Bedrock: A Framework for Verifying Low-level Programs

Gregory Malecha (gmalecha@cs.harvard.edu)
Adam Chlipala, Thomas Braibant, Patrick Hulin, Edward Yang (MIT)

Harvard University SEAS

IBM PL Day '12 – June 28

Type Safety Isn't Always Enough!

```
let getFirst (buf : 'a array) : 'a option :=
  let len = Array.length buf in
  if len = 0 then None
  else Some (Array.get buf (len - 1))
```

Type Safety Isn't Always Enough!

```
let getFirst (buf : 'a array) : 'a option :=
  let len = Array.length buf in
  if len = 0 then None
  else Some (Array.get buf (len - 1))
```

Why not negative?

Type Safety Isn't Always Enough!

```
let getFirst (buf : 'a array) : 'a option :=
  let len = Array.length buf in
  if len = 0 then None
  else Some (Array.get buf (len - 1))
```

Why not negative?

Index out of bounds?

Type Safety Isn't Always Enough!

Most informative text!

```
let getFirst (buf : 'a array) : 'a option :=
  let len = Array.length buf in
  if len = 0 then None
  else Some (Array.get buf (len - 1))
```

Why not negative?

Index out of bounds?

Type Safety Isn't Always Enough!

Most informative text! oops!

```
let getLast (buf : 'a array) : 'a option :=
  let len = Array.length buf in
  if len = 0 then None
  else Some (Array.get buf (len − 1))
```

Why not negative?

Index out of bounds?

Project Goals

A programming language for systems code that supports

- verification down to the machine code ...

Project Goals

A programming language for systems code that supports

- verification down to the machine code ...
- of low-level code ...

Project Goals

A programming language for systems code that supports

- verification down to the machine code ...
- of low-level code ...
- with a small trusted computing base ...

Project Goals

A programming language for systems code that supports

- verification down to the machine code ...
- of low-level code ...
- with a small trusted computing base ...
- while supporting higher-order specifications ...

Project Goals

A programming language for systems code that supports

- verification down to the machine code ...
- of low-level code ...
- with a small trusted computing base ...
- while supporting higher-order specifications ...
- and is both extensible ...

Project Goals

A programming language for systems code that supports

- verification down to the machine code ...
- of low-level code ...
- with a small trusted computing base ...
- while supporting higher-order specifications ...
- and is both extensible ...
- and reasonable to program with.

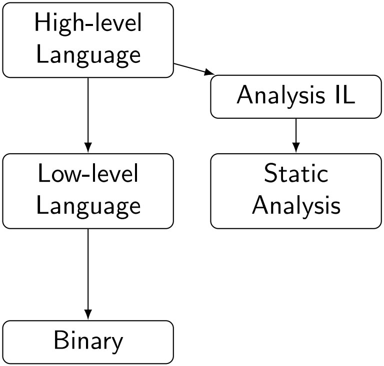
Project Goals

A programming language for systems code that supports

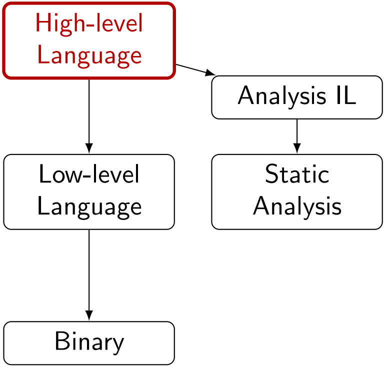
- verification down to the machine code ...
- of low-level code ...
- with a small trusted computing base ...
- while supporting higher-order specifications ...
- and is both extensible ...
- and reasonable to program with.

Bedrock

Typical Program Verification

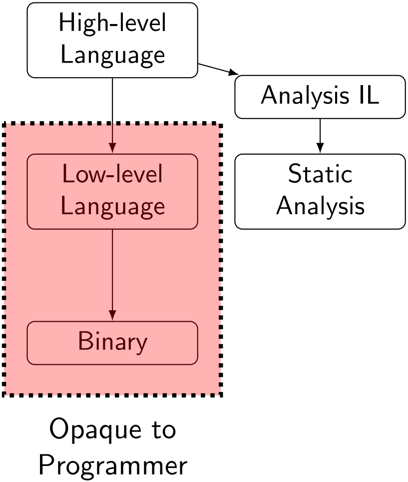


Typical Program Verification



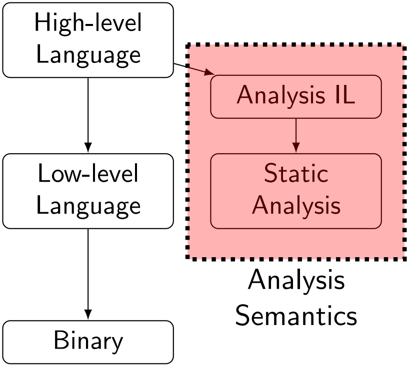
- Focus is on the high-level language!

Typical Program Verification



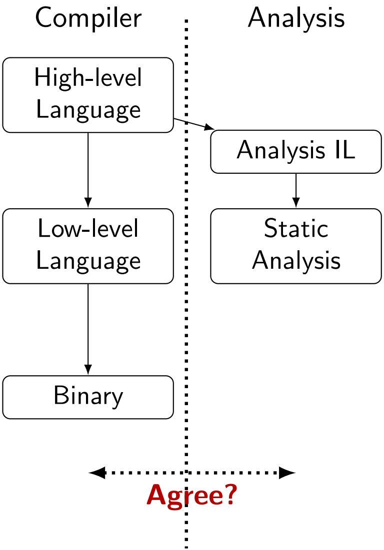
- Focus is on the high-level language!
- Compiler focuses on converting to a binary.

Typical Program Verification



- Focus is on the high-level language!
- Compiler focuses on converting to a binary.

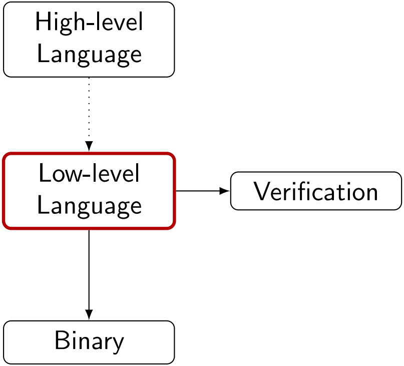
Typical Program Verification



- Focus is on the high-level language!
- Compiler focuses on converting to a binary.
- Need the same semantics.

