Peter Müller

BUILDING DEDUCTIVE PROGRAM VERIFIERS

ETH zürich

VTSA 2023

GHOST-ALLOC

$$\frac{\overline{\mathcal{V}}(a)}{\mathsf{True} \Rrightarrow_{\mathcal{E}} \exists \gamma.\, \boxed{a}^{\gamma}}$$

GHOST

$\boxed{a \cdot b}^{\gamma}$

HOARE-VS

$$\frac{P \Rrightarrow_{\mathcal{E}} P' \qquad \{P'\}\, e\, \{v.\, Q'\}_{\mathcal{E}} \qquad \forall v. \ldots}{\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}}}$$

INV-ALLOC

$$P \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

HOA…

$\{\triangleright P \ldots$

HOARE-CTX

$$\frac{\{P * Q\}\, e\, \{v.\, R\}_{\mathcal{E}} \qquad \mathsf{persistent}(Q \ldots}{Q \twoheadrightarrow \{P\}\, e\, \{v.\, R\}_{\mathcal{E}}}$$

PERSISTENT-SEP

$$\frac{\mathsf{persistent}(P) \qquad \mathsf{pe}\ldots}{\mathsf{persistent}(P * \ldots}$$

$$\frac{\mathcal{Q} \colon \mathit{Values} \to \mathit{Assertions}}{\{\mathsf{emp}\}\ \mathsf{alloc}()\ \{\ell.\, \mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{Acq}(\ell, \mathcal{Q})\}}$$

$$\frac{\mathsf{normalizable}(\mathcal{Q}(v))}{\{\mathsf{Rel}(\ell, \mathcal{Q}) * \mathcal{Q}(v)\}\ [\ell]_{\mathbf{rel}} := v\ \{\mathsf{Init}(\ell)\}}$$

$$\frac{\forall v.\, \mathsf{precise}(\mathcal{Q}(v)) \land \mathsf{normalizable}(\mathcal{Q}(v))}{\{\mathsf{Acq}(\ell, \mathcal{Q}) * \mathsf{Init}(\ell)\}\ [\ell]_{\mathbf{acq}}\ \{v.\, \mathsf{Acq}(\ell, \mathcal{Q}[v := \mathsf{emp}]) * \mathcal{Q}(v)\}}$$

$$\frac{}{\{\mathsf{Rel}(\ell, \mathcal{Q}) * \triangle \mathcal{Q}(v)\}\ [\ell]_{\mathbf{rlx}} := v\ \{\mathsf{Init}(\ell)\}}$$

$$\frac{\forall v.\, \mathsf{precise}(\mathcal{Q}(v)) \land \mathsf{normalizable}(\mathcal{Q}(v))}{\{\mathsf{Acq}(\ell, \mathcal{Q}) * \mathsf{Init}(\ell)\}\ [\ell]_{\mathbf{rlx}}\ \{v.\, \mathsf{Acq}(\ell, \mathcal{Q}[v := \mathsf{emp}]) * \triangledown \mathcal{Q}(v)\}}$$

$$\frac{\mathcal{Q} \colon \mathit{Values} \to \mathit{Assertions}}{\{\mathsf{emp}\}\ \mathsf{alloc}()\ \{\ell.\, \mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{RMWAcq}(\ell, \mathcal{Q})\}}$$

Let $\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) := \mathsf{Rel}(\ell, \mathcal{Q}_{\mathtt{rel}}) * \mathsf{RMWAcq}(\ell, \mathcal{Q}_{\mathtt{acq}}) * \mathsf{Init}(\ell)$ **in**

$$\frac{\begin{array}{c} \mathcal{Q}_{\mathtt{acq}}(v) \Rightarrow \exists z.\, \mathcal{A}(z) * \mathcal{T}(z) \\ \forall z.\, (P * \mathcal{T}(z) \Rightarrow \mathcal{Q}_{\mathtt{rel}}(v') \land \varphi(z)) \\ \forall z.\, \mathsf{pure}(\varphi(z)) \\ \mathsf{normalizable}(P) \\ \{\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * P\}\ [\ell]_\sigma\ \{a.\, a \neq v \to R\} \\ \sigma \in \{\mathbf{acq}, \mathbf{rlx}\} \end{array}}{\begin{array}{c} \{\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * P\} \\ \mathsf{CAS}_{\mathbf{acq\_rel}, \sigma}(\ell, v, v') \\ \left\{\begin{array}{l} a.\ (a = v \land \exists z.\, \mathcal{A}(z) \land \varphi(z)) \\ \lor (a \neq v \land R) \end{array}\right\} \end{array}}$$

$$\frac{\begin{array}{c} \mathcal{Q}_{\mathtt{acq}}(v) \Rightarrow \exists z.\, \mathcal{A}(z) * \mathcal{T}(z) \\ \forall z.\, (P * \mathcal{T}(z) \Rightarrow \mathcal{Q}_{\mathtt{rel}}(v') \land \varphi(z)) \\ \forall z.\, \mathsf{pure}(\varphi(z)) \\ \mathsf{normalizable}(P) \\ \{\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * P\}\ [\ell]_\sigma\ \{a.\, a \neq v \to R\} \\ \sigma \in \{\mathbf{acq}, \mathbf{rlx}\} \end{array}}{\begin{array}{c} \{\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * P\} \\ \mathsf{CAS}_{\mathbf{rel}, \sigma}(\ell, v, v') \\ \left\{\begin{array}{l} a.\ (a = v \land \exists z.\, \triangledown \mathcal{A}(z) \land \varphi(z)) \\ \lor (a \neq v \land R) \end{array}\right\} \end{array}}$$

$$\frac{\begin{array}{c} \mathcal{Q}_{\mathtt{acq}}(v) \Rightarrow \exists z.\, \mathcal{A}(z) * \mathcal{T}(z) \\ \forall z.\, (P * \mathcal{T}(z) \Rightarrow \mathcal{Q}_{\mathtt{rel}}(v') \land \varphi(z)) \\ \forall z.\, \mathsf{pure}(\varphi(z)) \\ \{\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * \triangle P\}\ [\ell]_\sigma\ \{a.\, a \neq v \to R\} \\ \sigma \in \{\mathbf{acq}, \mathbf{rlx}\} \end{array}}{\begin{array}{c} \{\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * \triangle P\} \\ \mathsf{CAS}_{\mathbf{acq}, \sigma}(\ell, v, v') \\ \left\{\begin{array}{l} a.\ (a = v \land \exists z.\, \mathcal{A}(z) \land \varphi(z)) \\ \lor (a \neq v \land R) \end{array}\right\} \end{array}}$$

$$\frac{\begin{array}{c} \mathcal{Q}_{\mathtt{acq}}(v) \Rightarrow \exists z.\, \mathcal{A}(z) * \mathcal{T}(z) \\ \forall z.\, (P * \mathcal{T}(z) \Rightarrow \mathcal{Q}_{\mathtt{rel}}(v') \land \varphi(z)) \\ \forall z.\, \mathsf{pure}(\varphi(z)) \\ \{\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * \triangle P\}\ [\ell]_\sigma\ \{a.\, a \neq v \to R\} \\ \sigma \in \{\mathbf{acq}, \mathbf{rlx}\} \end{array}}{\begin{array}{c} \{\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * \triangle P\} \\ \mathsf{CAS}_{\mathbf{rlx}, \sigma}(\ell, v, v') \\ \left\{\begin{array}{l} a.\ (a = v \land \exists z.\, \triangledown \mathcal{A}(z) \land \varphi(z)) \\ \lor (a \neq v \land R) \end{array}\right\} \end{array}}$$

$$\frac{\begin{array}{c} \mathcal{Q}(v) * P \Rightarrow \mathsf{false} \\ \{\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * P\}\ [\ell]_\sigma\ \{a.\, a \neq v \to R\} \\ \tau \in \{\mathbf{rlx}, \mathbf{rel}, \mathbf{acq}, \mathbf{acq\_rel}\} \qquad \sigma \in \{\mathbf{acq}, \mathbf{rlx}\} \end{array}}{\{\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * P\}\ \mathsf{CAS}_{\tau, \sigma}(\ell, v, v')\ \{a.\, a \neq v \land R\}}$$

Frame rule

$$\frac{\ldots X.\, \langle p_p \mid p(x) \rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\, \langle q_p(x, y) \mid q(x, y) \rangle}{\ldots_p \mid r(x) * p(x) \rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\, \langle r' * q_p(x, y) \mid r(x) * q(x, y) \rangle}$$

Substitution rule

$$\frac{\ldots \mid p(x) \rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\, \langle q_p(x, y) \mid q(x, y) \rangle \qquad f : X' \to X}{\ldots p_p \mid p(f(x')) \rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\, \langle q_p(f(x'), y) \mid q(f(x'), y) \rangle}$$

Atomicity weakening rule

$$\frac{\ldots_p \mid p' * p(x) \rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\, \langle q_p(x, y) \mid q'(x, y) * q(x, y) \rangle}{\ldots_p * p' \mid p(x) \rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\, \langle q_p(x, y) * q'(x, y) \mid q(x, y) \rangle}$$

Open region rule

$$\frac{\ldots (\mathbf{t}_a^\lambda(x)) * p(x) \rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\, \langle q_p(x, y) \mid I(\mathbf{t}_a^\lambda(x)) * q(x, y) \rangle}{\ldots p_p \mid \mathbf{t}_a^\lambda(x) * p(x) \rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\, \langle q_p(x, y) \mid \mathbf{t}_a^\lambda(x) * q(x, y) \rangle}$$

Use atomic rule

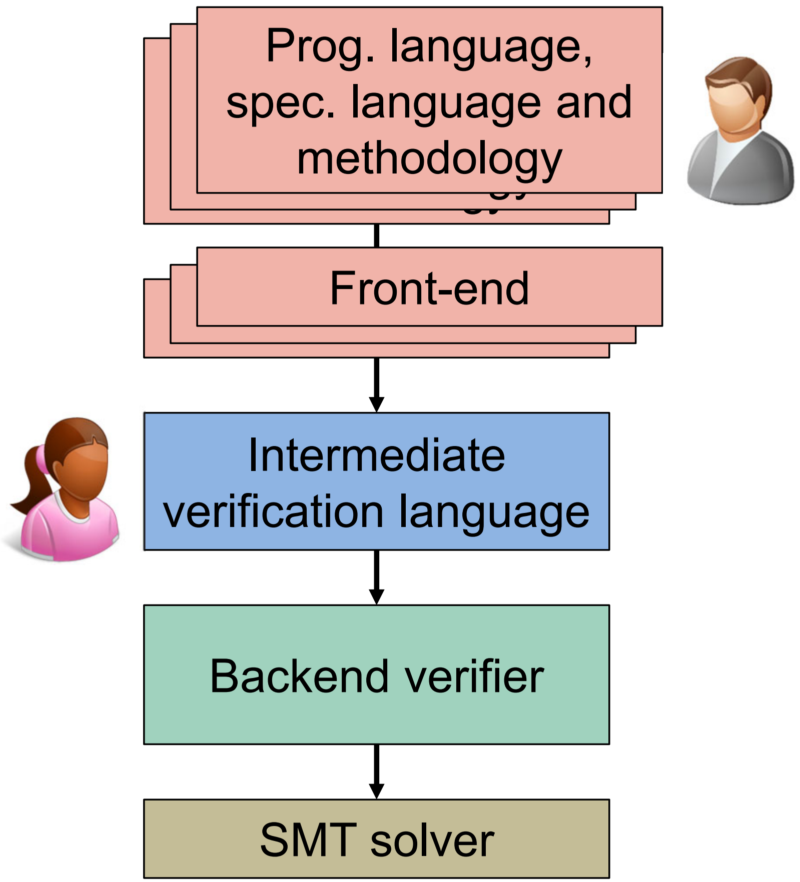
$$\frac{\ldots \notin \mathcal{A} \quad \forall x \in X.\, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \qquad \ldots)) * p(x) * [\mathrm{G}]_a \rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\, \langle q_p(x, y) \mid I(\mathbf{t}_a^\lambda(f(x))) * q(x, y) \rangle}{\ldots_a^\lambda(x) * p(x) * [\mathrm{G}]_a \rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\, \langle q_p(x, y) \mid \mathbf{t}_a^\lambda(f(x)) * q(x, y) \rangle}$$

