BUILDING DEDUCTIVE PROGRAM VERIFIERS
Intermediate verification language

Backend verifier

SMT solver
Outline

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

### Types

\[ T ::= \text{Bool} \mid \text{Int} \mid \text{Rational} \mid \text{Real} \]

### Expressions

\[ E ::= \text{constant} \mid \text{variable} \mid E + E \mid E \times E \mid E - E \]
\[ \mid E < E \mid E \land E \mid E \lor E \mid \sim E \mid \ldots \quad (+ \text{syntactic sugar}) \]

### Assertions

\[ A ::= E \mid \forall x : T ::= A \mid \exists x : T ::= A \mid A \Rightarrow A \mid \ldots \]

### Program statements

\[ S ::= v := E \quad \text{assignment} \]
\[ \mid S ; S \quad \text{sequential composition} \]
\[ \mid \text{if} \ (\ast) \{S\} \text{ else } \{S\} \quad \text{nondeterministic choice} \]
\[ \mid \text{assert} \ A \quad \text{assertion} \]
\[ \mid \text{assume} \ A \quad \text{assumption} \]
\[ \mid \text{havoc} \ v \quad \text{nondeterministic assignment} \]

All types are mathematical (unbounded)

We assume that expressions and programs are well-typed
Hoare logic

\begin{align*}
\{ A[E/x] \} & \quad \text{x := E} \quad \{ A \} \\
\{ A \} & \quad S \quad \{ C \} \quad \{ C \} \quad S' \quad \{ B \} \\
& \quad \{ A \} \quad S \quad S' \quad \{ B \} \\
\{ A \} & \quad S \quad \{ C \} \quad \{ B \} \quad S' \quad \{ C \} \\
& \quad \{ A \land B \} \quad \text{if (*)} \quad \{ S \} \quad \text{else} \quad \{ S' \} \quad \{ C \} \\
\forall x : A & \quad \text{havoc x} \quad \{ A \} \\
A & \Rightarrow A' \quad \{ A' \} \quad S \quad \{ B' \} \quad B' \Rightarrow B \\
& \quad \{ A \} \quad S \quad \{ B \} \\
\{ A \land B \} & \quad \text{assert} \quad A \quad \{ B \} \\
\{ A \Rightarrow B \} & \quad \text{assume} \quad A \quad \{ B \} \\
\end{align*}

Our Hoare triples have a partial correctness meaning
Challenges for automating proof search

- Writing Hoare-style proofs requires creativity

\[
\begin{align*}
\{ A \} & \text{s} \{ C \} \\
\{ C \} & \text{s'} \{ B \} \\
\{ A \} & \text{s} \{ B \}
\end{align*}
\]

How do we find intermediate assertions?

\[
\begin{align*}
A & \Rightarrow A' \\
\{ A' \} & \text{s} \{ B' \} \\
B' & \Rightarrow B
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Statement S</th>
<th>$\text{wp[S]}(B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := E$</td>
<td>$B[E/x]$</td>
</tr>
<tr>
<td>$S; S'$</td>
<td>$\text{wp}<a href="%5Ctext%7Bwp%7D%5BS'%5D(B)">S</a>$</td>
</tr>
<tr>
<td>if (*){ S } else { S' }</td>
<td>$\text{wp}<a href="B">S</a> \land \text{wp}<a href="B">S'</a>$</td>
</tr>
<tr>
<td>assert A</td>
<td>$A \land B$</td>
</tr>
<tr>
<td>assume A</td>
<td>$A \Rightarrow B$</td>
</tr>
<tr>
<td>havoc x</td>
<td>$\forall x : B$</td>
</tr>
</tbody>
</table>

