
  - `cbv` – call-by-value evaluation
  - `vm_compute` – call-by-value evaluation with byte-code compilation

vm_compute

- Currently the fastest evaluation strategy
- Properties
  - ✓ Implemented in the kernel
  - ✗ Does not accept terms with unification variables
  - ✗ Unfolds everything it can!
    - Need to refold to keep working with the term.

Traditional Soundness

```
Theorem verifyOk : ∀ x y,
  P x y →
  verify x y = true →
  denote x = denote y.
```

✓ Works great!

Simplification Soundness

```
Theorem simplifyOk : ∀ x y,
  P x →
  simpilfy x = y →
  denote y → denote x.
```

✗ Unification variables :'(

cbv

- Slower than `vm_compute` but still much faster than `lazy`/`simpl`/`hnf`

✓ Can customize reduction
  - Keep certain symbols opaque based on blacklist (suboptimal)
  - Avoid manifesting the result of the computation

✗ This meta information is not stored in the proof (can make proof checking slow)
  - Often need to include an explicit cast in the proof term to record what is expected from reduction
  - Checking proof term still uses `lazy`