Update region rule

$$\frac{\ldots (x)) * p(x) \rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Y.\, \left\langle q_p(x, y)\ \middle|\ \begin{array}{l} I(\mathbf{t}_a^\lambda(Q(x))) * q_1(x, y) \\ \lor I(\mathbf{t}_a^\lambda(x)) * q_2(x, y) \end{array} \right\rangle}{\begin{array}{c} \forall\!\!\forall x \in X.\, \langle p_p \mid \mathbf{t}_a^\lambda(x) * p(x) * a \Mapsto \blacklozenge \rangle \\ \mathbb{C} \\ \exists\!\!\exists y \in Y.\, \left\langle q_p(x, y)\ \middle|\ \begin{array}{l} \exists z \in Q(x).\, \mathbf{t}_a^\lambda(z) * q_1(x, y) * a \Mapsto (x, z) \\ \lor \mathbf{t}_a^\lambda(x) * q_2(x, y) * a \Mapsto \blacklozenge \end{array} \right\rangle \end{array}}$$

Make atomic rule

$$\frac{\begin{array}{c} \{(x, y) \mid x \in X, y \in Q(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \\ \ldots Q(x), \mathcal{A} \vdash \begin{array}{c} \{p_p * \exists x \in X.\, \mathbf{t}_a^\lambda(x) * a \Mapsto \blacklozenge\} \\ \mathbb{C} \\ \{\exists x \in X, y \in Q(x).\, q_p(x, y) * a \Mapsto (x, y)\} \end{array} \end{array}}{\ldots \mid \mathbf{t}_a^\lambda(x) * [\mathrm{G}]_a \rangle\ \mathbb{C}\ \ \exists\!\!\exists y \in Q(x).\, \langle q_p(x, y) \mid \mathbf{t}_a^\lambda(y) * [\mathrm{G}]_a \rangle}$$



Prog. language, spec. language and methodology
Front-end
Intermediate verification language
Backend verifier
SMT solver

Outline

- Automated program verification
- Reasoning about the heap
- Abstraction
- Concurrency
- Conclusion

Guarded Commands

Types

$T ::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{Rational} \mid \texttt{Real}$

Expressions

$E ::= c\,(onstant) \mid v\,(ariable) \mid E + E \mid E * E \mid E - E$
$\mid E < E \mid E \land E \mid E \lor E \mid \neg E \mid \ldots$ (+ syntactic sugar)

Assertions

$\mathbf{A} ::= E \mid \forall x : T :: \mathbf{A} \mid \exists x : T :: \mathbf{A} \mid \mathbf{A} \Rightarrow \mathbf{A} \mid \ldots$

Program statements

$S ::= v := E$	assignment
$\mid S; S$	sequential composition
$\mid$ **if** $(*)\ \{S\}$ **else** $\{S\}$	nondeterministic choice
$\mid$ **assert A**	assertion
$\mid$ **assume A**	assumption
$\mid$ **havoc** v	nondeterministic assignment

All types are mathematical (unbounded)

We assume that expressions and programs are well-typed

Hoare logic

$$\frac{}{\{\, \mathbf{A}[E\,/\,x] \,\}\ x := E\ \{\, \mathbf{A} \,\}}$$

$$\frac{}{\{\, \mathbf{A} \wedge \mathbf{B} \,\}\ \textbf{assert}\ \mathbf{A}\ \{\, \mathbf{B} \,\}}$$

$$\frac{\{\, \mathbf{A} \,\}\ S\ \{\, \mathbf{C} \,\} \qquad \{\, \mathbf{C} \,\}\ S'\ \{\, \mathbf{B} \,\}}{\{\, \mathbf{A} \,\}\ S;\ S'\ \{\, \mathbf{B} \,\}}$$

$$\frac{}{\{\, \mathbf{A} \Rightarrow \mathbf{B} \,\}\ \textbf{assume}\ \mathbf{A}\ \{\, \mathbf{B} \,\}}$$

$$\frac{\{\, \mathbf{A} \,\}\ S\ \{\, \mathbf{C} \,\} \qquad \{\, \mathbf{B} \,\}\ S'\ \{\, \mathbf{C} \,\}}{\{\, \mathbf{A} \wedge \mathbf{B} \,\}\ \textbf{if}\ (\,*\,)\ \{\, S \,\}\ \textbf{else}\ \{\, S' \,\}\ \{\, \mathbf{C} \,\}}$$

$$\frac{}{\{\, \forall x : \mathbf{A} \,\}\ \textbf{havoc}\ x\ \{\, \mathbf{A} \,\}}$$

$$\frac{\mathbf{A} \Rightarrow \mathbf{A}' \qquad \{\, \mathbf{A}' \,\}\ S\ \{\, \mathbf{B}' \,\} \qquad \mathbf{B}' \Rightarrow \mathbf{B}}{\{\, \mathbf{A} \,\}\ S\ \{\, \mathbf{B} \,\}}$$

Our Hoare triples have a partial correctness meaning

Challenges for automating proof search

- Writing Hoare-style proofs requires creativity

$$\frac{\{\,\mathbf{A}\,\}\ \mathsf{S}\ \{\,\mathbf{C}\,\} \qquad \{\,\mathbf{C}\,\}\ \mathsf{S'}\ \{\,\mathbf{B}\,\}}{\{\,\mathbf{A}\,\}\ \mathsf{S};\ \mathsf{S'}\ \{\,\mathbf{B}\,\}}$$

How do we find intermediate assertions?

$$\frac{\mathbf{A} \Rightarrow \mathbf{A}' \qquad \{\,\mathbf{A}'\,\}\ \mathsf{S}\ \{\,\mathbf{B}'\,\} \qquad \mathbf{B}' \Rightarrow \mathbf{B}}{\{\,\mathbf{A}\,\}\ \mathsf{S}\ \{\,\mathbf{B}\,\}}$$

Where and how do we
weaken and strengthen assertions?

- How do we decide whether an implication holds?
  - We delegate the task to an SMT solver

Weakest preconditions

Statement S	$wp[\![S]\!](\mathbf{B})$
`x := E`	$\mathbf{B}[E / x]$
`S; S'`	$wp[\![S]\!](wp[\![S']\!](\mathbf{B}))$
if $(*)$ $\{ S \}$ **else** $\{ S' \}$	$wp[\![S]\!](\mathbf{B}) \wedge wp[\![S']\!](\mathbf{B})$
assert $\mathbf{A}$	$\mathbf{A} \wedge \mathbf{B}$
assume $\mathbf{A}$	$\mathbf{A} \Rightarrow \mathbf{B}$
havoc x	$\forall x : \mathbf{B}$

To automate the proof of a triple

$$\{ \mathbf{A} \} \; S \; \{ \mathbf{B} \}$$

we decide

$$\mathbf{A} \Rightarrow wp[\![S]\!](\mathbf{B})$$

Encoding into guarded commands: conditionals

- Other statements can be encoded into guarded commands

- Conditional statements

$$\texttt{if } (E)\ \{ S \}\ \texttt{else}\ \{ S' \}$$

$$\frac{\{ \mathbf{A} \land E \}\ S\ \{ \mathbf{B} \} \qquad \{ \mathbf{A} \land \neg E \}\ S'\ \{ \mathbf{B} \}}{\{ \mathbf{A} \}\ \texttt{if } (E)\ \{ S \}\ \texttt{else}\ \{ S' \}\ \{ \mathbf{B} \}}$$

can be encoded using nondeterministic choice and assume

$$[\![\ \texttt{if } (E)\ \{ S \}\ \texttt{else}\ \{ S' \}\ ]\!] \ = \ \texttt{if } (*)\ \{ \texttt{assume } E;\ S \}\ \texttt{else}\ \{ \texttt{assume } \neg E;\ S' \}$$

Encoding into guarded commands: loops

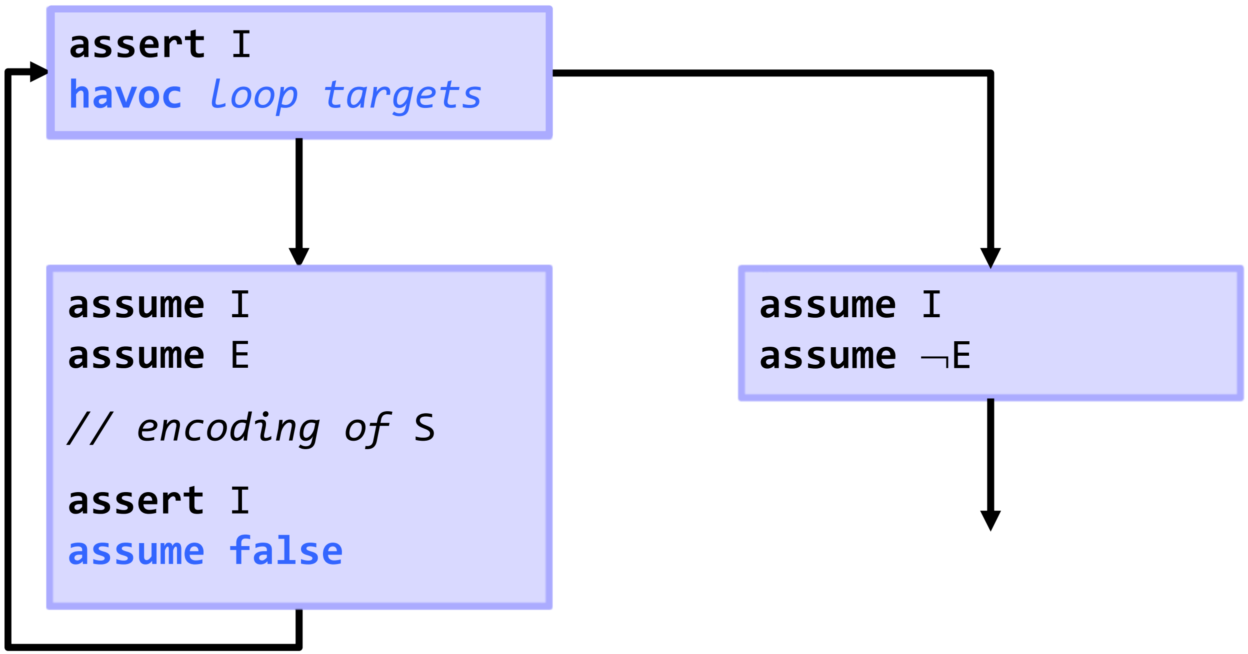
- While statements are verified using loop invariants

```
while (E) {S}
```

Hoare logic

$$\frac{\{\, I \wedge E \,\}\ S\ \{\, I \,\}}{\{\, I \,\}\ \mathtt{while}\ (E)\ S\ \{\, I \wedge \neg E \,\}}$$