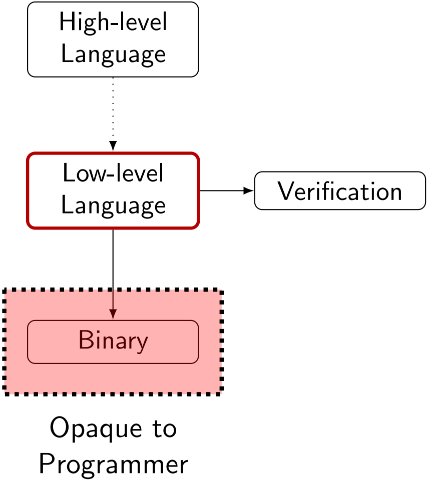
Verification in Bedrock

- Focus is on a low-level language.



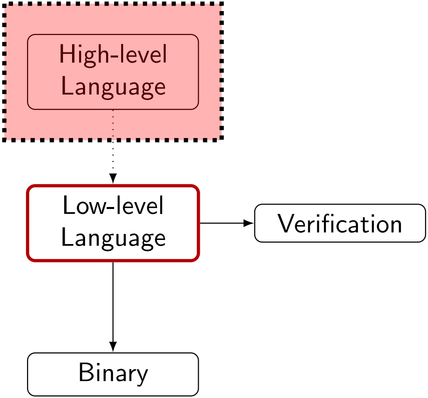
Verification in Bedrock

- Focus is on a low-level language.
- Minimal details hidden by the framework.



Verification in Bedrock

- Focus is on a low-level language.
- Minimal details hidden by the framework.
- Achieve abstraction by parametrization over the semantics.



Outline

Outline

Verifying a Simple Program

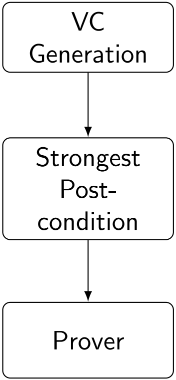
A Simple Program

$\{\ \}$
$\texttt{Rv} := 0$
$\{\ \texttt{st'}\ \sim> \texttt{st'.Rv} = 0)\ \}$

Verifying a Simple Program

A Simple Program

```
{ }
Rv := 0
{ st' ~> st'.Rv = 0) }
```



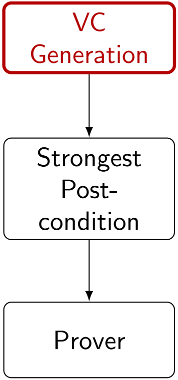
Verifying a Simple Program

A Simple Program

```
{ }
Rv := 0
{ st' ~> st'.Rv = 0) }
```

Strongest Post-condition

```
{ } st
∧ evalInstrs st [ Rv := 0 ] st'
→ { Rv = 0 } st'
```



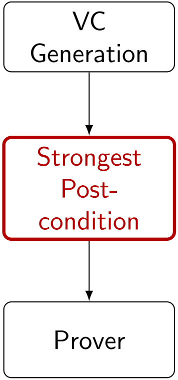
Verifying a Simple Program

A Simple Program

```
{ }
Rv := 0
{ st' ~> st'.Rv = 0) }
```

Strongest Post-condition

```
{ } st
∧ evalInstrs st [ Rv := 0 ] st'
→ { Rv = 0 } st'
```



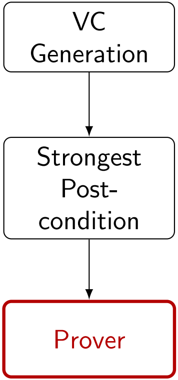
Verifying a Simple Program

A Simple Program

```
{ }
Rv := 0
{ st' ~> st'.Rv = 0) }
```

Proof Obligation

```
{ Rv = 0 } st'
→ { Rv = 0 } st'
```

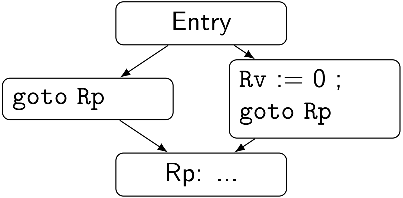


Outline

Bedrock: Extensible Control

Conditional Code

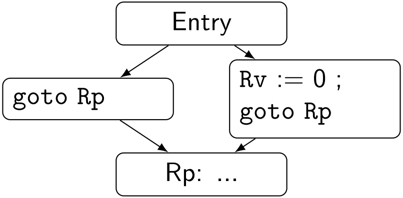
{ Entry : ([], br Rv Eq 0 Tr Fa)
; Tr : ([], goto Rp)
; Fa : ([Rv := 0], goto Rp) }



Bedrock: Extensible Control

Conditional Code

```
{ Entry : ([],        br Rv Eq 0 Tr Fa)
; Tr    : ([],        goto Rp)
; Fa    : ([Rv := 0], goto Rp) }
```

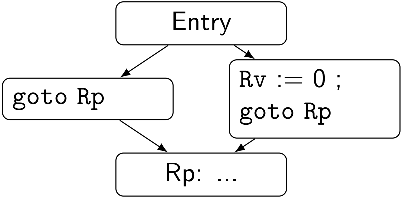


This looks like if!

Bedrock: Extensible Control

Conditional Code

```
{ Entry : ([],         br Rv Eq 0 Tr Fa)
; Tr    : ([],         goto Rp)
; Fa    : ([ Rv := 0], goto Rp) }
```



This looks like if!

Abstract it!

Control Abstraction

- Build syntax combinators in the meta-language

If Combinator

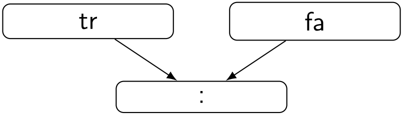
```
Definition If (cmp : Cmp) (l : Rhs) (r : Rhs) (tr : Code) (fa : Code) :
    Code := ...
```

Control Abstraction

- Build syntax combinators in the meta-language

If Combinator

```
Definition If (cmp : Cmp) (l : Rhs) (r : Rhs) (tr : Code) (fa : Code) :
    Code := ...
```

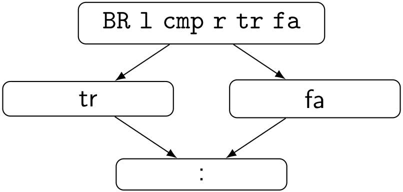


Control Abstraction

- Build syntax combinators in the meta-language

If Combinator

```
Definition If (cmp : Cmp) (l : Rhs) (r : Rhs) (tr : Code) (fa : Code) :
    Code := ...
```

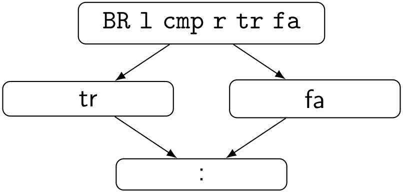


Control Abstraction

- Build syntax combinators in the meta-language

If Combinator

```
Definition If (cmp : Cmp) (l : Rhs) (r : Rhs) (tr : Code) (fa : Code) :
    Code := ...
```



Don't want to reason about this every time

Verified Control Abstraction

- Package the combinator with a proof rule.
- Verify the proof rule once.