To automate the proof of a triple

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

we decide

$$A \Rightarrow \text{wp}[S](B)$$
Encoding into guarded commands: conditionals

- Other statements can be encoded into guarded commands

- Conditional statements

\[
\text{if} \ (E) \ \{S\} \ \text{else} \ \{S'\}
\]

\[
\frac{\{A \land E\} \ S \ \{B\} \quad \{A \land \neg E\} \ S' \ \{B\}}{\{A\} \ \text{if} \ (E) \ \{S\} \ \text{else} \ \{S'\} \ \{B\}}
\]

can be encoded using nondeterministic choice and assume

\[
[\text{if} \ (E) \ \{S\} \ \text{else} \ \{S'\}] = \text{if} \ (\ast) \ \{\text{assume} \ E; \ S\} \ \text{else} \ \{\text{assume} \ \neg E; \ S'\}
\]
Encoding into guarded commands: loops

- While statements are verified using loop invariants

```latex
\textbf{while} (E) \{S\}
```

- Encoding

\[
\begin{align*}
\text{Hoare logic} \quad &\{I \land E\} S \{I\} \\
&\{I\} \text{ while} (E) S \{I \land \neg E\}
\end{align*}
\]

```latex
\begin{align*}
\text{assert} \ I \\
\text{havoc} \ Loop \ targets
\end{align*}
```

```latex
\begin{align*}
\text{assume} \ I \\
\text{assume} \ E \\
// \text{encoding of} \ S \\
\text{assert} \ I \\
\text{assume} \ false
\end{align*}
```

```latex
\begin{align*}
\text{assume} \ I \\
\text{assume} \ \neg E
\end{align*}
```
Encoding into guarded commands: loop termination

- Termination can be proved with termination measures

- Encoding

```
assume I
havoc Loop targets

assume I
assume E
assert 0 ≤ R
oldR := R
// encoding of S
assert I
assert R < oldR
assume false
```
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(\overline{x}: \overline{T})
  returns (\overline{y}: \overline{T})
  requires A
  ensures B
  \{ S \}
  ```

  assume A
  \hspace{1cm} // 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

  \[ \overline{z} := P(\overline{E}) \]

  where \( x \) is not free in \( E \)

  ```
  assert A[\overline{E}/\overline{x}]
  havoc \overline{z}
  assume B[\overline{E}/\overline{x}][\overline{z}/\overline{y}] 
  ```
Summary

1. **Prog. language, spec. language and methodology**
   - Loops, procedures
   - Safety, functional correctness, termination

2. **Front-end**
   - Translates programs and specifications to IL

3. **Guarded commands language**
   - Can express programs and specifications

4. **Verification condition generator**
   - Extracts proof obligations automatically using wp

5. **SMT solver**
   - 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

- 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

\[
D ::= \ldots \mid \text{field } f : T
\]

- Fields are declared globally

### Types

\[
T ::= \ldots \mid \text{Ref}
\]

- Only one type of references

### Expressions

\[
E ::= \ldots \mid \text{null} \mid E.f
\]

- Pre-defined null-reference

### Statements

\[
S ::= \ldots \\
| v := \text{new}(f) \mid v := \text{new}(\ast) \\
| x.f := E
\]

- Allocation with given list of fields or all fields
- as before
- allocation
- field update
Field access: naïve proof rules

- Naïve approach: treat field accesses like variable assignment

\[
\begin{align*}
\text{Field read} & \quad \{ E \neq \text{null} \land A[E.f/v] \} \quad v := E.f \quad \{ A \} \\
\text{Field update} & \quad \{ x \neq \text{null} \land A[E/x.f] \} \quad x.f := E \quad \{ A \}
\end{align*}
\]

- Additional precondition prevents null-dereferencing

The naïve proof rule for field update is unsound.
Naïve rule for field update ignores aliasing

Field read

\[
\{ E \neq \texttt{null} \land A[E.f / v] \} \ v := E.f \ \{ A \}
\]

Field update

\[
\{ x \neq \texttt{null} \land A[E / x.f] \} \ x.f := E \ \{ A \}
\]

```plaintext
field val: Int

method foo(p: Ref)
{
  var q: Ref
  assume p != null && p.val == 5
  \{ p \neq \texttt{null} \land p \neq \texttt{null} \land p.val = 5 \}
  q := p
  \{ p \neq \texttt{null} \land q \neq \texttt{null} \land q.val = 5 \}
  p.val := 7
  \{ q \neq \texttt{null} \land q.val = 5 \}
  assert q.val == 5
}
```
The frame problem

- 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