- Encoding

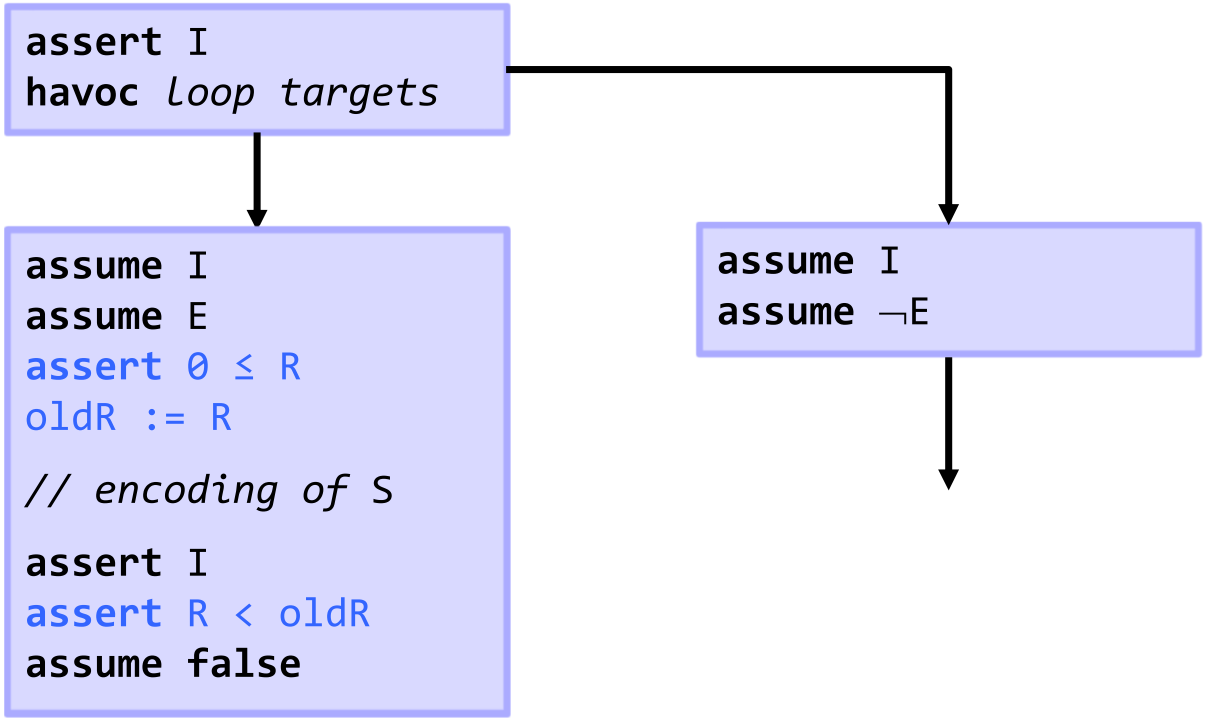


Encoding into guarded commands: loop termination

- Termination can be proved with termination measures

```
while (E)
  invariant I
  decreases R
{S}
```

- Encoding



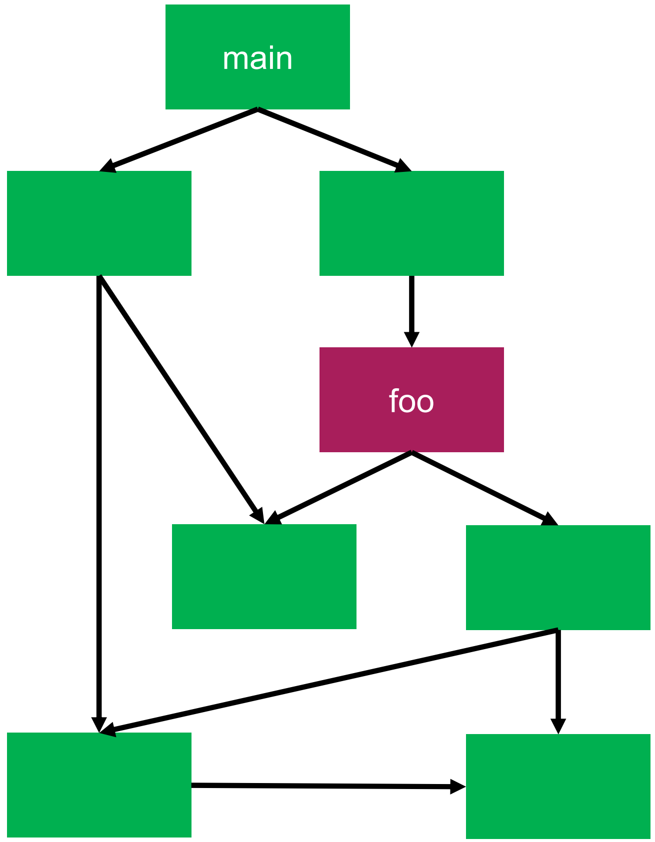
Encoding of calls

```
method indexOf(s: Seq[Int], e: Int) returns (res: Int)
{ … }
```

```
method client() {
  var i: Int
  i := indexOf(Seq(1, 3, 2), 3)
  assert i == 1
}
```

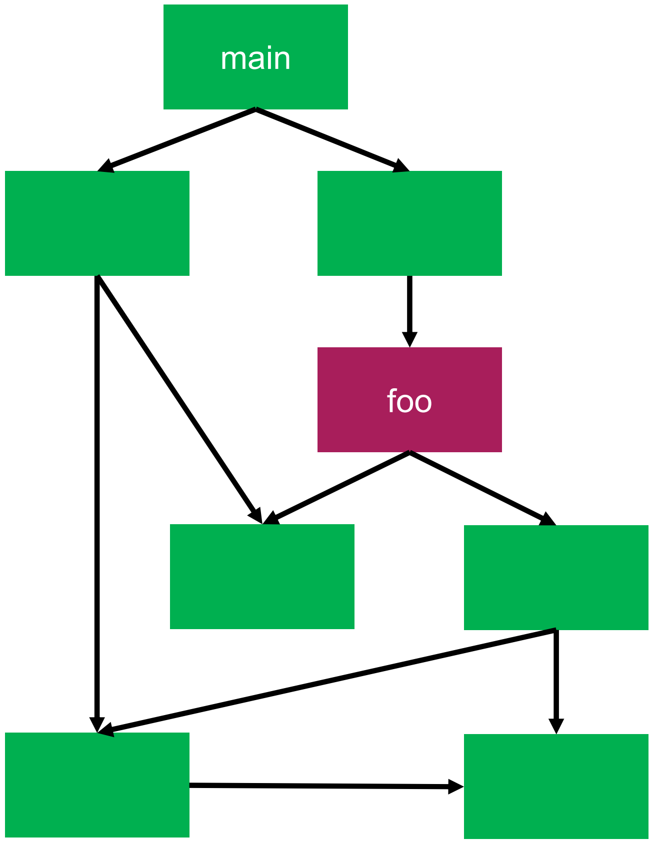
Modular Verification

- Verify each procedure separately
  - Scalability
- Do not use the implementation of callees
  - Software evolution
  - Dynamic method binding, foreign functions
- Do not use the implementation of callers and other procedures
  - Correctness guarantees for libraries
  - Software evolution



Contracts

- Contracts specify the intended behavior of parts of the program
- For the verification of a procedure, use the contracts of the rest of the program, not the implementation
- Verify calls in terms of procedure pre- and postconditions



Encoding into guarded commands: procedures

- Procedure declarations

```
method P(x̄: T̄)
    [returns (ȳ: T̄)]
    [requires A]
    [ensures B]
{S}
```

```
assume A
// encoding of S
assert B
```

To handle recursion, proof may assume that all procedures satisfy their specifications

For terminating programs, the correctness argument is not cyclic

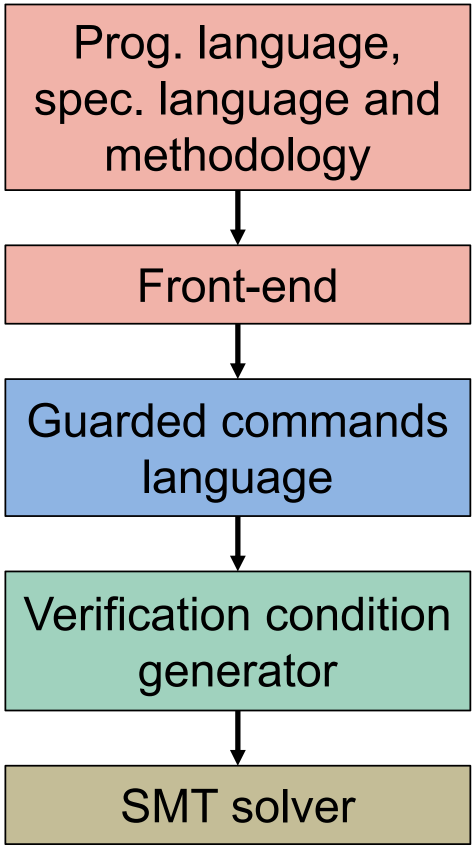
- Procedure calls

```
z̄ := P(Ē)
```

where x is not free in E

```
assert A[Ē / x̄]
havoc z̄
assume B[Ē / x̄][z̄ / ȳ]
```

Summary



Loops, procedures
Safety, functional correctness, termination

Translates programs and specifications to IL

Can express programs and specifications

Extracts proof obligations automatically using wp

Resolves proof obligations

- viper.ethz.ch
- Try online: http://viper.ethz.ch/tutorial
- Install as VS Code extension

Outline

- Automated program verification
- Reasoning about the heap
- Abstraction
- Concurrency
- Conclusion

Heap model: an object-based language

```
field val: Int

method foo() returns (res: Int)
{
  var cell: Ref
  cell := new(val)
  cell.val := 5
  res := cell.val
}
```

- A heap is a set of objects
- No classes: each object has all fields declared in the entire program
  - Type rules of a source language can be encoded
  - Memory consumption is not a concern since programs are not executed
- Objects are accessed via references
  - Field read and update operations
  - No information hiding
- No explicit de-allocation (garbage collector)
  - Conceptually, objects could remain allocated

Extended programming language

Declarations

$D ::= \; \ldots \mid \textbf{field}\; f : \; T$

Fields are declared globally

Types

$T ::= \; \ldots \mid \textbf{Ref}$

Only one type of references

Expressions

$E ::= \; \ldots \mid \textbf{null} \mid E.f$

Pre-defined null-reference

Statements

$S ::= \; \ldots$ as before

$\mid v := \textbf{new}(\overline{f}) \mid v := \textbf{new}(*)$ allocation

$\mid x.f := E$ field update

Allocation with given list of fields or all fields

Field access: naïve proof rules

- Naïve approach: treat field accesses like variable assignment

Field read

$$\frac{}{\{\, E \neq \mathbf{null} \wedge \mathbf{A}[E.f \,/\, v] \,\}\ \ v := E.f\ \ \{\, \mathbf{A} \,\}}$$

Field update

$$\frac{}{\{\, x \neq \mathbf{null} \wedge \mathbf{A}[E \,/\, x.f] \,\}\ \ x.f := E\ \ \{\, \mathbf{A} \,\}}$$

- Additional precondition prevents null-dereferencing

The naïve proof rule for field update is unsound.