Verified Control Abstraction

- Package the combinator with a proof rule.
- Verify the proof rule once.

$$\frac{\{\ \mathsf{P}\ \}\ (\mathsf{cmp}\ \mathsf{l}\ \mathsf{r}) \quad \{\ \mathsf{P} \wedge \mathsf{cmp}\ \mathsf{l}\ \mathsf{r} = \mathsf{true}\}\ \mathsf{tr}\ \{\ \mathsf{Q}\ \} \quad \{\ \mathsf{P} \wedge \mathsf{cmp}\ \mathsf{l}\ \mathsf{r} = \mathsf{false}\}\ \mathsf{fa}\ \{\ \mathsf{Q}\ \}}{\{\ \mathsf{P}\ \}\ \mathsf{if}\ (\mathsf{cmp}\ \mathsf{l}\ \mathsf{r})\ \mathsf{tr}\ \mathsf{fa}\ \{\ \mathsf{Q}\ \}}\ \mathsf{If}$$

Verified Control Abstraction

- Package the combinator with a proof rule.
- Verify the proof rule once.

$$\frac{\{\,\mathsf{P}\,\}\ (\mathsf{cmp\ l\ r}) \quad \{\,\mathsf{P} \wedge \mathsf{cmp\ l\ r} = \mathsf{true}\}\ \mathsf{tr}\ \{\,\mathsf{Q}\,\} \quad \{\,\mathsf{P} \wedge \mathsf{cmp\ l\ r} = \mathsf{false}\}\ \mathsf{fa}\ \{\,\mathsf{Q}\,\}}{\{\,\mathsf{P}\,\}\ \mathsf{if}\ (\mathsf{cmp\ l\ r})\ \mathsf{tr\ fa}\ \{\,\mathsf{Q}\,\}}\ \mathsf{If}$$

If Combinator Sketch

```
let If cmp l r tr fa := fun P ⇒
  let tr := tr (P ∧ cmp l r = true) in
  let fa := fa (P ∧ cmp l r = false) in
  { Ent : L
  ; Blocks : { L : (BR cmp l r tr.Ent fa.Ent) } ∪ tr.Blocks ∪ fa.Blocks
  ; Post : tr.Post ∨ fa.Post
  ; Safe : safeTest l cmp r ∧ tr.Safe ∧ fa.Safe }
```

Verified Control Abstraction

- Package the combinator with a proof rule.
- Verify the proof rule once.

$$\frac{\{\ P\ \}\ (\mathsf{cmp}\ \mathsf{l}\ \mathsf{r}) \quad \{\ P \land \mathsf{cmp}\ \mathsf{l}\ \mathsf{r} = \mathsf{true}\}\ \mathsf{tr}\ \{\ Q\ \} \quad \{\ P \land \mathsf{cmp}\ \mathsf{l}\ \mathsf{r} = \mathsf{false}\}\ \mathsf{fa}\ \{\ Q\ \}}{\{\ P\ \}\ \mathsf{if}\ (\mathsf{cmp}\ \mathsf{l}\ \mathsf{r})\ \mathsf{tr}\ \mathsf{fa}\ \{\ Q\ \}}\ \mathsf{If}$$

Precondition

If Combinator Sketch

```
let If cmp l r tr fa := fun P ⇒
  let tr := tr (P ∧ cmp l r = true) in
  let fa := fa (P ∧ cmp l r = false) in
  { Ent : L
  ; Blocks : { L : (BR cmp l r tr.Ent fa.Ent) } ∪ tr.Blocks ∪ fa.Blocks
  ; Post : tr.Post ∨ fa.Post
  ; Safe : safeTest l cmp r ∧ tr.Safe ∧ fa.Safe }
```

Verified Control Abstraction

- Package the combinator with a proof rule.
- Verify the proof rule once.

$$\frac{\{\ P\ \}\ (\mathsf{cmp}\ \mathsf{l}\ \mathsf{r}) \quad \{\ P \wedge \mathsf{cmp}\ \mathsf{l}\ \mathsf{r} = \mathsf{true}\}\ \mathsf{tr}\ \{\ Q\ \} \quad \{\ P \wedge \mathsf{cmp}\ \mathsf{l}\ \mathsf{r} = \mathsf{false}\}\ \mathsf{fa}\ \{\ Q\ \}}{\{\ P\ \}\ \mathsf{if}\ (\mathsf{cmp}\ \mathsf{l}\ \mathsf{r})\ \mathsf{tr}\ \mathsf{fa}\ \{\ Q\ \}}\ \mathsf{If}$$

If Combinator Sketch

Add branch fact

```
let If cmp l r tr fa := fun P ⇒
  let tr := tr (P ∧ cmp l r = true) in
  let fa := fa (P ∧ cmp l r = false) in
  { Ent : L
  ; Blocks : { L : (BR cmp l r tr.Ent fa.Ent) } ∪ tr.Blocks ∪ fa.Blocks
  ; Post : tr.Post ∨ fa.Post
  ; Safe : safeTest l cmp r ∧ tr.Safe ∧ fa.Safe }
```

Verified Control Abstraction

- Package the combinator with a proof rule.
- Verify the proof rule once.

$$\frac{\{\ \mathsf{P}\ \}\ (\mathsf{cmp\ l\ r}) \quad \{\ \mathsf{P} \wedge \mathsf{cmp\ l\ r} = \mathsf{true}\}\ \mathsf{tr}\ \{\ \mathsf{Q}\ \} \quad \{\ \mathsf{P} \wedge \mathsf{cmp\ l\ r} = \mathsf{false}\}\ \mathsf{fa}\ \{\ \mathsf{Q}\ \}}{\{\ \mathsf{P}\ \}\ \mathsf{if}\ (\mathsf{cmp\ l\ r})\ \mathsf{tr\ fa}\ \{\ \mathsf{Q}\ \}}\ \mathsf{If}$$

If Combinator Sketch

```
let If cmp l r tr fa := fun P ⇒
  let tr := tr (P ∧ cmp l r = true) in
  let fa := fa (P ∧ cmp l r = f
  { Ent : L
  ; Blocks : { L : (BR cmp l r tr.Ent fa.Ent) } ∪ tr.Blocks ∪ fa.Blocks
  ; Post : tr.Post ∨ fa.Post
  ; Safe : safeTest l cmp r ∧ tr.Safe ∧ fa.Safe }
```

Combine the blocks

Verified Control Abstraction

- Package the combinator with a proof rule.
- Verify the proof rule once.

$$\frac{\{\ \mathsf{P}\ \}\ (\mathsf{cmp\ l\ r}) \qquad \{\ \mathsf{P} \wedge \mathsf{cmp\ l\ r} = \mathsf{true}\}\ \mathsf{tr}\ \{\ \mathsf{Q}\ \} \qquad \{\ \mathsf{P} \wedge \mathsf{cmp\ l\ r} = \mathsf{false}\}\ \mathsf{fa}\ \{\ \mathsf{Q}\ \}}{\{\ \mathsf{P}\ \}\ \mathsf{if}\ (\mathsf{cmp\ l\ r})\ \mathsf{tr\ fa}\ \{\ \mathsf{Q}\ \}}\ \mathsf{If}$$

