# Bedrock: A Framework for Verifying Low-level Programs

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

Harvard University SEAS

NEPLS'12 – June 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))
```

# 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.

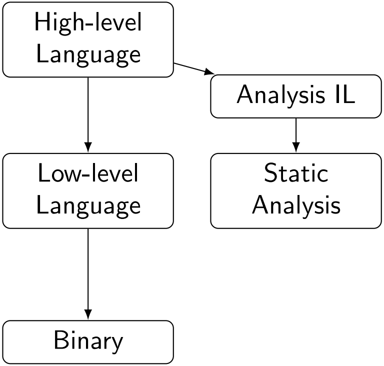
# Project Goals

A programming language for systems code that supports

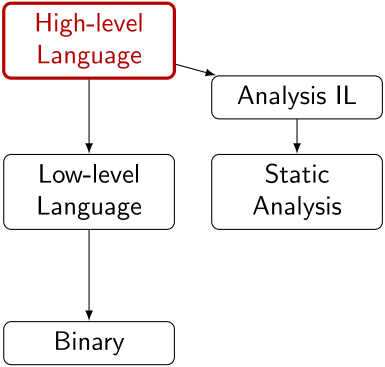
- verification down to the machine code ...
- of low-level code ...
- with a small trusted computing base ...
- while supporting higher-order specifications ...
- and is both extensible ...
- and reasonable to program with.

**Bedrock2**

# Typical Program Verification

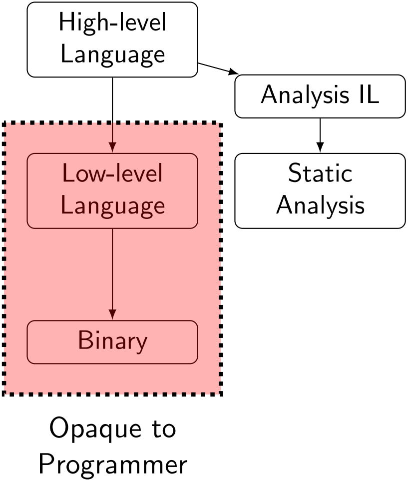


# Typical Program Verification



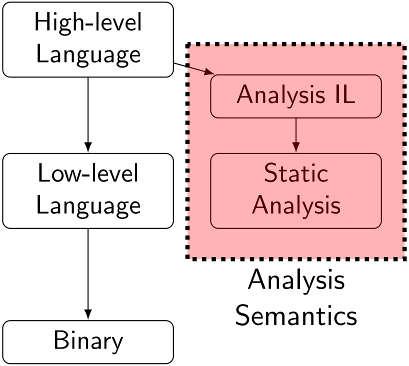

- Focus is on the high-level language!

# Typical Program Verification



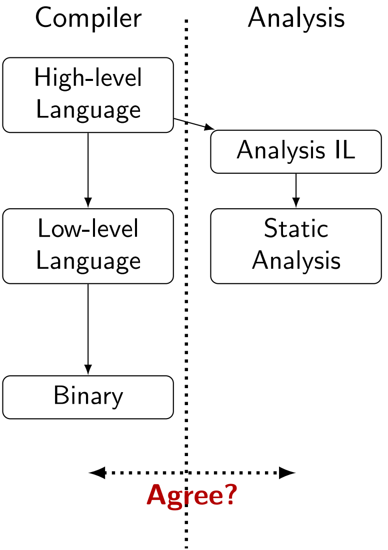

- Focus is on the high-level language!
- Compiler focuses on converting to a binary.

# Typical Program Verification




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

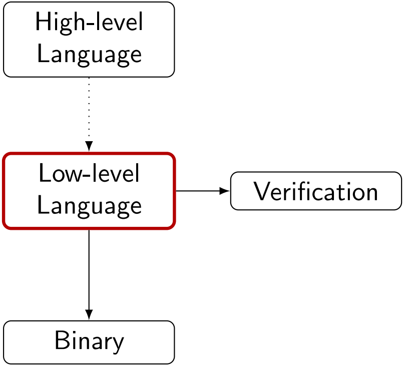
# Typical Program Verification




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

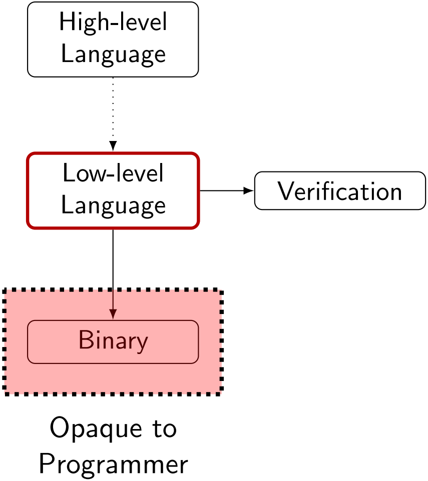
# Verification in Bedrock

- Focus is on a low-level language.



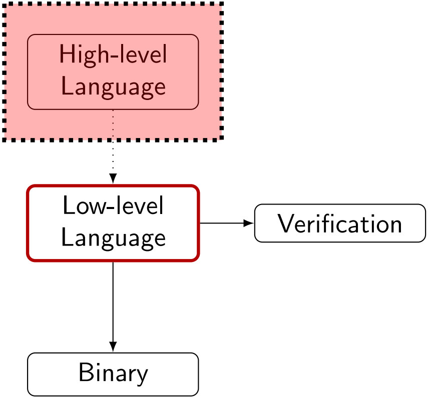
# Verification in Bedrock

- 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

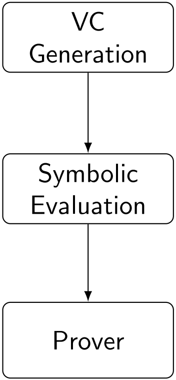
## A Simple Program