Naïve rule for field update ignores aliasing

Field read

$$\frac{}{\{\, E \neq \mathbf{null} \wedge \mathbf{A}[E.f \,/\, v] \,\}\ \ v := E.f\ \ \{\, \mathbf{A} \,\}}$$

Field update

$$\frac{}{\{\, x \neq \mathbf{null} \wedge \mathbf{A}[E \,/\, x.f] \,\}\ \ x.f := E\ \ \{\, \mathbf{A} \,\}}$$

```
field val: Int

method foo(p: Ref)
{
  var q: Ref
  assume p != null && p.val == 5
  { p ≠ null ∧ p ≠ null ∧ p.val = 5 }
  q := p
  { p ≠ null ∧ q ≠ null ∧ q.val = 5 }
  p.val := 7
  { q ≠ null ∧ q.val = 5 }
  assert q.val == 5
}
```

The frame problem

```
field f: Int
field g: Int
```

```
method set(p: Ref, v: Int)
  requires p != null
  ensures p.f == v
{
  p.f := v
}
```

```
x.f := 0
x.g := 0
set(x, 5)
assert x.g == 0
```

- Bad idea: inspect body of callee to determine which field locations are modified
  - Not modular
  - Does not work for abstract methods
- Bad idea: assume conservatively that all field locations may be modified
  - Callee needs a specification for all field locations, even those it does not change
  - Not modular: procedure specifications need to change when a new field is declared

Summary of challenges

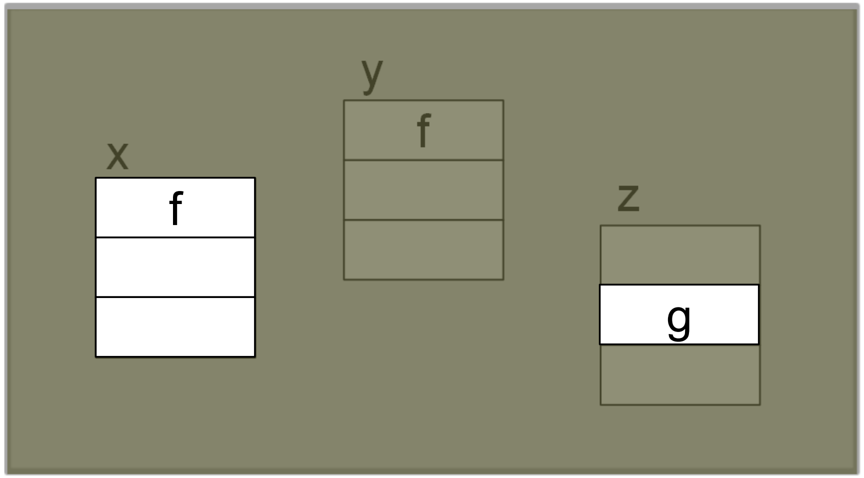
Heap data structures pose three major challenges for sequential verification

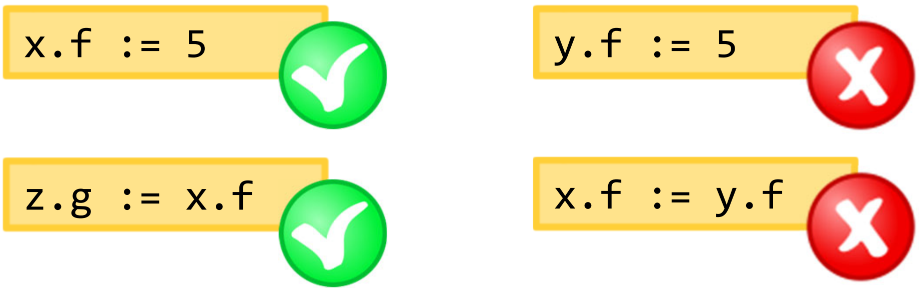
- Reasoning about aliasing
- Framing, especially for dynamic data structures
- Writing specifications that preserve information hiding

And additional challenges for concurrent programs, e.g., data races

Access permissions

- Associate each heap location with a permission
- Permissions are held by method executions or loop iterations
- Read or write access to a memory location requires permission
- Permissions are created when the heap location is allocated
- Permissions can be transferred, but not duplicated or forged





Permission assertions

- Permissions are denoted in assertions by access predicates
  - Access predicates are not permitted under negations, disjunctions, and on the left of implications

Assertions

A ::= ... | **acc**(E.f)

- Assertions may contain both permissions and value constraints

```
acc(p.f) && p.f > 0
```

- Many assertions that occur in a program must be self-framing, that is, include all permissions to evaluate the heap accesses in the assertion

```
requires p.f > 0
```

- An assertion that does not contain access predicates is called pure

Separating conjunction

- To handle aliasing, we introduce a new connective: separating conjunction

- **A** * **B** holds in a state if:
  - both **A** and **B** hold, and
  - the sum of the permissions in **A** and **B** are held in that state
  - **A** * **B** and **A** $\wedge$ **B** are equivalent if **A** and **B** are pure

- Holding permission to locations p.f and q.f implies that p and q do not alias

$$\textbf{acc}(p.f) * \textbf{acc}(q.f) \Rightarrow p \neq q$$

- Viper's && is separating conjunction

- For the call `swap(x, x)`, the precondition is equivalent to false

```
method swap(a: Ref, b: Ref)
  requires acc(a.f) && acc(b.f)
```

Field access: proof rules with permissions

Field read

$$\frac{}{\{\ \mathbf{acc}(x.f) * \mathbf{A}[x.f\,/\,v]\ \}\ \ v := x.f\ \ \{\ \mathbf{acc}(x.f) * \mathbf{A}\ \}}$$

Field update

$$\frac{}{\{\ \mathbf{acc}(x.f)\ \}\ \ x.f := E\ \ \{\ \mathbf{acc}(x.f) * x.f = E\ \}}$$

where E does not contain field accesses

- Each field access requires (and preserves) the corresponding permission
- Permission to a location implies that the receiver is non-null

Framing

Frame rule

$$\frac{\{\ \mathbf{A}\ \}\ \ \mathtt{S}\ \ \{\ \mathbf{B}\ \}}{\{\ \mathbf{A} * \mathbf{C}\ \}\ \ \mathtt{S}\ \ \{\ \mathbf{B} * \mathbf{C}\ \}}$$

where S does not assign to a local variable that is free in **C**

- The frame **C** must be self-framing
  - If heap locations constrained by **C** are disjoint from those modified by S, **C** is preserved
  - Otherwise, the precondition is equivalent to false (the triple holds trivially)

- Example

$$\frac{\overline{\{\ \mathbf{acc}(\mathtt{x.f})\ \}\ \ \mathtt{x.f:=5}\ \ \{\ \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5\ \}}}{\{\ \mathbf{acc}(\mathtt{x.f}) * \mathbf{acc}(\mathtt{y.f}) * \mathtt{y.f} = 7\ \}\ \ \mathtt{x.f:=5}\ \ \{\ \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5 * \mathbf{acc}(\mathtt{y.f}) * \mathtt{y.f} = 7\ \}}$$

Framing for method calls

```
method set(p: Ref, v: Int)
  requires acc(p.f)
  ensures  acc(p.f) && p.f == v
{
  p.f := v
}
```

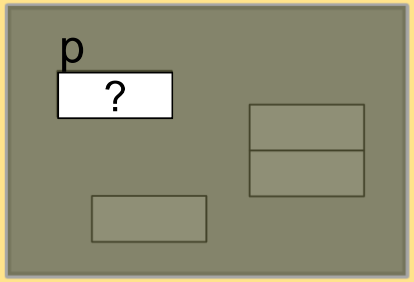
```
// assume we have acc(x.f) && acc(y.f)
assume y.f == 7
set(x, 5)
assert x.f == 5 && y.f == 7
```

$$\dfrac{\dfrac{\{\ \mathbf{acc}(\mathtt{p.f})\ \}\ \ \mathbf{method}\ \mathtt{set(p,\ v)}\ \ \{\ \mathbf{acc}(\mathtt{p.f}) * \mathtt{p.f} = \mathtt{v}\ \}}{\{\ \mathbf{acc}(\mathtt{x.f})\ \}\ \ \mathtt{set(x,\ 5)}\ \ \{\ \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5\ \}}}{\{\ \mathbf{acc}(\mathtt{x.f}) * \mathbf{acc}(\mathtt{y.f}) * \mathtt{y.f} = 7\ \}\ \ \mathtt{set(x,\ 5)}\ \ \{\ \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5 * \mathbf{acc}(\mathtt{y.f}) * \mathtt{y.f} = 7\ \}}$$

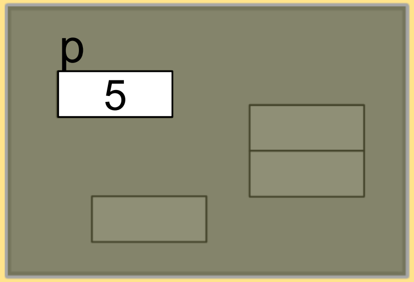
- A method may modify only heap locations to which it has permission

Permission transfer

```
method set(p: Ref, v: Int)
  requires acc(p.f)
  ensures  acc(p.f) && p.f == v
{
```

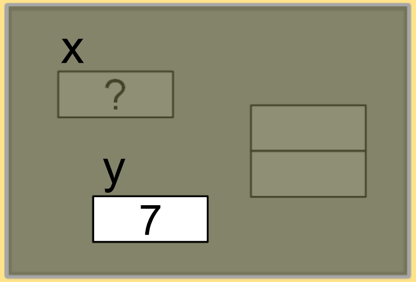



```
  p.f := v
```




```
}
```

```
// assume we have acc(x.f) && acc(y.f)
assume x.f == 2 && y.f == 7
```

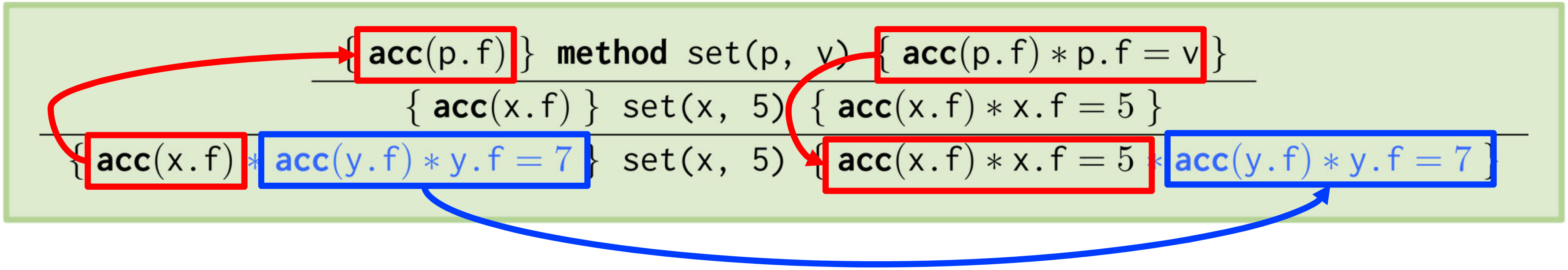



```
set(x, 5)
```

Framing!

```
assert x.f == 5 && y.f == 7
```

Permission transfer for method calls



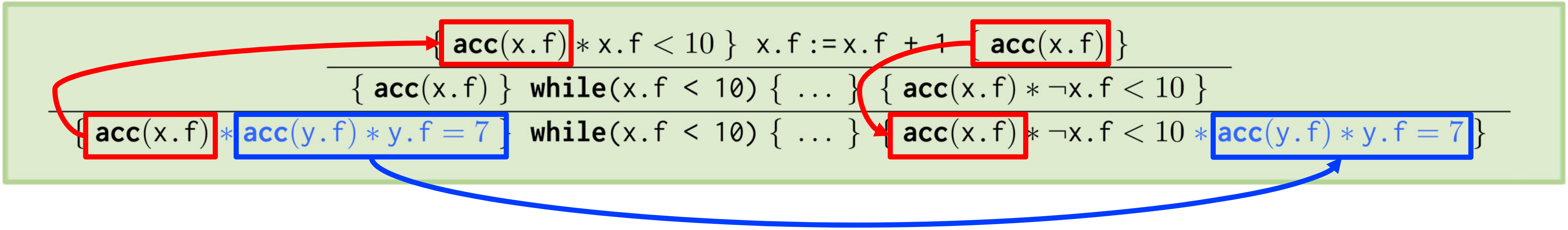
- Permissions are held by method executions or loop iterations
- Calling a method transfers permissions from the caller to the callee (according to the method precondition)
- Returning from a method transfers permissions from the callee to the caller (according to the method postcondition)
- Residual permissions are framed around the call