If Combinator Sketch

```
let If cmp l r tr fa := fun P ⇒
  let tr := tr (P ∧ cmp l r = true) in
  let fa := fa (P ∧ cmp l r = false) in
  { Ent : L
  ; Blocks : { L : (BR cmp l r                              Blocks
  ; Post : tr.Post ∨ fa.Post
  ; Safe : safeTest l cmp r ∧ tr.Safe ∧ fa.Safe }
```

Combine post condition

Verified Control Abstraction

- Package the combinator with a proof rule.
- Verify the proof rule once.

$$\frac{\{\ \mathsf{P}\ \}\ (\mathsf{cmp}\ \mathsf{l}\ \mathsf{r}) \qquad \{\ \mathsf{P} \wedge \mathsf{cmp}\ \mathsf{l}\ \mathsf{r} = \mathsf{true}\}\ \mathsf{tr}\ \{\ \mathsf{Q}\ \} \qquad \{\ \mathsf{P} \wedge \mathsf{cmp}\ \mathsf{l}\ \mathsf{r} = \mathsf{false}\}\ \mathsf{fa}\ \{\ \mathsf{Q}\ \}}{\{\ \mathsf{P}\ \}\ \mathsf{if}\ (\mathsf{cmp}\ \mathsf{l}\ \mathsf{r})\ \mathsf{tr}\ \mathsf{fa}\ \{\ \mathsf{Q}\ \}}\ \mathsf{If}$$

If Combinator Sketch

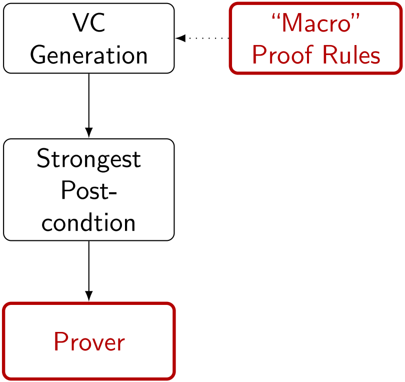
```
let If cmp l r tr fa := fun P ⇒
  let tr := tr (P ∧ cmp l r = true) in
  let fa := fa (P ∧ cmp l r = false) in
  { Ent : L
  ; Blocks : { L : (BR cmp l r tr.Ent fa.Ent) } ∪ tr.Blocks ∪ fa.Blocks
  ; Post : tr.Post ∨ fa.Post
  ; Safe : safeTest l cmp r ∧ tr.Safe ∧ fa.Safe }
```

Combine safety conditions

Verification with Extended Control

Always-0 with Conditionals

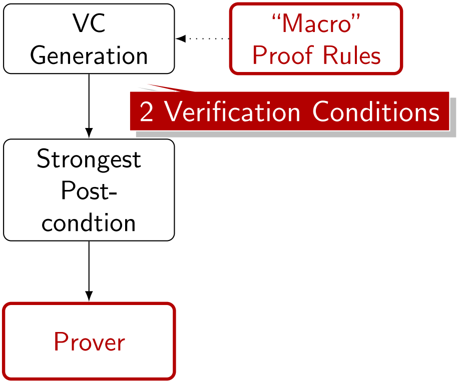
{ }
If (Rv = 0) { skip }
Else { Rv := 0 }
{ st′ ∼>st'.Rv = 0) }



Verification with Extended Control

Always-0 with Conditionals

```
{ }
If (Rv = 0) { skip }
Else { Rv := 0 }
{ st'.Rv = 0) }
```



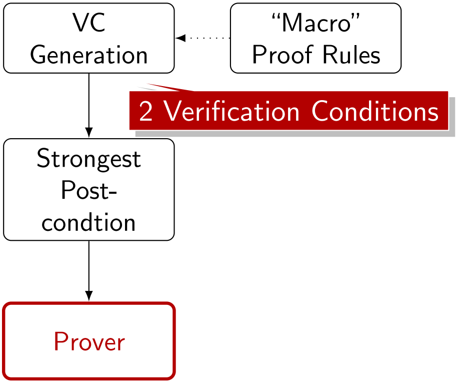
Verification with Extended Control

Always-0 with Conditionals

```
{ }
If (Rv = 0) { skip }
Else { Rv := 0 }
{ st'.Rv = 0) }
```

New VC

```
{ } st
∧ evalCond st (Rv = 0)
→ { Rv = 0 } st
```



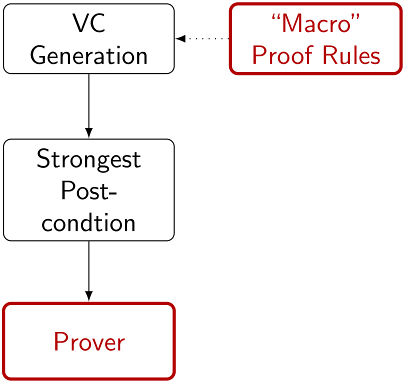
Verification with Extended Control

Always-0 with Conditionals

```
{ }
If (Rv = 0) { skip }
Else { Rv := 0 }
{ st'.Rv = 0) }
```

Proof Obligation

```
{ Rv = 0 } st
→ { Rv = 0 } st
```



Outline

Verification with Memory

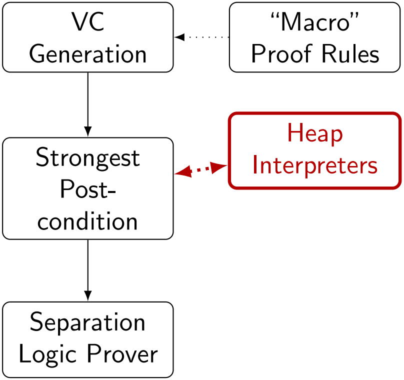
Always-0 with Memory

$\{\ \exists$ v, ![Rv $\mapsto$ v] st $\}$
$[Rv] := 0 ;
$\{$![Rv $\mapsto$ 0] st' $\}$

Verification with Memory

Always-0 with Memory

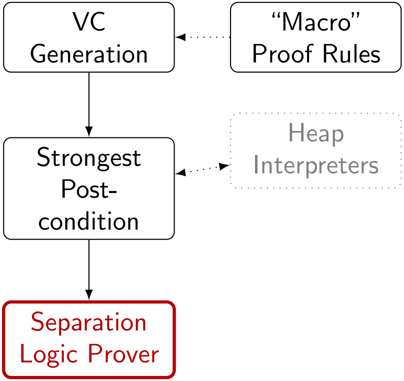
{ ∃ v, ![Rv ↦ v] st }
$[Rv] := 0 ;
{ ![Rv ↦ 0] st' }