```
{Rp @@ (st' ~> st'.Rv = 0)}
Rv := 0 ;
goto Rp
```

# Verifying a Simple Program

## A Simple Program

```
{Rp @@ (st' ~> st'.Rv = 0)}
Rv := 0 ;
goto Rp
```



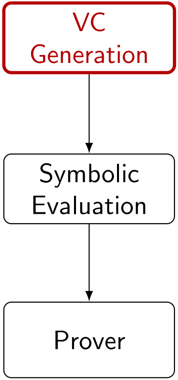
# Verifying a Simple Program

## A Simple Program

```
{Rp @@ (st' ~> st'.Rv = 0)}
Rv := 0 ;
goto Rp
```

## Symbolic Evaluation

```
{Rp@@(st' ~> st'.Rv = 0)} st
∧ evalInstrs st [ Rv := 0 ] st'
→ { Rv = 0 } st'
```



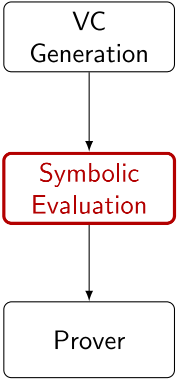
# Verifying a Simple Program

## A Simple Program

```
{Rp @@ (st' ~> st'.Rv = 0)}
Rv := 0 ;
goto Rp
```

## Symbolic Evaluation

```
{Rp@@(st' ~> st'.Rv = 0)} st
∧ evalInstrs st [ Rv := 0 ] st'
→ { Rv = 0 } st'
```



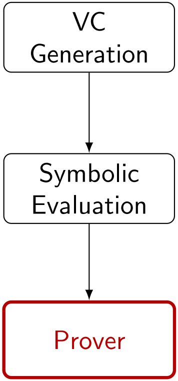
# Verifying a Simple Program

## A Simple Program