Framing for loops

```
// assume we have acc(x.f) && acc(y.f)
x.f := 0
y.f := 7
while (x.f < 10)
  invariant acc(x.f)
{
  x.f := x.f + 1
}
assert y.f == 7
```

$$\dfrac{\dfrac{\{\, \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} < 10 \,\}\ \ \mathtt{x.f := x.f + 1}\ \ \{\, \mathbf{acc}(\mathtt{x.f}) \,\}}{\{\, \mathbf{acc}(\mathtt{x.f}) \,\}\ \ \mathbf{while}(\mathtt{x.f < 10})\ \{\ \ldots\ \}\ \ \{\, \mathbf{acc}(\mathtt{x.f}) * \neg \mathtt{x.f} < 10 \,\}}}{\{\, \mathbf{acc}(\mathtt{x.f}) * \mathbf{acc}(\mathtt{y.f}) * \mathtt{y.f} = 7 \,\}\ \ \mathbf{while}(\mathtt{x.f < 10})\ \{\ \ldots\ \}\ \ \{\, \mathbf{acc}(\mathtt{x.f}) * \neg \mathtt{x.f} < 10 * \mathbf{acc}(\mathtt{y.f}) * \mathtt{y.f} = 7 \,\}}$$

Permission transfer for loops

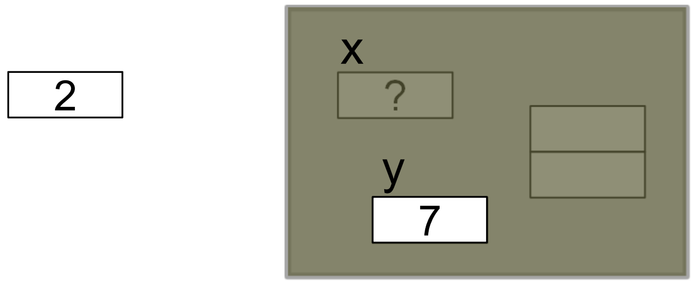


- Permissions are held by method executions or loop iterations
- Entering a loop transfers permissions from the enclosing context to the loop (according to the loop invariant)
- Leaving a loop transfers permissions from the loop to the enclosing context (according to the loop invariant)
- Residual permissions are framed around the loop

Permission transfer: inhale and exhale operations

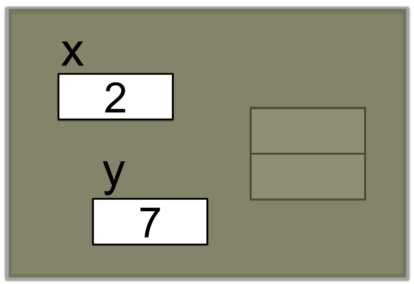
- **`inhale A`** means:
  - obtain all permissions required by assertion **A**
  - assume all logical constraints

```
inhale acc(x.f) && x.f == 2
```



- **`exhale A`** means:
  - assert all logical constraints
  - check and remove all permissions required by assertion **A**
  - havoc any locations to which all permission is lost

```
exhale acc(x.f) && x.f == 2
```



Encoding of method bodies and calls

```
method foo() returns (…)
  requires A
  ensures  B
{ S }
```

```
x := foo()
```

- Encoding without heap
  - Body
    ```
    assume A
    // encoding of S
    assert B
    ```
  - Call
    ```
    assert A[…]
    havoc x
    assume B[…]
    ```
- Encoding with heap
  - Body
    ```
    inhale A
    // encoding of S
    exhale B
    ```
  - Call
    ```
    exhale A[…]
    havoc x
    inhale B[…]
    ```
- **`inhale`** and **`exhale`** are permission-aware analogues of **`assume`** and **`assert`**

Verifying memory safety

- Memory safety is the absence of errors related to memory accesses, such as, null-pointer dereferencing, access to un-allocated memory, dangling pointers, out-of-bounds accesses, double free, etc.

- Using permissions, Viper verifies memory safety by default

```
var x: Ref
x.f := 5
```

```
var x: Ref
x := null
x.f := 5
```

```
method free(p: Ref)
  requires acc(p.f)
```

model de-allocation via method call

```
free(x)
x.f := 5
```

```
free(x)
free(x)
```

Heaps

- Encode references and fields

```
type Ref              // type for references
const null: Ref       // null references

type Field T          // polymorphic type for field names
```

```
field f: Int
field g: Ref
```

```
const f: Field int
const g: Field Ref
```

- Heaps map references and field names to values

```
type HeapType = <T>[Ref, Field T]T    // polymorphic map
```

- Represent the program heap as global variable

```
var Heap: HeapType
```

Permissions and field access

- Permissions are tracked in a global permission mask

```
type MaskType = <T>[Ref, Field T]bool
var Mask: MaskType
```

- Convention: ¬`Mask[null, f]` for all fields `f`

- Field access

```
v := x.f
```

```
assert Mask[x,f]
v := Heap[x,f]
```

```
x.f := E
```

```
assert Mask[x,f]
Heap[x,f] := E
```

  - Field access requires permission!

Inhale

- **`inhale A`** means:
  - obtain all permissions required by assertion **A**
  - assume all logical constraints
- Encoding is defined recursively over the structure of **A**

`inhale E`	`assume [[E]]`	
`inhale acc(E.f)`	`assume ¬Mask[[[E]],f]` `Mask[[[E]],f] := true`	Reaching more than full permission goes to magic
`inhale E => A`	`if([[E]]) { [[inhale A]] }`	
`inhale A && B`	`[[inhale A]]; [[inhale B]]`	Separating conjunction: add sum of permissions

- The encoding also asserts that E is well-defined (omitted here)

Exhale (simplified)

- **`exhale A`** means:
  - assert all logical constraints
  - check and remove all permissions required by assertion **A**
  - havoc any locations to which all permission is lost
- Encoding is defined recursively over the structure of **A**

`exhale E`	`assert [[E]]`	

`exhale acc(E.f)`	`assert Mask[[[E]],f]`
	`Mask[[[E]],f] := false`
	`havoc Heap[[[E]],f]`	

`exhale E => A`	`if([[E]]) { [[exhale A]] }`	
`exhale A && B`	`[[exhale A]]; [[exhale B]]`	Separating conjunction: remove sum of permissions

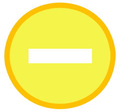
- The encoding also asserts that E is well-defined (omitted here)

Challenges revisited

Heap data structures pose three major challenges for sequential verification

- Reasoning about aliasing
  - Permissions and separating conjunction

- Framing, especially for dynamic data structures
  - Sound frame rule, but no support yet for unbounded data structures



- Writing specifications that preserve information hiding
  - Not solved, but see next section

And additional challenges for concurrent programs, e.g., data races

- Permissions are an excellent basis, but see later

Outline

- Automated program verification
- Reasoning about the heap
- Abstraction
- Concurrency
- Conclusion

Running example: linked lists

```
field elem: Int
field next: Ref

method head(this: Ref) returns (res: Int)
  requires acc(this.elem)
  ensures  acc(this.elem)
  ensures  res == this.elem
{
  res := this.elem
}
```

```
method append(this: Ref, e: Int)
  requires // permission to all nodes
  ensures  // list was extended
{
  if(this.next == null) {
    var n: Ref
    n := new(*)
    n.next := null
    this.elem := e
    this.next := n
  } else {
    append(this.next, e)
  }
}
```

- Specification reveals implementation details
- Permissions and behavior cannot be expressed so far

Predicates

- User-defined predicates consist of a predicate name, a list of parameters, and a self-framing assertion

Declarations

$D ::= \; \ldots \mid \mathbf{predicate}\; P(\overline{x} \colon \overline{T}) \; \{ \; A \; \}$

```
predicate node(this: Ref) {
  acc(this.elem) && acc(this.next)
}
```

- Predicate instances are assertions

Assertions

$A ::= \; \ldots \mid P(\overline{E})$

```
method head(this: Ref) returns (res: Int)
  requires node(this)
  ensures  node(this)
{ … }
```

Recursive predicates

- Predicate definitions may be recursive

Declarations

D ::= ... | **predicate** P($\overline{p}$: $\overline{T}$) { A }

Assertions

A ::= ... | P($\overline{E}$)

- Recursive predicates may denote a statically-unbounded number of permissions

```
predicate list(this: Ref) {
  acc(this.elem) && acc(this.next) &&
  (this.next != null ==> list(this.next))
}
```

- Recursive predicate definitions are interpreted as least fixed points

- All instances of the predicate have finite unfoldings

- If `list(x)` holds, we have `x!=x.next`

- `list` describes a finite linked list

Static verification with recursive predicates

- A program verifier in general cannot know statically how far to unfold recursive definitions

```
predicate list(this: Ref) {
  acc(this.next) &&
  (this.next != null ==> list(this.next))
}
```

```
inhale list(x)
y.next := null  // do we have permission?
```

Iso-recursive predicates

- An iso-recursive semantics distinguishes between a predicate instance and its body

```
predicate list(this: Ref) {
  acc(this.elem) && acc(this.next) &&
  (this.next != null ==> list(this.next))
}
```

```
inhale list(x)
x.next := null  // no permission
```

- Intuition: permissions are held by method executions, loop iterations, or predicate instances

Folding and unfolding predicates

- Exchanging a predicate instance for its body, and vice versa, is done via extra statements in the program

Statements

S ::= ...

| **fold** P($\bar{E}$)

| **unfold** P($\bar{E}$)

```
predicate list(this: Ref) {
  acc(this.elem) && acc(this.next) &&
  (this.next != null ==> list(this.next))
}
```

- An unfold statement exchanges a predicate instance for its body

```
inhale list(x)
unfold list(x)
x.next := null
```

- A fold statement exchanges a predicate body for a predicate instance

```
inhale list(x)
unfold list(x)
x.next := null
fold list(x)
exhale list(x)
```

Encoding of predicates

- Recall that permissions are tracked in a global permission mask

```
type MaskType = <T>[Ref, Field T]bool
var Mask: MaskType
```

- We use the same mask to track predicate instances

- An unfold statement exchanges a predicate instance for its body

```
unfold P(Ē)
```

```
exhale P(Ē)
inhale body(P(Ē))
```

- A fold statement exchanges a predicate body for a predicate instance

```
fold P(Ē)
```

```
exhale body(P(Ē))
inhale P(Ē)
```

Representation invariants

- Data structures typically maintain several consistency conditions
  - Value constraints, e.g., references being non-null or integers being positive
  - Structural constraints, e.g., a tree being balanced

- Such representation invariants are
  - Established by constructors
  - Assumed and preserved by all operations

- Representation invariants can be expressed as part of a predicate

```
predicate list(this: Ref) {
  acc(this.elem) && acc(this.next) &&
  (this.next != null ==> list(this.next) &&
                           0 <= this.elem)
}
```

```
method append(this: Ref, e: Int)
  requires list(this)
  ensures  list(this)
{
  unfold list(this)  // assume invariant
  …
  fold list(this)    // check invariant
}
```