```
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
```
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 may contain both permissions and value constraints

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

- 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 \( \land \) 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
  \[ \text{acc}(p.f) \ast \text{acc}(q.f) \implies p \neq q \]

- Viper's && is separating conjunction

- For the call `swap(x, x)`, the precondition is equivalent to false
  \[ \text{method swap}(a: \text{Ref}, b: \text{Ref})
  \text{requires acc}(a.f) \&\& \text{acc}(b.f) \]
Field access: proof rules with permissions

- Each field access requires (and preserves) the corresponding permission

- Permission to a location implies that the receiver is non-null

Field read

\[
\{ \text{acc}(x.f) \ast A[x.f \mapsto v] \} \quad v := x.f \quad \{ \text{acc}(x.f) \ast A \}
\]

Field update

\[
\{ \text{acc}(x.f) \} \quad x.f := E \quad \{ \text{acc}(x.f) \ast x.f = E \}
\]

where \(E\) does not contain field accesses
Framing

- 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

\[
\begin{align*}
\{ \text{acc}(x,f) \} & \quad x.f := 5 & \{ \text{acc}(x,f) * x.f = 5 \} \\
\{ \text{acc}(x,f) * \text{acc}(y,f) * y.f = 7 \} & \quad x.f := 5 & \{ \text{acc}(x,f) * x.f = 5 * \text{acc}(y,f) * y.f = 7 \}
\end{align*}
\]
Framing for method calls

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

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

  p.f := v

  p
}
```

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

set(x, 5)

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

{ acc(x.f) * x.f < 10 } x.f := x.f + 1 { acc(x.f) }
{ acc(x.f) } while(x.f < 10) { ... } { acc(x.f) * ¬x.f < 10 }
{ acc(x.f) * acc(y.f) * y.f = 7 } while(x.f < 10) { ... } { acc(x.f) * ¬x.f < 10 * acc(y.f) * 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

- **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 of method bodies and calls

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

- **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

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

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

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

```plaintext
free(x)
```

```plaintext
free(x)
```

```plaintext
free(x)
```

model de-allocation via method call
Heaps

- Encode references and fields

```go
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

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

- Represent the program heap as global variable

```go
var Heap: HeapType
```
Permissions and field access

- Permissions are tracked in a global permission mask

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

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

- Field access

```plaintext
v := x.f
x.f := E

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

  - `inhale E => A`
    - `if([[E]]) { [[inhale A]] }

  - `inhale A && B`
    - `[[inhale A]]; [[inhale B]]`

- 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]]
```

- 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

- Specification reveals implementation details

```plaintext
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)
  }
}
```

- 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.

```plaintext
Declarations
D ::= ... | predicate P(\bar{x}: \bar{T}) { A }
```

- Predicate instances are assertions.

```plaintext
Assertions
A ::= ... | P(\bar{E})
```

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

```plaintext
method head(this: Ref) returns (res: Int)
    requires node(this)
    ensures node(this)
    { ... }
```
Recursive predicates

- Predicate definitions may be recursive

Declanations
D ::= ... | predicate P(p: T) { A }

Assertions
A ::= ... | P(E)

- Recursive predicate definitions are interpreted as least fixed points

- All instances of the predicate have finite unfoldings

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))
}

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

- list describes a finite linked list
A program verifier in general cannot know statically how far to unfold recursive definitions

```java
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

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

- 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

\[
\begin{align*}
\text{Statements} \\
S ::= \quad & \ldots \\
| \text{fold } P(\overline{E}) \\
| \text{unfold } P(\overline{E})
\end{align*}
\]

\[
\text{predicate list}(\text{this: } \text{Ref}) \\
\quad \text{acc}(\text{this}.\text{elem}) \&\& \text{acc}(\text{this}.\text{next}) \&\& \\
\quad (\text{this}.\text{next} \neq \text{null} \Rightarrow \text{list}(\text{this}.\text{next}))
\]