```
{Rp @@ (st' ~> st'.Rv = 0)}
Rv := 0 ;
goto Rp
```

## Proof Obligation

```
{Rv = 0 ∧
 Rp @@ (st' ~> st'.Rv = 0)} st'
→ { Rv = 0 } st'
```

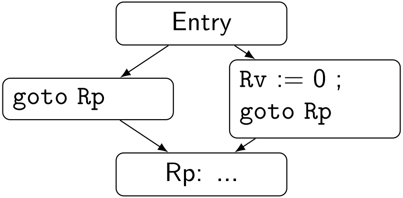


# Outline

# Bedrock2: Extensible Control

## Conditional Code

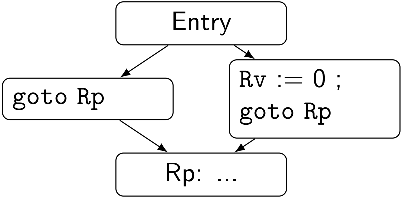
$$
\begin{array}{llll}
\{ & \texttt{Entry} : & ([], & \texttt{br Rv Eq 0 Tr Fa}) \\
; & \texttt{Tr} \quad : & ([], & \texttt{goto Rp}) \\
; & \texttt{Fa} \quad : & ([\,\texttt{Rv} := 0], & \texttt{goto Rp}) \; \}
\end{array}
$$



# Bedrock2: Extensible Control

## Conditional Code

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

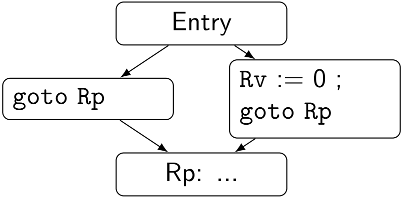



This looks like if!

# Bedrock2: 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

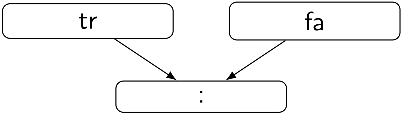
```
Definition If (cmp : Cmp) (l : Rhs) (r : Rhs) (tr : Code) (fa : Code) :
    Code := ...
```

# Control Abstraction

- Build syntax combinators in the meta-language

## If Combinator

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

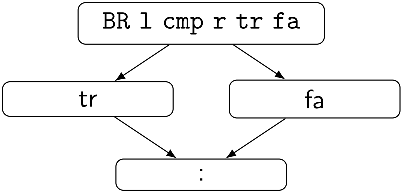


# Control Abstraction

- Build syntax combinators in the meta-language

## If Combinator

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

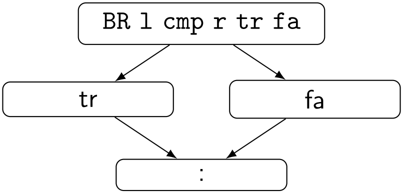


# Control Abstraction

- 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.

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

# Verified Control Abstraction

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

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

## 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.

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

## If Combinator Sketch

Precondition

```
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.

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

## 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.

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

## 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 the blocks

# Verified Control Abstraction

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

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

## 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 post condition

# Verified Control Abstraction

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

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

## If Combinator Sketch

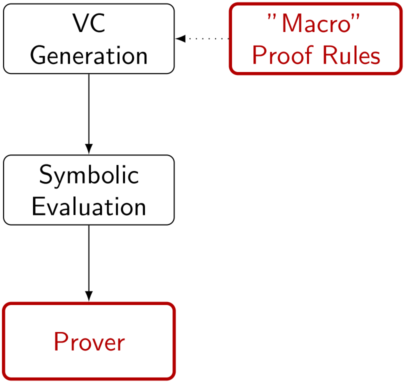
```
let If cmp l r tr fa := fun P ⇒
  let tr := tr (P ∧ cmp l r = true) in
  let fa := fa (P ∧ cmp l r = false) in
  { Ent : L
  ; Blocks : { L : (BR cmp l r tr.Ent fa.Ent) } ∪ tr.Blocks ∪ fa.Blocks
  ; Post : tr.Post ∨ fa.Post
  ; Safe : safeTest l cmp r ∧ tr.Safe ∧ fa.Safe
  }
```