Unfolding-expressions

- Unfold and fold are statements because they change the state (heap and mask)

- Unfolding-expressions allow one to temporarily unfold a predicate during the evaluation of an expression

Expressions

$$E ::= \ \dots \ \mid \ \textbf{unfolding}\ P(\overline{E})\ \textbf{in}\ E'$$

- They enable inspecting fields whose permissions are folded inside a predicate

```
predicate list(this: Ref) {
  acc(this.elem) && acc(this.next) && acc(this.len) &&
  (this.next == null ==> this.len == 0) &&
  (this.next != null ==> list(this.next) &&
      unfolding list(this.next) in this.len == this.next.len + 1)
}
```

Specifying functional behavior

- Using old-expressions and unfolding-expressions, we can specify some aspects of functional behavior

- But: Approach does not work when behavior depends on an unbounded number of fields (e.g., sorting a list)

- And: specifications reveal implementation details

```
predicate list(this: Ref) {
  acc(this.next) && acc(this.len) &&
  (this.next == null ==> this.len == 0) &&
  (this.next != null ==> list(this.next) &&
      unfolding list(this.next) in
      this.len == this.next.len + 1)
}
```

```
method append(this: Ref, e: Int)
  requires list(this)
  ensures  list(this)
  ensures  (unfolding list(this) in this.len) ==
        old(unfolding list(this) in this.len + 1)
```

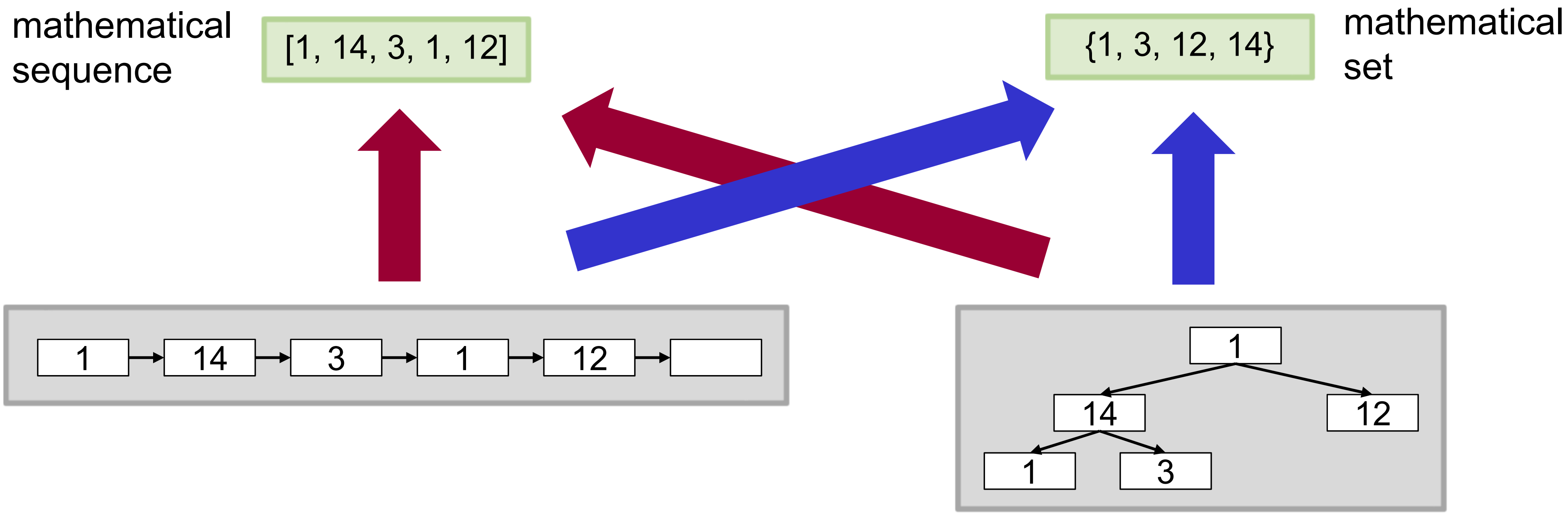
Challenges revisited

Heap data structures pose three major challenges for sequential verification

- Reasoning about aliasing
  - Permissions and separating conjunction
- Framing, especially for dynamic data structures
  - Sound frame rule, predicates
- Writing specifications that preserve information hiding
  - Not solved

Data abstraction

- To write implementation-independent specifications, we map the concrete data structure to mathematical concepts and specify the behavior in terms of those



Data abstraction via abstraction functions

- Viper provides heap-dependent functions
  - side-effect free
  - terminating
  - deterministic
- Function bodies are expressions
- Functions may be recursive, but termination is not checked by default

```
function content(this: Ref): Seq[Int]
{
  this.next == null ?
    Seq[Int]() :
    Seq(this.elem) ++ content(this.next)
}
```

(incomplete declaration)

Expressions

$E ::= \ \dots \mid f(\overline{E})$

Encoding of heap-dependent functions

- Heap-dependent functions are encoded as uninterpreted functions

- Function body is encoded as a definitional axiom

```
function f(x: T): T' {
  E
}
```

```
function f(x: T, h: HeapType): T'

axiom forall x: T, h: HeapType :: f(x, h) == [[E]]
```

(will be revised later)

  - [[_]] is the encoding function (omitted for types), parametric in the heap
  - A proof obligation checks that the function body is well-defined (omitted here)

- Function calls are encoded as applications of these functions

```
f(E)
```

```
f([[E]], Heap)
```

Another frame problem

```
function content(this: Ref): Seq[Int]
{
  this.next == null ?
    Seq[Int]() :
    Seq(this.elem) ++ content(this.next)
}
```

```
// assume we have list(x) && acc(y.f)
tmp := content(x)
y.f := 5
assert tmp == content(x)
```

```
tmp := content(x, Heap)
assert Mask[y,f]
Heap[y,f] := 5
assert tmp == content(x, Heap)
```

- Each heap update modifies the (global) heap
- Any information about heap-dependent functions is lost
- Recovering the information by inspecting the function body would violate information hiding and would not work for abstract functions

Read effects

- Heap-dependent functions must have a precondition that frames the function body, that is, provides all permissions to evaluate the body
- The precondition over-approximates the locations the function value depends on (its read effect)
- If permission to a location is not included in the precondition, modifying it cannot affect the function value, which allows framing

```
function content(this: Ref): Seq[Int]
  requires list(this)
{
  unfolding list(this) in
  (this.next == null ?
    Seq[Int]() :
    Seq(this.elem) ++ content(this.next)
  )
}
```

```
// assume we have list(x) && acc(y.f)
tmp := content(x)
y.f := 5
assert tmp == content(x)
```

Framing axioms

- The read effect is used to generate a framing axiom for the function
- If two heaps agree on a function's read effect then the function yields the same result in both heaps

```
function get(x: Ref): Int
  requires acc(x.elem)
{ … }
```

```
function get(x: Ref, h: HeapType): int

axiom forall x: Ref, h1: HeapType, h2: HeapType ::
  h1[x,elem] == h2[x,elem] ==> get(x, h1) == get(x, h2)
```

Actual axiom is more complex to break symmetry, which causes unnecessary quantifier instantiations

- The encoding for predicates in function preconditions is analogous, but needs to consider all heap locations included in a predicate

Partial functions

- Preconditions of heap-dependent functions specify the read effect

```
function length(this: Ref): Int
  requires list(this)
{ … }
```

- Like method preconditions, they may also constrain the function arguments (including the heap)

```
function first(this: Ref): Int
  requires list(this) && 0 < length(this)
{
  content(this)[0]
}
```

- Definitional axioms provide a partial definition of the (total) uninterpreted function

```
function f(x: T): T’
  requires A
{ E }
```

```
function f(x: T, h: HeapType): T’

axiom forall x: T, h: HeapType ::
      [[A]] ==> f(x, h) == [[E]]
```

Challenges revisited

Heap data structures pose three major challenges for sequential verification

- Reasoning about aliasing
  - Permissions and separating conjunction
- Framing, especially for dynamic data structures
  - Sound frame rule, predicates
- Writing specifications that preserve information hiding
  - Data abstraction, heap-dependent functions

Outline

- Automated program verification
- Reasoning about the heap
- Abstraction
- Concurrency
- Conclusion

Reasoning about concurrent programs – challenges

```
x.f := x.f + 1   ||   x.f := x.f + 1
```

Data races

```
acquire x        ||
x.f := 5         ||   acquire x
release x        ||   x.f := 0
acquire x        ||   release x
y := 10 / x.f    ||
release x        ||
```

Reasoning about thread interference

```
acquire x   ||   acquire y
acquire y   ||   acquire x
…           ||   …
release x   ||   release x
release y   ||   release y
```

Deadlock

```
x.f := 0
acquire x         ||   acquire x
x.f := x.f + 1    ||   x.f := x.f + 1
release x         ||   release x
acquire x
assert x.f == 2
```

Reasoning about thread cooperation

Thread-modular verification

- All verification techniques introduced so far are procedure-modular
  - Reason about calls in terms of the callee's specification
  - Verification of a method does not consider callers or implementation of callees

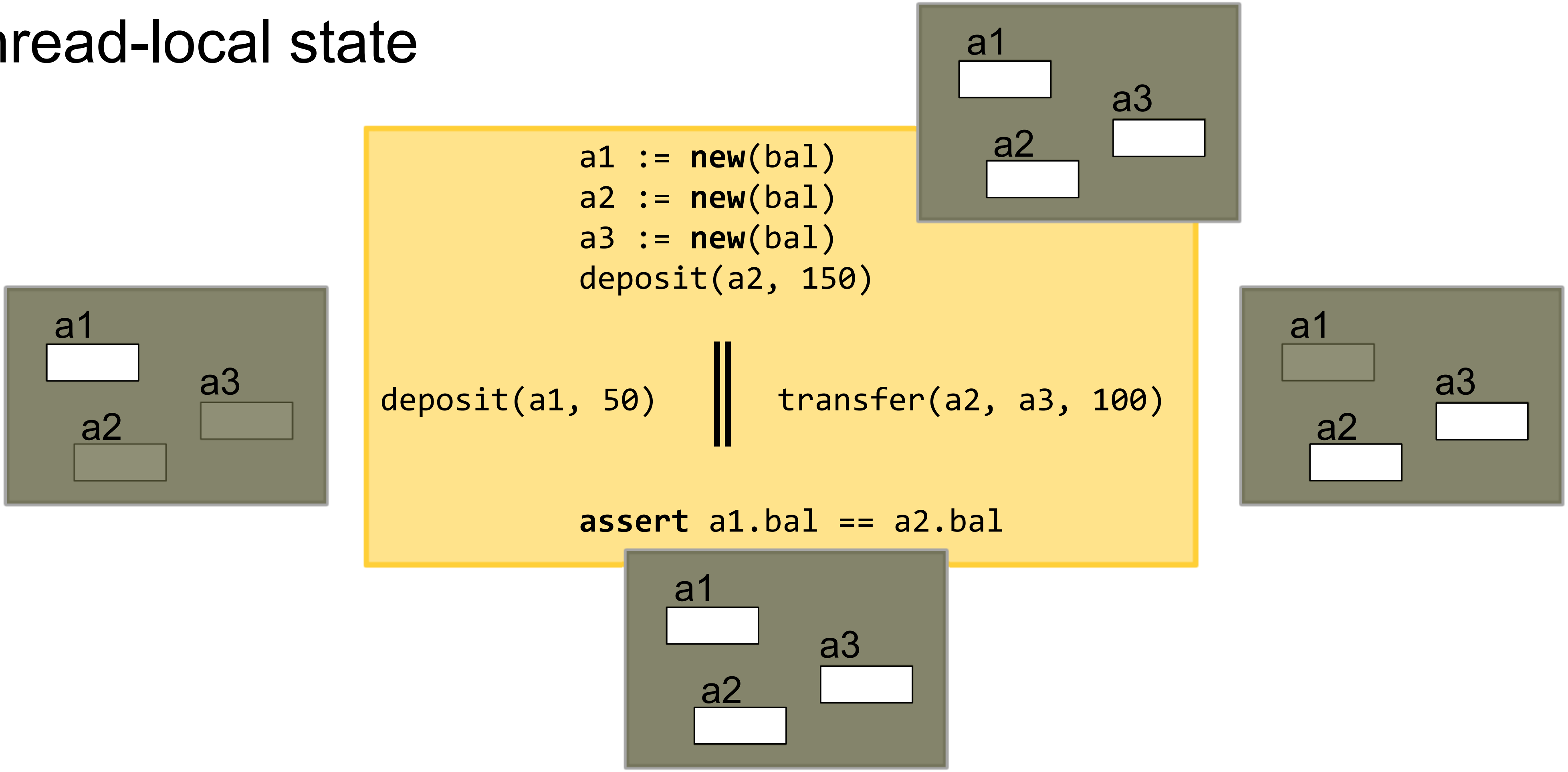
```
method create() returns (res: Ref)
  ensures  list(res)
  ensures  content(res) == Seq[Int]()
{
  res := new(*)
  res.next := null
  fold list(res)
}
```