Verification with Memory

Always-0 with Memory

{ ∃ v, ![Rv ↦ v] st }
$[Rv] := 0 ;
{ ![Rv ↦ 0] st' }



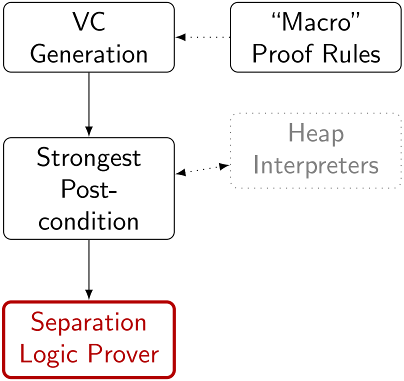
Verification with Memory

Always-0 with Memory

{ ∃ v, ![Rv ↦ v] st }
$[Rv] := 0 ;
{ ![Rv ↦ 0] st' }

Proof Obligation

{ ![Rv ↦ 0] } st
→ { ![Rv ↦ 0] } st



Verification with Memory

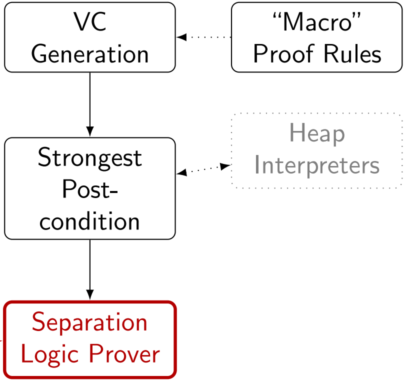
Always-0 with Memory

```
{ ∃ v, ![Rv ↦ v] st }
$[Rv] := 0 ;
{ ![ Rv ↦ 0] st' }
```

Proof Obligation

```
{ ![ Rv ↦ 0] } st
→ { ![Rv ↦ 0] } st
```

```
(True → True) ∧
(Rv ↦ 0) ⇒ (Rv ↦ 0)
```



A Simple Separation Logic Prover

- Solve implications by repeated cancellation

A Simple Goal

$$p_2 \mapsto v_2 * p_1 \mapsto v_1 * P \Rightarrow P * p_1 \mapsto v_1 * p_2 \mapsto v_2$$

A Simple Separation Logic Prover

- Solve implications by repeated cancellation

A Simple Goal

$$p_2 \mapsto v_2 * p_1 \mapsto v_1 \quad \Rightarrow \quad p_1 \mapsto v_1 * p_2 \mapsto v_2$$

A Simple Separation Logic Prover

- Solve implications by repeated cancellation

A Simple Goal

$$p_2 \mapsto v_2 \quad \Rightarrow \quad p_2 \mapsto v_2$$

A Simple Separation Logic Prover

- Solve implications by repeated cancellation

A Simple Goal

$$\emptyset \quad \Rightarrow \quad \emptyset$$

- Proves $\emptyset \Rightarrow \emptyset$ by reflexivity.

Outline

Reasoning about Abstract Data Types: Lists

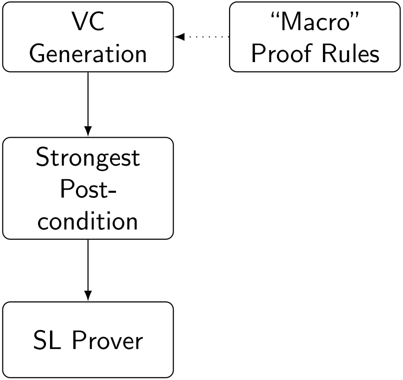
Linked List Head

{ ∃ ls, ![llist Rv ls] st }
If (Rv = 0) { skip }
Else { Rv = $[Rv] } ;
{ ![llist st.Rv ls] st' ∧
    st'. Rv = hd ls }

Reasoning about Abstract Data Types: Lists

Linked List Head

```
{ ∃ ls, ![llist Rv ls] st }
If (Rv = 0) { skip }
Else { Rv = $[Rv] } ;
{ ![ llist st.Rv ls] st' ∧
    st'.Rv = hd ls }
```



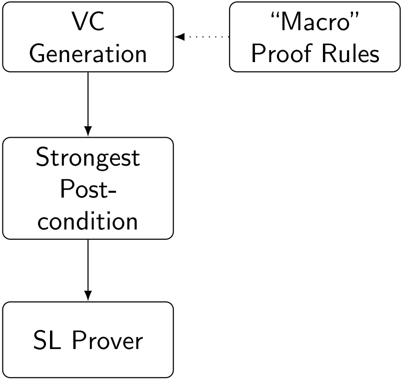
Reasoning about Abstract Data Types: Lists

Linked List Head

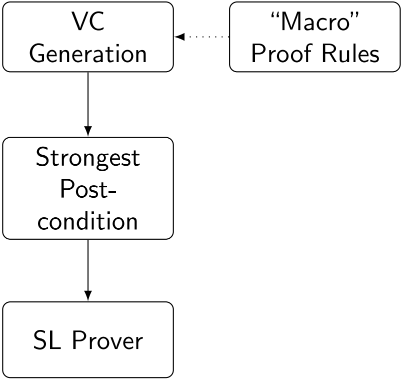
$\{ \exists\ \texttt{ls},\ ![\texttt{llist Rv ls}]\ \texttt{st} \}$
$\texttt{If}\ (\texttt{Rv} = 0)\ \{ \texttt{skip} \}$
$\texttt{Else}\ \{ \texttt{Rv} = \$[\texttt{Rv}] \}\ ;$
$\{ ![\ \texttt{llist st.Rv ls}]\ \texttt{st'} \wedge$
$\quad \texttt{st'.Rv} = \texttt{hd ls} \}$

Post Condition

$\{\exists\ \texttt{ls},\ ![\text{llist Rv ls}] \wedge \texttt{Rv} \neq 0\}\ \texttt{st}$
$\wedge\ \texttt{evalInstrs st}\ [\texttt{Rv} := \$[\texttt{Rv}]]\ \texttt{st'}$
$\rightarrow\ \{\exists\ \texttt{ls},\ ![\texttt{llist st.Rv ls}]$
$\quad \wedge\ \texttt{st'.Rv} = \texttt{hd ls}\}\ \texttt{st'}$



Reasoning about Abstract Data Types: Lists



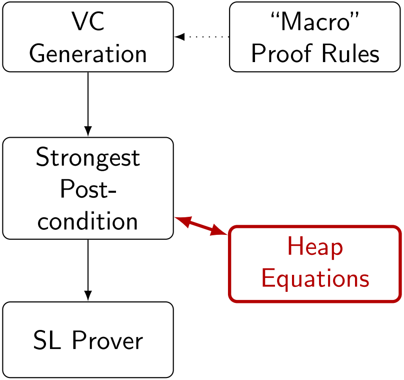
Linked List Head

```
{ ∃ ls, ![llist Rv ls] st }
If (Rv = 0) { skip }
Else { Rv = $[Rv] } ;
{ ![ llist st.Rv ls] st' ∧
    st'.Rv = hd ls }
```