Combine safety conditions

# Verification with Extended Control

## Always-0 with Conditionals

```
{Rp @@ (st' ~> st'.Rv = 0)}
If (Rv = 0) {
  skip
} Else {
  Rv := 0
} ;
goto Rp
```



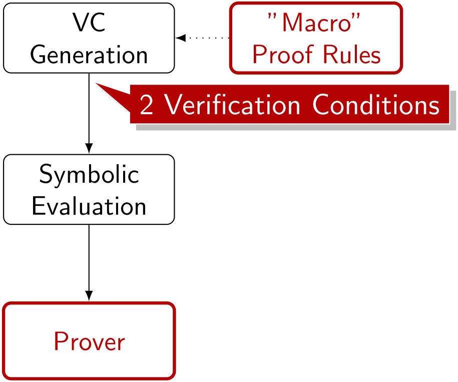
# Verification with Extended Control

## Always-0 with Conditionals

```
{Rp @@ (st' ~> st'.Rv = 0)}
If (Rv = 0) {
  skip
} Else {
  Rv := 0
} ;
goto Rp
```

## New VC

```
{Rp@@(st' ~> st'.Rv = 0)} st
∧ evalCond st (Rv = 0)
→ { Rv = 0 } st
```



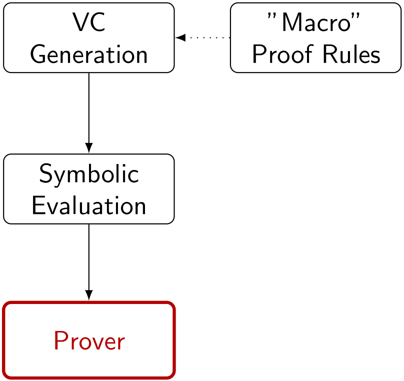
# Verification with Extended Control

## Always-0 with Conditionals

```
{Rp @@ (st' ~> st'.Rv = 0)}
If (Rv = 0) {
  skip
} Else {
  Rv := 0
} ;
goto Rp
```

## Proof Obligation

```
{Rp@@(st' ~> st'.Rv = 0)
 ∧ Rv = 0 } st
→ { Rv = 0 } st
```



# Outline

# Verification with Memory

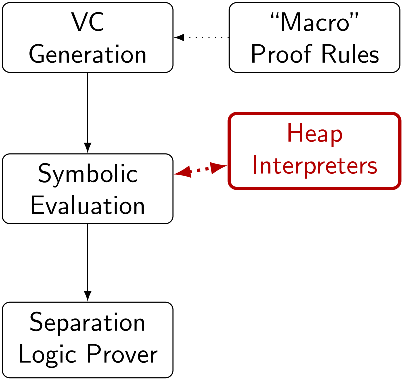
## Always-0 with Memory

```
{ ∃ v, ![Rv ↦ v] ∧
  Rp @@ (st' ~> ![Rv ↦ 0] st') }
$[Rv] := 0 ;
goto Rp
```

# Verification with Memory

### Always-0 with Memory

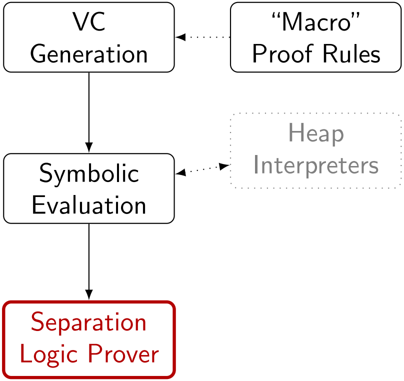
```
{ ∃ v, ![Rv ↦ v] ∧
  Rp @@ (st' ~> ![Rv ↦ 0] st') }
$[Rv] := 0 ;
goto Rp
```



# Verification with Memory

## Always-0 with Memory

```
{ ∃ v, ![Rv ↦ v] ∧
  Rp @@ (st' ~> ![Rv ↦ 0] st') }
$[Rv] := 0 ;
goto Rp
```



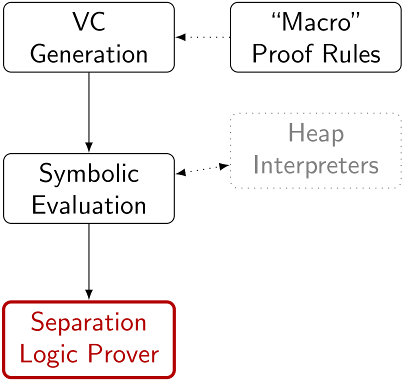
# Verification with Memory

## Always-0 with Memory

```
{ ∃ v, ![Rv ↦ v] ∧
  Rp @@ (st' ~> ![Rv ↦ 0] st') }
$[Rv] := 0 ;
goto Rp
```

## Proof Obligation

```
{ Rp @@ (st' ~> ![Rv ↦ 0] st') st
  ![ Rv ↦ 0] } st
→ { ![Rv ↦ v] } st
```



# Verification with Memory

## Always-0 with Memory

```
{ ∃ v, ![Rv ↦ v] ∧
  Rp @@ (st' ~> ![Rv ↦ 0] st') }
$[Rv] := 0 ;
goto Rp
```

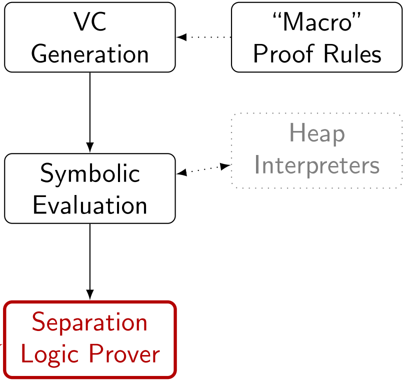
## Proof Obligation

```
{ Rp @@ (st' ~> ![Rv ↦ 0] st') st
  ![ Rv ↦ 0] } st
→ { ![Rv ↦ v] } st
```