- An unfold statement exchanges a predicate instance for its body

\[
\begin{align*}
\text{inhale } \text{list}(\text{x}) \\
\text{unfold } \text{list}(\text{x}) \\
\text{x}.\text{next} := \text{null}
\end{align*}
\]

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

\[
\begin{align*}
\text{inhale } \text{list}(\text{x}) \\
\text{unfold } \text{list}(\text{x}) \\
\text{x}.\text{next} := \text{null} \\
\text{fold } \text{list}(\text{x}) \\
\text{exhale } \text{list}(\text{x})
\end{align*}
\]
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

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

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

```plaintext
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

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

```java
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

```plaintext
def 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)

def 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

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

(incomplete declaration)

Expressions

\[ E ::= \ldots | f(\bar{E}) \]
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]]
  ```

  - `[[_]]` 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)
  ```

(Will be revised later)
Another frame problem

- 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

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

```java
// assume we have list(x) && acc(y.f)

tmp := content(x)
y.f := 5
assert tmp == content(x)

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

```javascript
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.

```plaintext
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

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

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

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

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

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

Data races

Release

Deadlock

Reasoning about thread interference

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

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

- 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

```plaintext
a1 := new(bal)
a2 := new(bal)
a3 := new(bal)
deposit(a2, 150)
deposit(a1, 50)    \parallel    transfer(a2, a3, 100)
assert a1.bal == a2.bal
```
Structured parallelism

- Permissions and separating conjunction lead to a simple proof rule

\[
\begin{align*}
\{ A_1 \} & \quad S_1 & \quad \{ B_1 \} & \quad \{ A_2 \} & \quad S_2 & \quad \{ B_2 \} \\
\{ A_1 \ast A_2 \} & \quad S_1 \parallel S_2 & \quad \{ B_1 \ast B_2 \}
\end{align*}
\]

where \( S_1 \) does not assign to local variables free in \( S_2, A_2, \) or \( B_2 \) (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

\[
\begin{align*}
\{ \text{acc}(x.f) \} & \quad x.f := 7 \quad \{ \ldots \} & \quad \{ \text{acc}(x.f) \} & \quad y := x.f \quad \{ \ldots \} \\
\{ \text{acc}(x.f) \ast \text{acc}(x.f) \} & \quad x.f := 7 \parallel y := x.f \quad \{ \ldots \}
\end{align*}
\]
Encoding structured parallelism

- The proof rule employs the familiar permission transfer. We can encode this proof rule via exhale and inhale operations.

- We can encode parallel composition like two half method calls (adjusted to handle old-expressions).

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

```
exhale A1[...]
exhale A2[...]
havoc res1, res2
inhale B1[...]
inhale B2[...]
```
Example: parallel list search

```plaintext
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 leftRes: Bool
  leftRes := contains(courses, today)

  var rightRes: Bool
  rightRes := contains(seminars, today)
  var res2: Bool
  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

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

```java
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
    }
}
```
Lock invariants

- A lock guards accesses to certain memory locations

```java
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

```java
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

```java
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
  
  \[ \text{acc}(\text{this.val}) \land 0 < \text{this.val} \]
  \[ \text{list}(\text{this}) \land 0 < \text{length}(\text{this}) \]

- Self-framingness is crucial for soundness

  \[ 0 < \text{this.val} \]

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

  \[
  \begin{array}{ll}
  \text{acquire } & x \\
  \text{inhale} & \text{Inv}(x) \\
  \text{release } & x \\
  \text{exhale} & \text{Inv}(x)
  \end{array}
  \]

- 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
Example: Go verification in Gobra

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

```go
func swap(x *int, y *int) {
    tmp := *x
    *x = *y
    *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.

- 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

```go
func swap(x *int, y *int) {
    tmp := *x
    *x = *y
    *y = tmp
}
```
Ownership types in Rust

- 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

- 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

```rust
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(node) => 1 + node.next.len(),
        }
    }
}

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

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

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

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

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

- Verification infrastructure for permission-based reasoning
- Basis for our other verifiers
- viper.ethz.ch
Modularity is important for scalability, components, and evolution

Intermediate languages enable reuse of infrastructure

Permissions enable modular reasoning about resources

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