Post Condition

```
{∃ ls, ![llist Rv ls] ∧ Rv ≠ 0} st
∧ evalInstrs st [Rv := $[Rv]] st'
→ {∃ ls, ![llist st.Rv ls]
    ∧ st'.Rv = hd ls} st'
```

Reasoning about Abstract Data Types: Lists



Linked List Head

```
{ ∃ ls, ![llist Rv ls] st }
If (Rv = 0) { skip }
Else { Rv = $[Rv] } ;
{ ![ llist st.Rv ls] st' ∧
    st'.Rv = hd ls }
```

Post Condition

```
{∃ ls, ![llist Rv ls] ∧ Rv ≠ 0} st
∧ evalInstrs st [Rv := $[Rv]] st'
→ {∃ ls, ![llist st.Rv ls]
    ∧ st'.Rv = hd ls} st'
```

Stuck!

```
∀ p ls, p <> 0 →
llist p ls ⇒ ∃ v p' ls', p ↦ v ∗
  p+4 ↦ p' ∗ llist p' ls' ∗
  ls = v :: ls'
```

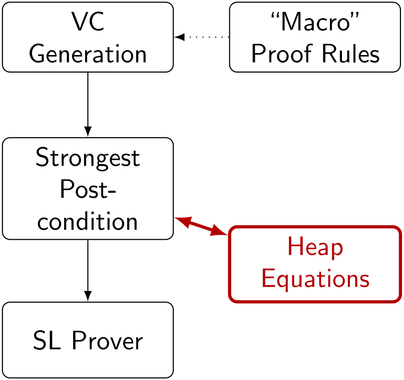
Reasoning about Abstract Data Types: Lists

Linked List Head

{ ∃ ls, ![llist Rv ls] st }
If (Rv = 0) { skip }
Else { Rv = $[Rv] } ;
{ ![llist st.Rv ls] st' ∧ st'.Rv = hd ls }

Symbolic Evaluation

{∃ p' v ls', ls = v :: ls' ∧ Rv≠0 ∧
![Rv ↦ v ∗ Rv + 4 ↦ p' ∗ llist p' ls']} st
∧ evalInstrs st [Rv := $[Rv]] st'
→ {∃ ls, ![llist st.Rv ls]
∧ st'.Rv = hd ls} st'



∀ p ls, p ≠ 0 →
llist p ls ⇒ ∃ v p' ls', p ↦ v ∗ p+4 ↦ p' ∗ llist p' ls'

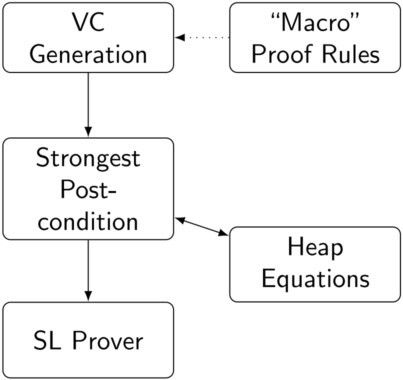
Reasoning about Abstract Data Types: Lists

Linked List Head

```
{ ∃ ls, ![llist Rv ls] st }
If (Rv = 0) { skip }
Else { Rv = $[Rv] } ;
{ ![ llist st.Rv ls] st' ∧ st'.Rv = hd
    ls }
```

Proof Obligation

```
{∃ ls p' v ls', ls=v::ls' ∧ Rv=v ∧
![ Rv ↦ v ∗ Rv+4 ↦ p' ∗
  llist p' ls']} st'
→ {∃ ls p' v ls', Rv = hd ls ∧
    ![ llist st.Rv ls]} st'
```



Reasoning about Abstract Data Types: Lists

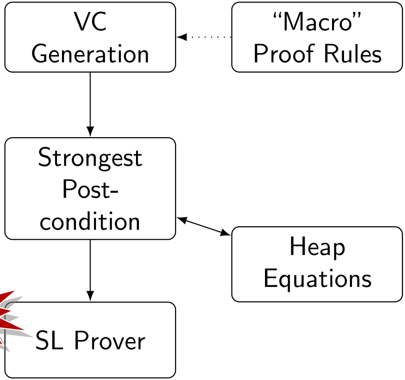
Linked List Head

{ ∃ ls, ![llist Rv ls] st }
If (Rv = 0) { skip }
Else { Rv = $[Rv] } ;
{ ![llist st.Rv ls] st' ∧ st'.Rv = hd ls }

Proof Obligation

Stuck!

{∃ ls p' v ls', ls=v::ls' ∧ Rv=v ∧
![Rv ↦ v ∗ Rv+4 ↦ p' ∗
llist p' ls']} st'
→ {∃ ls p' v ls', Rv = hd ls ∧
![llist st.Rv ls]} st'



Reasoning about Abstract Data Types: Lists

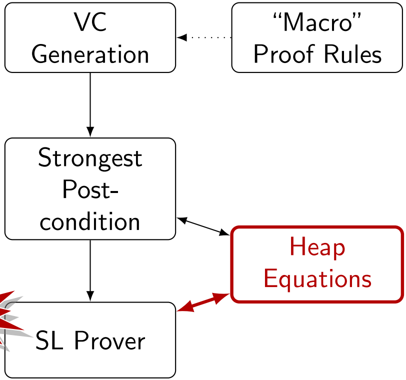
Linked List Head

{ ∃ ls, ![llist Rv ls] st }
If (Rv = 0) { skip }
Else { Rv = $[Rv] } ;
{ ![llist st.Rv ls] st' ∧ st'.Rv = hd ls }

Proof Obligation

Stuck!