---

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

---



# 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

$$p_2 \mapsto v_2 \qquad \Rightarrow \qquad p_2 \mapsto v_2$$

# 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

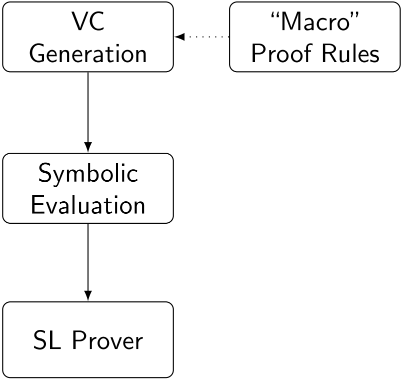
## Linked List Head

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

# Reasoning about Abstract Data Types: Lists

## Linked List Head

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



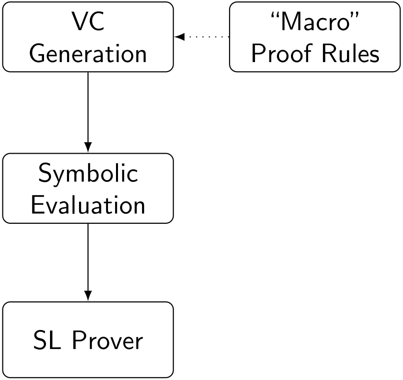
# Reasoning about Abstract Data Types: Lists

## Linked List Head

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

## Symbolic Evaluation

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



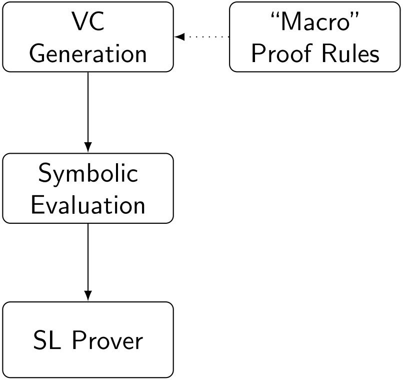
# Reasoning about Abstract Data Types: Lists

## Linked List Head

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

## Symbolic Evaluation

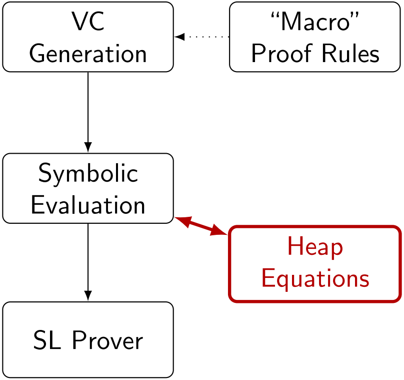
```
{∃ ls, ![llist Rv ls] ∧ Rv ≠ 0} st
∧ evalInstrs st [Rv := $[Rv]] st'
→ {∃ ls, ![llist st.Rv ls]
    ∧ st'.Rv = hd ls} st'
```

# Reasoning about Abstract Data Types: Lists

## Linked List Head

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




## Symbolic Evaluation

```
{∃ ls, ![llist Rv ls] ∧ Rv ≠ 0} 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' ∗
  ls = v :: ls'
```

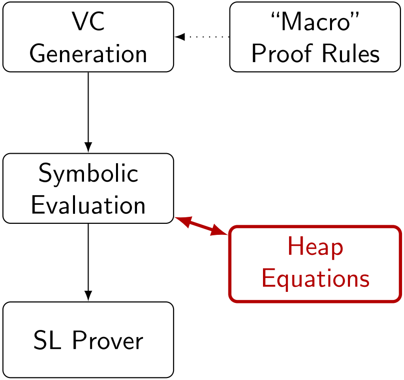
# Reasoning about Abstract Data Types: Lists