- We will now present techniques that are also thread-modular
  - Reason about a thread execution without knowing which other threads might run concurrently

```
acquire x                 ||    acquire x
x.f := 5                  ||    x.f := 0
release x                 ||    release x
acquire x                 ||
y := 10 / x.f             ||
release x                 ||
```

- Both forms of modularity are crucial for verification to scale

Thread-local state



- The parallel branches operate on disjoint memory; data races are not possible

Structured parallelism

- Permissions and separating conjunction lead to a simple proof rule

$$\frac{\{\ \mathbf{A}_1\ \}\ \mathtt{S}_1\ \{\ \mathbf{B}_1\ \} \qquad \{\ \mathbf{A}_2\ \}\ \mathtt{S}_2\ \{\ \mathbf{B}_2\ \}}{\{\ \mathbf{A}_1 * \mathbf{A}_2\ \}\ \mathtt{S}_1 \parallel \mathtt{S}_2\ \{\ \mathbf{B}_1 * \mathbf{B}_2\ \}}$$

where S_1 does not assign to local variables free in S_2, $\mathbf{A}_2$, or $\mathbf{B}_2$ (and analogous for S_2)

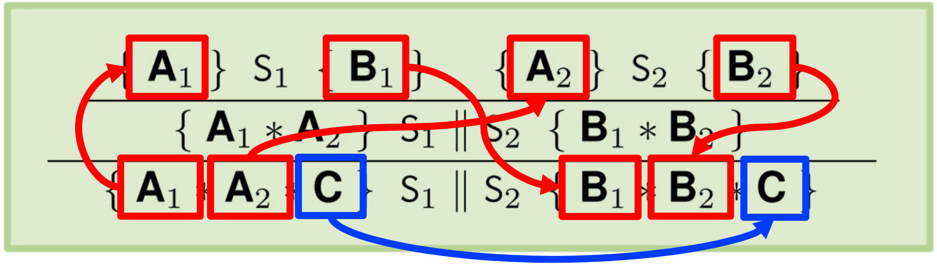
- Separating conjunction prevents interference between the parallel branches (since the only potentially-shared memory is the heap)

- Programs with data races have an unsatisfiable precondition

$$\frac{\{\ \mathbf{acc}(\mathtt{x.f})\ \}\ \mathtt{x.f\ :=\ 7}\ \{\ \ldots\ \} \qquad \{\ \mathbf{acc}(\mathtt{x.f})\ \}\ \mathtt{y\ :=\ x.f}\ \{\ \ldots\ \}}{\{\ \mathbf{acc}(\mathtt{x.f}) * \mathbf{acc}(\mathtt{x.f})\ \}\ \mathtt{x.f\ :=\ 7} \parallel \mathtt{y\ :=\ x.f}\ \{\ \ldots\ \}}$$

Encoding structured parallelism

- The proof rule employs the familiar permission transfer



- We can encode this proof rule via exhale and inhale operations

```
method left(…) returns (res1: T)
  requires A1
  ensures  B1
{ // encoding of S1 }
```

Encode left and right branch
as methods with specifications

```
exhale A1[…]
exhale A2[…]
havoc res1, res2
inhale B1[…]
inhale B2[…]
```

Encode parallel composition like
two half method calls
(adjusted to handle old-expressions)

Example: parallel list search

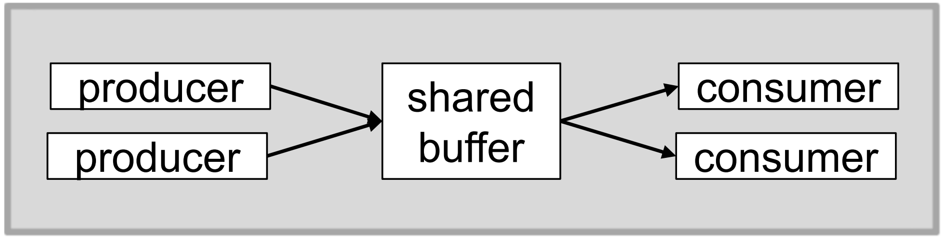
```
method busy(courses: Ref, seminars: Ref, exams: Ref, today: Int) returns (res: Bool)
  requires list(courses) && list(seminars) && list(exams)
  ensures  list(courses) && list(seminars) && list(exams)
  ensures  res == (today in content(courses) ||
                   today in content(seminars) ||
                   today in content(exams))
{
                                           ‖  var rightRes: Bool
                                           ‖  rightRes := contains(seminars, today)
  var leftRes: Bool                        ‖  var res2: Bool
  leftRes := contains(courses, today)      ‖  res2 := contains(exams, today)
                                           ‖  rightRes := rightRes || res2

  res := leftRes || rightRes
}
```

Shared state

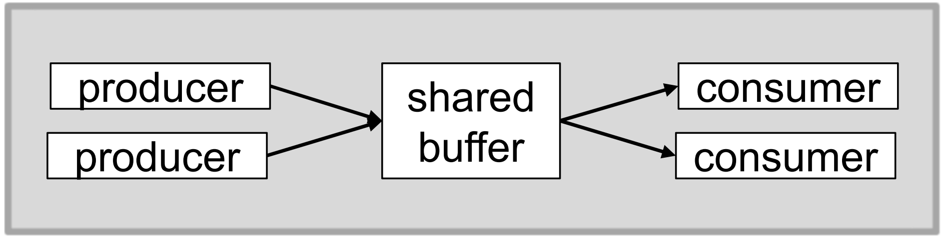
- The solution presented so far supports concurrency with thread-local state
- Threads exchange information upon fork and join, but cannot communicate or collaborate while they are running
- Communication between threads is typically supported by shared state or message passing
- We will focus on shared state, but message passing can also be supported using permissions

- Example: Producer-Consumer



- Concurrent accesses to mutable shared state require synchronization to prevent data races and ensure correctness
- We will focus on locks as a synchronization primitive

Synchronization via locks




```
method produce(buf: Ref)
{
  while(true) {
    acquire buf
    if(buf.val == null) {
      buf.val := new()
    }
    release buf
  }
}
```

```
method consume(buf: Ref)
{
  while(true) {
    acquire buf
    if(buf.val != null) {
      // consume buf.val
      buf.val := null
    }
    release buf
  }
}
```

- Permission to access buf.val cannot be obtained via the preconditions (that would prevent concurrent executions)
- Intuitively, permissions are obtained by acquiring a lock

Lock invariants

- A lock guards accesses to certain memory locations

```
class Buffer {
  @GuardedBy("this")
  Product val;
}
```

Java provides annotations to document which locations are guarded by a lock

- We associate each lock with a lock invariant

```
class Buffer {
  lock invariant acc(this.val)
  Product val;
}
```

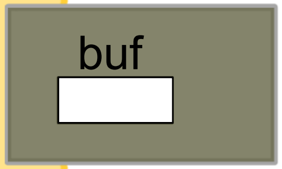
Permissions in the lock invariant express which locations are guarded by the lock

- Intuition: permissions are held by method executions, loop iterations, predicate instances, or locks

Locks and permission transfer

```
class Buffer {
  lock invariant acc(this.val)
  Product val;
}
```


```
method produce(buf: Ref)
{
  while(true) {
    acquire buf
    if(buf.val == null) {
      buf.val := new()
    }
    release buf
  }
}
```




```
method consume(buf: Ref)
{
  while(true) {
    acquire buf
    if(buf.val != null) {
      // consume buf.val
      buf.val := null
    }
    release buf
  }
}
```

More on lock invariants

- A lock invariant holds whenever the lock is not currently being held by a thread

- Lock invariants contain arbitrary self-framing assertions

```
acc(this.val) && 0 < this.val
```

```
list(this) && 0 < length(this)
```

- Self-framingness is crucial for soundness

```
0 < this.val
```

Methods could violate the invariant without acquiring the lock

Simplified encoding of locks

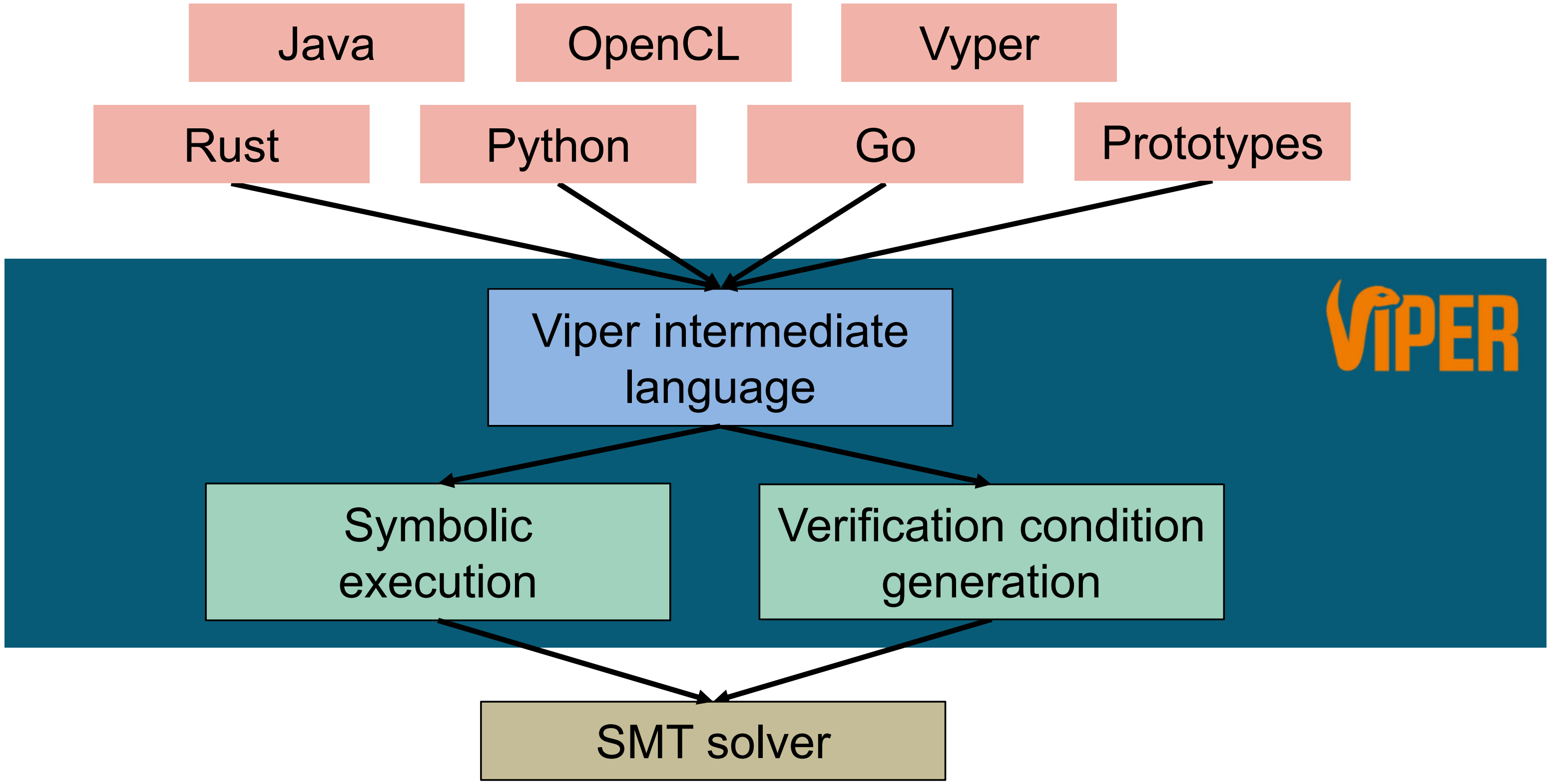
- Locks are encoded as references
- We model non-reentrant locks (repeated acquire leads to deadlock)
- Therefore, each acquire obtains permissions from the lock