{∃ ls p' v ls', ls=v::ls' ∧ Rv=v ∧
![Rv ↦ v ∗ Rv+4 ↦ p' ∗
llist p' ls']} st'
→ {∃ ls p' v ls', Rv = hd ls ∧
![llist st.Rv ls]} st'



∀ p ls, p ≠ 0 →
∃ v p' ls', p ↦ v ∗ p+4 ↦ p' ∗
llist p' ls' ⇒ llist p ls

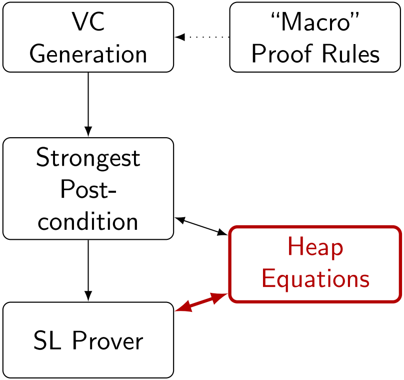
Reasoning about Abstract Data Types: Lists

Linked List Head

{ ∃ ls, ![llist Rv ls] st }
If (Rv = 0) { skip }
Else { Rv = $[Rv] }
{ ![llist st.Rv ls] st' ∧ st'.Rv = hd ls }

Proof Obligation

{∃ ls p' v ls', ls=v::ls' ∧ Rv=v ∧
![Rv ↦ v ∗ Rv+4 ↦ p' ∗
llist p' ls']} st' →
{∃ ls p' v ls', Rv = hd ls ∧
ls = v :: ls' ∧
![Rv ↦ v ∗ Rv + 4 ↦ p' ∗ llist p' ls']}
st'



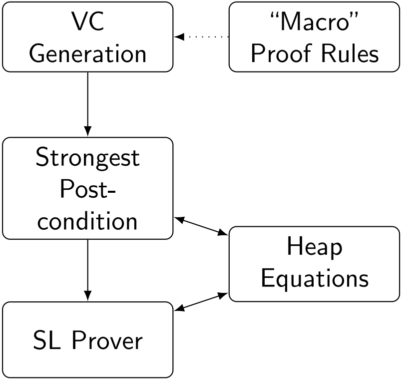
∀ p ls, p ≠ 0 →
∃ v p' ls', p ↦ v ∗ p+4 ↦ p' ∗
llist p' ls' ⇒ llist p ls

Outline

Strongest Post-condition and Data Abstraction

Read from Array

```
{ ![ Array Rv 1024 0s] st }
Sp := Rv[100] ;
Rv[100] := Sp
```



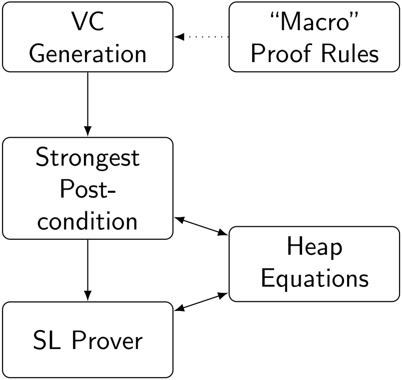
Strongest Post-condition and Data Abstraction

Read from Array

{ ![`Array` Rv 1024 0s] st }
Sp := Rv[100] ;
Rv[100] := Sp

Strongest Post-condition

{ ![`Array` Rv 1024 0s] } st
$\wedge$ `evalInstrs` st [Sp := Rv[100]; ...] st'
$\rightarrow$ ...



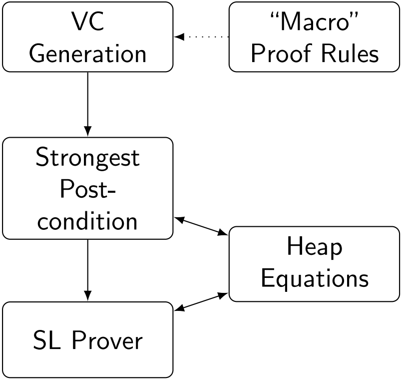
Strongest Post-condition and Data Abstraction

Read from Array

```
{ ![ Array Rv 1024 0s] st }
Sp := Rv[100] ;
Rv[100] := Sp
```

Strongest Post-condition

```
{ ![ Array Rv 1024 0s] } st
∧ evalInstrs st [Sp := Rv[100]; ...]  st'
→ ...
```



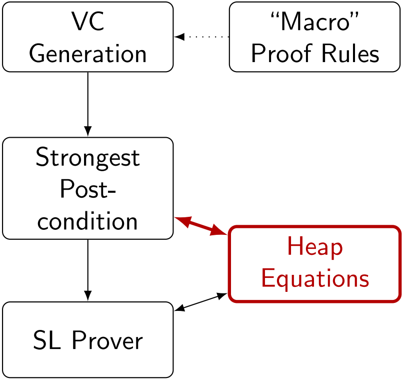
Strongest Post-condition and Data Abstraction

Read from Array

{ ![Array Rv 1024 0s] st }
Sp := Rv[100] ;
Rv[100] := Sp

Strongest Post-condition

{ ![Array Rv 1024 0s] } st
∧ evalInstrs st [Sp := Rv[100]; ...] st'
→ ...



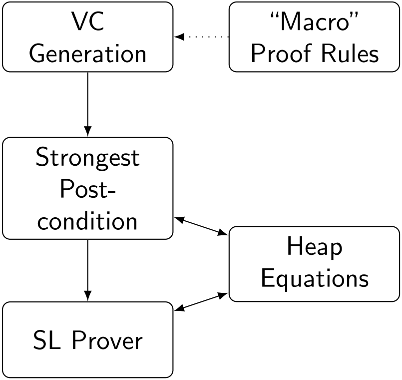
Strongest Post-condition and Data Abstraction

Read from Array

```
{ ![ Array Rv 1024 0s] st }
Sp := Rv[100] ;
Rv[100] := Sp
```

Strongest Post-condition

```
{ ![ Array Rv 1024 0s] } st
∧ evalInstrs st [Sp := Rv[100]; ...]  st'
→ ...
```



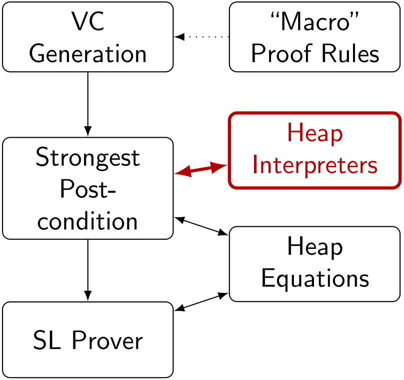
Strongest Post-condition and Data Abstraction

Read from Array

{ ![`Array Rv 1024 0s`] `st` }
`Sp` := `Rv`[100] ;
`Rv`[100] := `Sp`

Strongest Post-condition

{ ![`Array Rv 1024 0s`] } `st`
∧ `evalInstrs st` [`Sp` := `Rv`[100]; ...] `st'`
→ ...



Strongest Post-condition and Data Abstraction

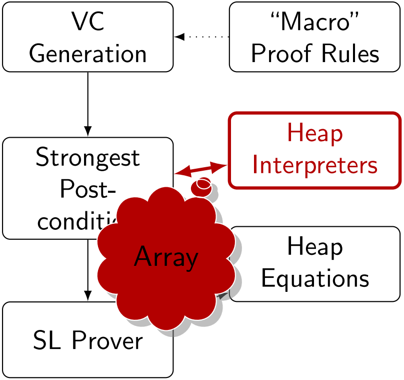
Read from Array

{ ![Array Rv 1024 0s] st }
Sp := Rv[100] ;
Rv[100] := Sp

Strongest Post-condition

{ ![Array Rv 1024 0s] } st
∧ evalInstrs st [Sp := Rv[100]; ...] st'
→ ...