## Linked List Head

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

## 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'
```

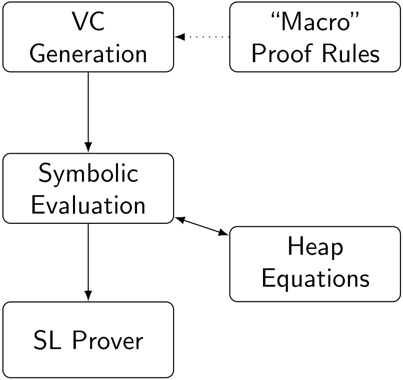
# Reasoning about Abstract Data Types: Lists

## Linked List Head

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

## 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

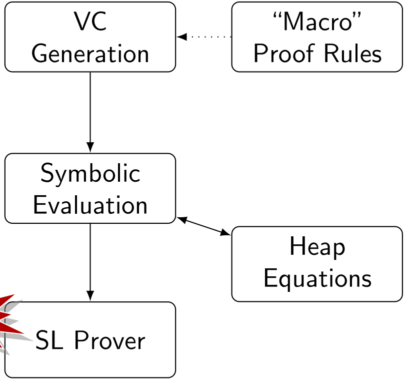
## Linked List Head

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

Stuck!

## Proof Obligation

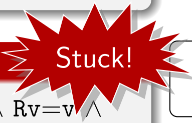
```
{∃ ls p' v ls', ls=v::ls' ∧ Rv=v ∧
![ Rv ↦ v ∗ Rv+4 ↦ p' ∗
  llist p' ls']} st'
→ {∃ ls p' v ls', Rv = hd ls ∧
![ llist st.Rv ls]} st'
```



# Reasoning about Abstract Data Types: Lists

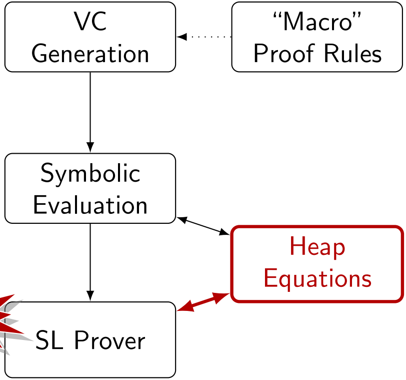
## Linked List Head

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




## 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'
```




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

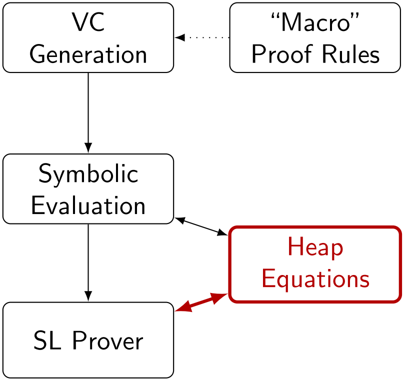
# Reasoning about Abstract Data Types: Lists

## Linked List Head

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

## 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' * s']} st'
```



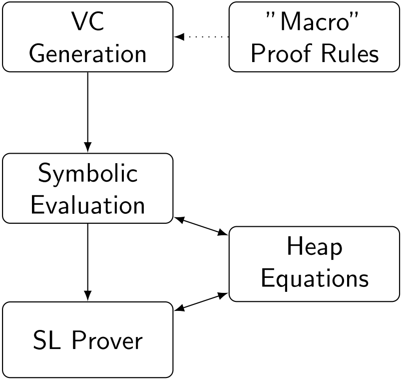

```
∀ p ls, p ≠ 0 →
 ∃ v p' ls', p ↦ v * p+4 ↦ p' *
  llist p' ls' ⇒ llist p ls
```

# Outline

# Symbolic Evaluation and Data Abstraction

## Read from Array

{ ![ Array Rv 1024 0s] st ∧
  Rp @@ ...}
Sp := Rv[100] ;
Rv[100] := Sp ;
goto Rp



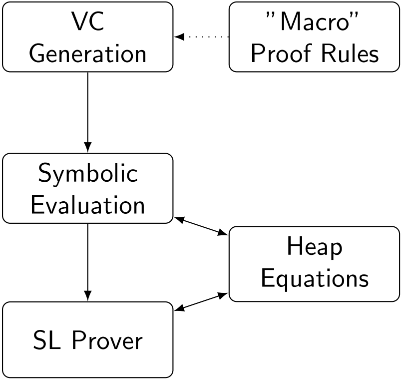
# Symbolic Evaluation and Data Abstraction

## Read from Array