acquire x	**inhale** Inv(x)
release x	**exhale** Inv(x)

- The rule for `acquire` does not prevent deadlock; extra proof obligations can be imposed to ensure that locks are acquired in an order

Outline

- Automated program verification
- Reasoning about the heap
- Abstraction
- Concurrency
- Conclusion


Java
OpenCL
Vyper
Rust
Python
Go
Prototypes
Viper intermediate language
Symbolic execution
Verification condition generation
SMT solver

Example: Go verification in Gobra

```
requires acc(x) && acc(y)
ensures  acc(x) && acc(y)
ensures  *x == old(*y)
ensures  *y == old(*x)
func swap(x *int, y *int) {
    tmp := *x
    *x = *y
    *y = tmp
}
```

gobra

- Go supports pointers to integers
- Parameters can be assigned to
- Locals get initialized by default

```
field val: Int

method swap(x: Ref, y: Ref)
  requires acc(x.val) && acc(y.val)
  ensures  acc(x.val) && acc(y.val)
  ensures  x.val == old(y.val)
  ensures  y.val == old(x.val)
{
  var yLocal: Ref  // declare locals
  var xLocal: Ref

  xLocal := x      // copy parameters
  yLocal := y

  var tmp: Int     // declare tmp
  inhale tmp == 0

  tmp := xLocal.val            // tmp = *x
  xLocal.val := yLocal.val     // *x = *y
  yLocal.val := tmp            // *y = tmp
}
```

Exposing the verification logic

- Gobra's specification and verification technique is very similar to Viper's

- Developers need to use permissions, declare predicates, use unfold and fold statements, etc.

```
requires acc(x) && acc(y)
ensures  acc(x) && acc(y)
ensures  *x == old(*y)
ensures  *y == old(*x)
func swap(x *int, y *int) {
    tmp := *x
    *x = *y
    *y = tmp
}
```

- The overhead for programmers is substantial (both amount and complexity of annotations)

- Many existing verifiers take this approach because it enables modular verification of programs in mainstream languages, including concurrent and heap-manipulating programs

Ownership types in Rust

```
fn swap(x: &mut i32, y: &mut i32) {
  let tmp = *x;
  *x = *y;
  *y = tmp;
}

fn client()
{
  let mut a = 17;
  swap(&mut a, &mut a);
}
```

```
error[E0499]: cannot borrow `a` as
mutable more than once at a time
  --> .\swap.rs:11:26
   |
11 |     swap(&mut a, &mut a);
   |     ---- ------  ^^^^^^
        second mutable borrow occurs here

error: aborting due to previous error
```

- Rust’s type system tracks ownership of memory locations
- It guarantees memory safety
- Can we leverage this guarantee to simplify verification?

Example: Rust verification in Prusti

```
#[ensures(*x == old(*y) )]
#[ensures(*y == old(*x) )]
fn swap(x: &mut i32, y: &mut i32) {
  let tmp = *x;
  *x = *y;
  *y = tmp;
}
```

P∗rust→∗i

- Prusti extracts permissions (and predicates) automatically from type information
- A Viper "core proof" of memory safety is generated completely automatically
- Users can add functional correctness specifications, by using a slight extension of Rust expressions

The overhead for programmers is substantially reduced
(both amount and complexity of annotations)

Comparison of annotation overhead: List zip example

```
#![feature(box_patterns)]

use prusti_contracts::*;

struct Node {
  elem: i32,
  next: List,
}

enum List {
  Empty,
  More(Box<Node>),
}

impl List {
  #[pure]
  #[ensures(result >= 0)]
  fn len(&self) -> usize {
    match self {
      List::Empty => 0,
      List::More(box node) =>
        1 + node.next.len(),
    }
  }

  #[ensures(result.len() ==
            self.len() + that.len())]
  pub fn zip(&self, that: &List) -> List {
    match self {
      List::Empty => that.cloneList(),
      List::More(box node) => {
        let new_node = Box::new(Node {
          elem: node.elem,
          next: that.zip(&node.next),
        });
        List::More(new_node)
      }
    }
  }
```

```
  #[ensures(result.len() == self.len())]
  pub fn cloneList(& self) -> List {
    match self {
      List::Empty => List::Empty,
      List::More(box node) => {
        let new_node = Box::new(Node {
          elem: node.elem,
          next: node.next.cloneList(),
        });
        List::More(new_node)
      }
    }
  }
}
```

```
field next: Ref
field elem: Int

predicate list(this: Ref) {
  acc(this.elem) && acc(this.next) &&
  (this.next != null ==> list(this.next))
}

function len(this: Ref): Int
  requires acc(list(this), wildcard)
{
  unfolding acc(list(this), wildcard) in
(this.next == null ? 0 : len(this.next) + 1)
}

method zip(this: Ref, that: Ref)
                           returns (res: Ref)
  requires acc(list(this), 1/2) &&
           acc(list(that), 1/2)
  ensures  acc(list(this), 1/2) &&
           acc(list(that), 1/2)
  ensures  list(res)
  ensures  res != null
  ensures  len(res) == len(this) + len(that)
{
  unfold acc(list(this), 1/2)
  if(this.next == null) {
    res := cloneList(that)
  } else {
    res := new(*)
    res.elem := this.elem
    var rest: Ref
    rest := zip(that, this.next)
    res.next := rest
    fold list(res)
  }
  fold acc(list(this), 1/2)
}
```

```
method cloneList(this: Ref) returns (res: Ref)
  requires acc(list(this), 1/2)
  ensures  acc(list(this), 1/2) && list(res)
  ensures  res != null
  ensures  len(res) == len(this)
{
  res := new(*)
  unfold acc(list(this), 1/2)
  if(this.next == null) {
    res.next := null
  } else {
    var tmp: Ref
    tmp := cloneList(this.next)
    res.elem := this.elem
    res.next := tmp
  }
  fold acc(list(this), 1/2)
  fold list(res)
}
```

Expressiveness

Language features

- Imperative code
- Object-oriented code
- Nominal, structural, and dynamic typing
- Closures
- Multithreading with shared state and message passing
- Weak-memory concurrency

Properties

- Memory safety
- Absence of overflows
- Termination
- Functional correctness
- Race freedom
- Deadlock freedom
- Secure information flow
- Resource manipulation
- Worst-case execution time

Limitations

- Limitations inherited from the SMT solver
  - Undecidable theories may lead to spurious errors
  - Verification time for large methods

- Annotation overhead
  - Typically 2-5 lines of annotations per line of code

- Trust assumptions
  - Correctness of SMT solver
  - Correctness of Viper
  - Correctness of front-end encoding

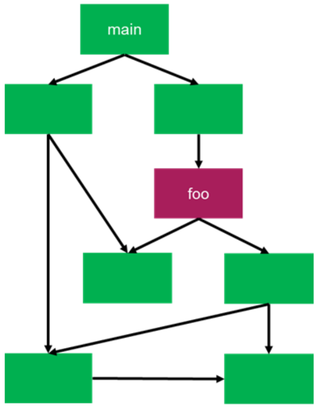
Verifiers developed at ETH

- Verification infrastructure for permission-based reasoning
- Basis for our other verifiers
- viper.ethz.ch

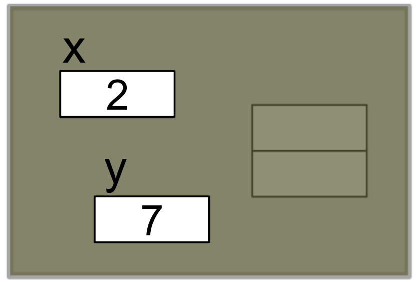
- Modular verification of Go programs
- Used for large-scale verification projects, e.g., verifiedSCION
- gobra.ethz.ch

- Modular verification of Rust programs
- Leverages Rust type system to simplify verification
- prusti.ethz.ch

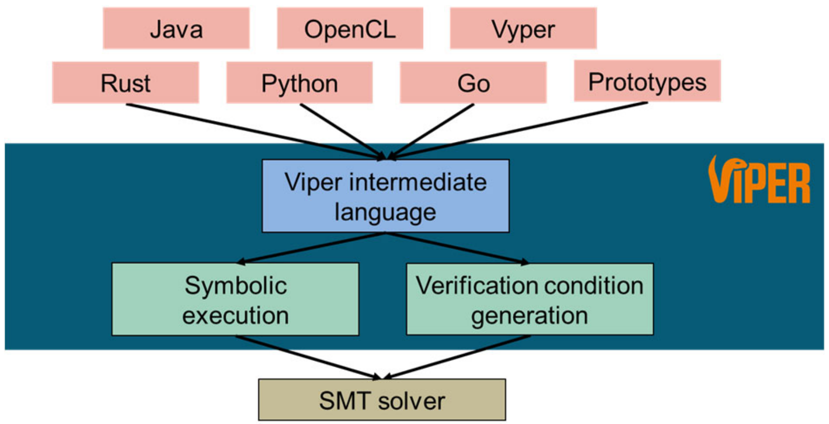
- Modular verification of Python programs
- Correctness and security properties
- Variant for Ethereum smart contracts in Vyper
- www.pm.inf.ethz.ch/research/nagini.html



Modularity is important for scalability, components, and evolution



Permissions enable modular reasoning about resources



Intermediate languages enable reuse of infrastructure

Viper lets you encode a wide variety of reasoning techniques

References

- John C. Reynolds:
  Separation Logic: A Logic for Shared Mutable Data Structures. LICS, 2002

- Matthew Parkinson and Gavin Bierman:
  Separation logic and abstraction. POPL, 2005

- Peter W. O'Hearn:
  Resources, Concurrency and Local Reasoning. CONCUR, 2004

- K. Rustan M. Leino and Peter Müller:
  A Basis for Verifying Multi-threaded Programs. ESOP, 2009

- Peter Müller, Malte Schwerhoff, and Alexander J. Summers:
  Viper: A Verification Infrastructure for Permission-Based Reasoning. VMCAI, 2016