Stuck!



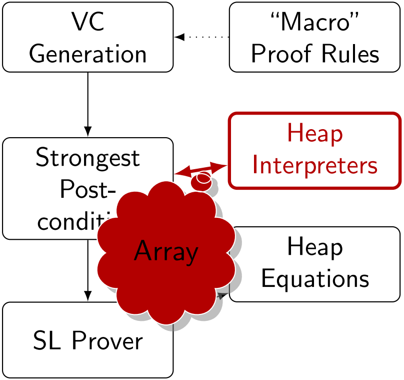
Strongest Post-condition and Data Abstraction

Read from Array

{ ![Array Rv 1024 0s] st }
Sp := Rv[100] ;
Rv[100] := Sp

Strongest Post-condition

{ ![Array Rv 1024 0s]
  ∧ Sp = get 0s 100 } st
∧ evalInstrs st [Rv[100] := Sp] st'
→ ...



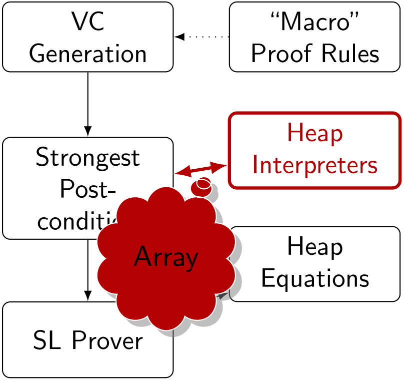
Strongest Post-condition and Data Abstraction

Read from Array

$\{$![Array Rv 1024 0s] st $\}$
Sp := Rv[100] ;
Rv[100] := Sp

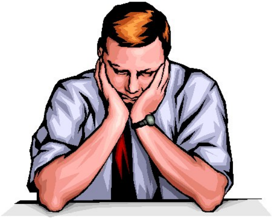
Strongest Post-condition

$\{$![Array Rv 1024 (update 100 Sp 0s)]
  $\wedge$ Sp = get 0s 100 $\}$ st'
$\rightarrow$ ...



Outline

Verification with Computational Proofs



- Constructing proofs can take a long time...
  - Verification needs to be fast.

"Traditional" Proofs: `Even 2048`

Definition of Even

$$\frac{}{\text{Even } 0}\ \text{Even\_0}$$

$$\frac{\text{Even } n}{\text{Even } n+2}\ \text{Even\_SS}$$

An Easy Proof Script

```
Theorem Even_2048 : Even 2048.
  repeat constructor.
Qed.
```

7s

7s

A **Huge** Proof

$$\cfrac{\cfrac{\cfrac{}{\text{Even } 0}\ \text{Even\_0}}{\ldots(1022 \text{ applications})}\ \text{Even\_SS}}{\cfrac{\text{Even } 2046}{\text{Even } 2048}\ \text{Even\_SS}}\ \text{Even\_SS}$$

Proofs by Computational Reflection

Definition of Even

$$\frac{}{\text{Even } 0}\ \text{Even\_0}$$

$$\frac{\text{Even } n}{\text{Even } n+2}\ \text{Even\_SS}$$

A Prover

```
Fixpoint is_even n : bool :=
  match n with
    | 0 ⇒ true
    | 1 ⇒ false
    | S (S n) ⇒ is_even n
  end.
Theorem is_even_Even : ∀ n,
  is_even n = true → Even n.
Qed.
```

A Good Proof

$$\dfrac{\dfrac{\dfrac{}{\text{true} = \text{true}}\ \text{Reflexivity}}{\text{is\_even } 2048 = \text{true}}\ [\text{computation}]}{\text{Even } 2048}\ \text{is\_even\_Even}$$

Proofs by Computational Reflection

Definition of Even

$$\frac{}{\text{Even } 0}\ \text{Even\_0}$$

$$\frac{\text{Even } n}{\text{Even } n+2}\ \text{Even\_SS}$$

Total Proof: 0s

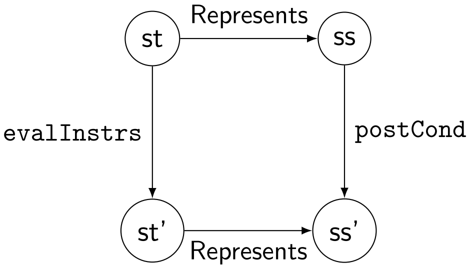
A Prover

```
Fixpoint is_even n : bool :=
  match n with
    | 0 ⇒ true
    | 1 ⇒ false
    | S (S n) ⇒ is_even n
  end.
Theorem is_even_Even : ∀ n,
  is_even n = true → Even n.
Qed.
```

A Good Proof

$$\dfrac{\dfrac{\dfrac{}{\text{true} = \text{true}}\ \text{Reflexivity}}{\text{is\_even } 2048 = \text{true}}\ [\text{computation}]}{\text{Even } 2048}\ \text{is\_even\_Even}$$

Applying Computational Reflection



Reflective Theorem

```
Theorem symEval_sound : ∀ instrs ss ss' st,
  Represents ss st →
  evalInstrs st instrs st' →
  postCond ss instrs = Some ss' →
  Represents ss' st'.
```

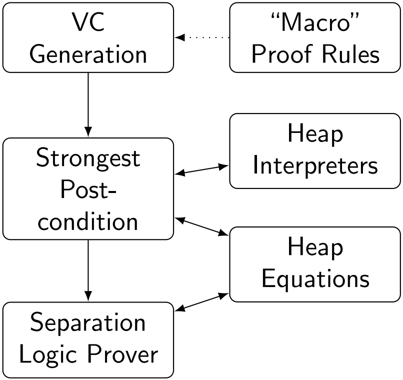
Outline

Future Directions

- Extend to other core languages
  - x86, LLVM
- Concurrency
- Low-level interaction
  - Virtual memory
  - Devices
- Optimization

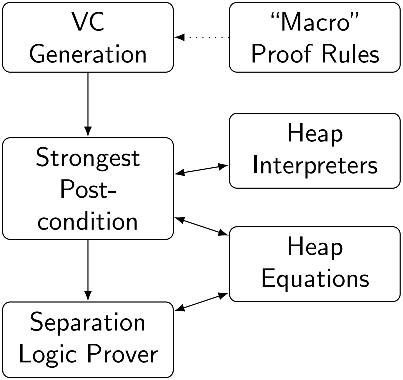
Overview: Bedrock

- Define higher-level syntax on low-level syntax
- VC generation and symbolic evaluation
- Avoid baking in features
  - Extensible heap interpreters
  - Extensible heap equations
- Separation logic prover



Overview: Bedrock

- Define higher-level syntax on low-level syntax
- VC generation and symbolic evaluation
- Avoid baking in features
  - Extensible heap interpreters
  - Extensible heap equations
- Separation logic prover



Questions?