```
{ ![ Array Rv 1024 0s] st ∧
  Rp @@ ...}
Sp := Rv[100] ;
Rv[100] := Sp ;
goto Rp
```

## Symbolic Evaluation

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



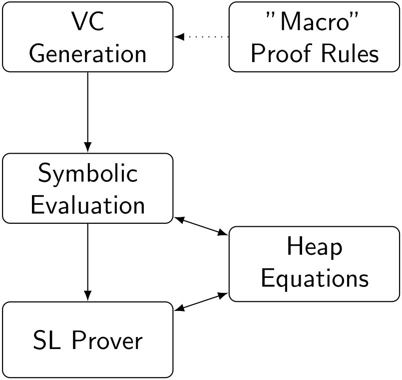
# Symbolic Evaluation and Data Abstraction

## Read from Array

{ ![ Array Rv 1024 0s] st ∧
  Rp @@ ...}
Sp := Rv[100] ;
Rv[100] := Sp ;
goto Rp

## Symbolic Evaluation

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

# Symbolic Evaluation and Data Abstraction

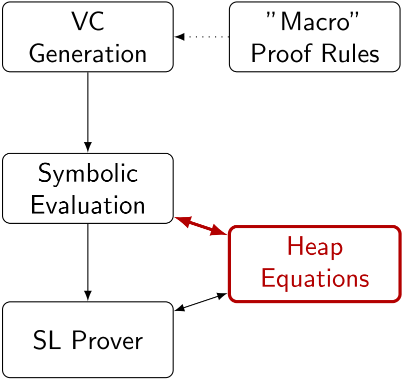
## Read from Array

```
{ ![ Array Rv 1024 0s] st ∧
  Rp @@ ...}
Sp := Rv[100] ;
Rv[100] := Sp ;
goto Rp
```

## Symbolic Evaluation

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

Bad!



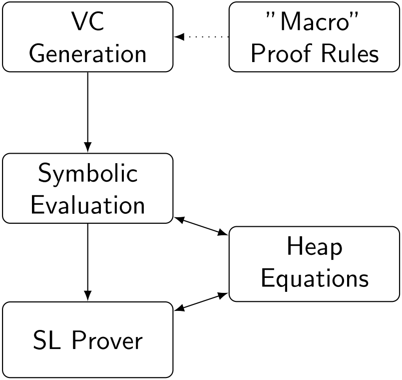
# Symbolic Evaluation and Data Abstraction

## Read from Array

```
{ ![ Array Rv 1024 0s] st ∧
  Rp @@ ...}
Sp := Rv[100] ;
Rv[100] := Sp ;
goto Rp
```

## Symbolic Evaluation

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

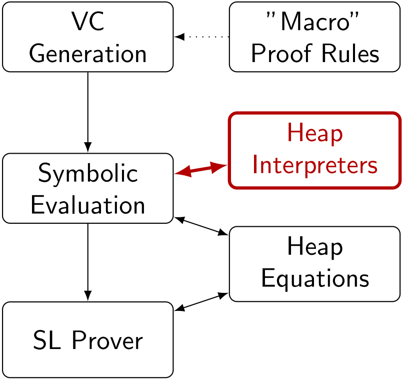
# Symbolic Evaluation and Data Abstraction

## Read from Array

```
{ ![ Array Rv 1024 0s] st ∧
  Rp @@ ...}
Sp := Rv[100] ;
Rv[100] := Sp ;
goto Rp
```

## Symbolic Evaluation

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

# Symbolic Evaluation and Data Abstraction

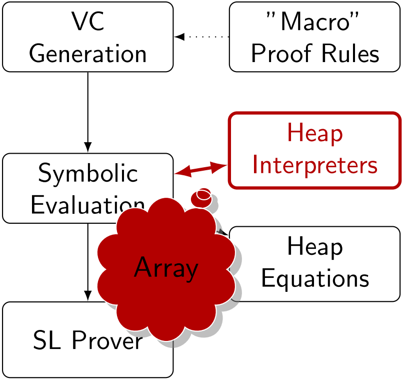
## Read from Array

{ ![ Array Rv 1024 0s] st ∧
  Rp @@ ...}
Sp := Rv[100] ;
Rv[100] := Sp ;
goto Rp

## Symbolic Evaluation

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

Stuck!



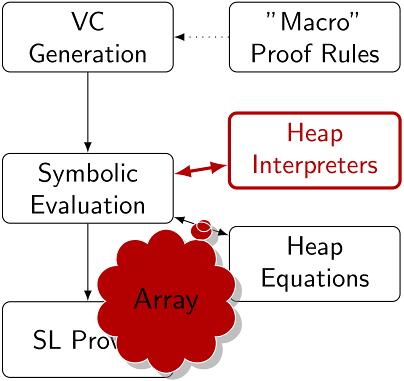
# Symbolic Evaluation and Data Abstraction

## Read from Array

```
{ ![ Array Rv 1024 0s] st ∧
  Rp @@ ...}
Sp := Rv[100] ;
Rv[100] := Sp ;
goto Rp
```

## Symbolic Evaluation

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



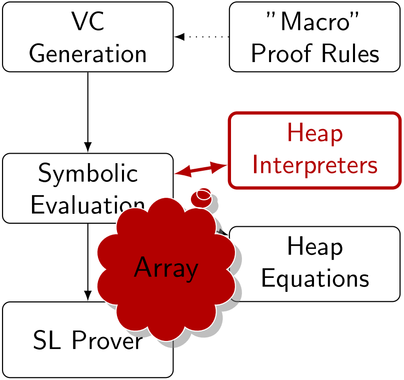
# Symbolic Evaluation and Data Abstraction

## Read from Array

{ ![ Array Rv 1024 0s] st ∧
  Rp @@ ...}
Sp := Rv[100] ;
Rv[100] := Sp ;
goto Rp

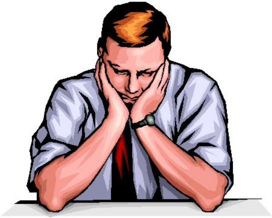
## Symbolic Evaluation

{ ![ Array Rv 1024 (update 100 Sp 0s)]
  ∧ Sp = get 0s 100 } st'
→ ...



# 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

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

# 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

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

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

## 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

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

Total Proof: 0s

# Applying Computational Reflection

$$\begin{tikzcd}[column sep=huge, row sep=huge] st \arrow[r, "Represents"] \arrow[d, "evalInstrs"'] & ss \arrow[d, "sym\_eval"] \\ st' \arrow[r, "Represents"'] & ss' \end{tikzcd}$$

## Reflective Theorem

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

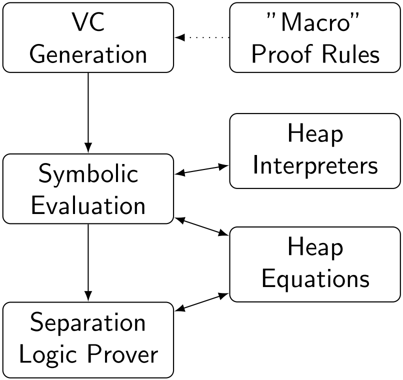
# Outline

# Future Directions

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

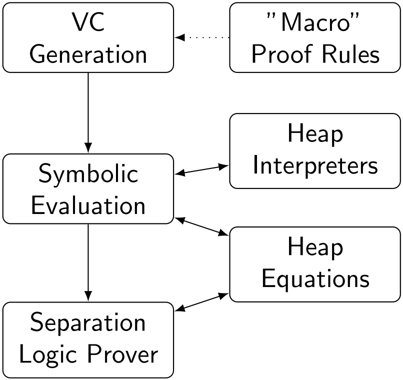
# Overview: Bedrock2

- 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: Bedrock2

- 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?**

# A Simple Core Language

$$
\begin{array}{llll}
(\mathit{Registers}) & r & ::= & \mathbf{Rv} \mid \mathbf{Rp} \mid \mathbf{Sp} \\
(\mathit{Expressions}) & e & ::= & r + r \mid r - r \mid r * r \mid \$[r + c] \\
(\mathit{Instructions}) & i & ::= & r := e \mid \$[r + c] := e \\
(\mathit{Branches}) & t & ::= & \mathbf{goto}\ c \mid \mathbf{goto}\ r \mid \mathbf{br}\ cmp\ r\ r\ l\ l \\
(\mathit{BasicBlocks}) & b & ::= & i*; t
\end{array}
$$