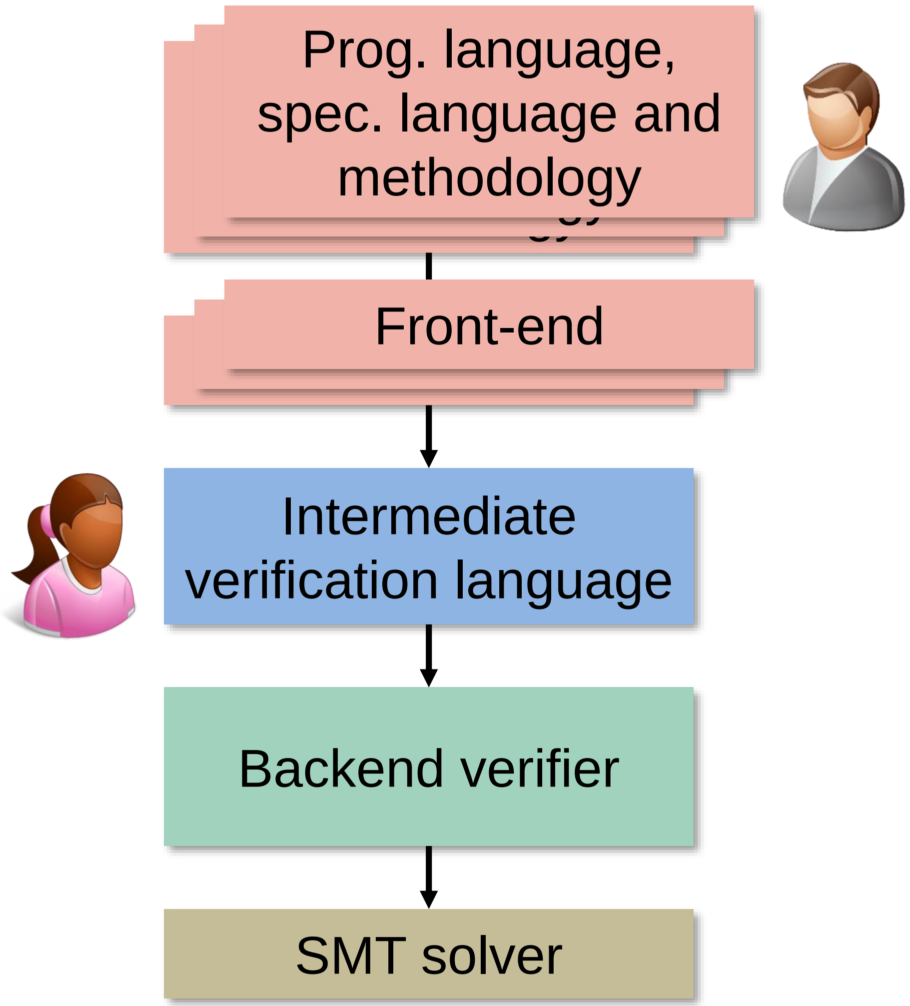
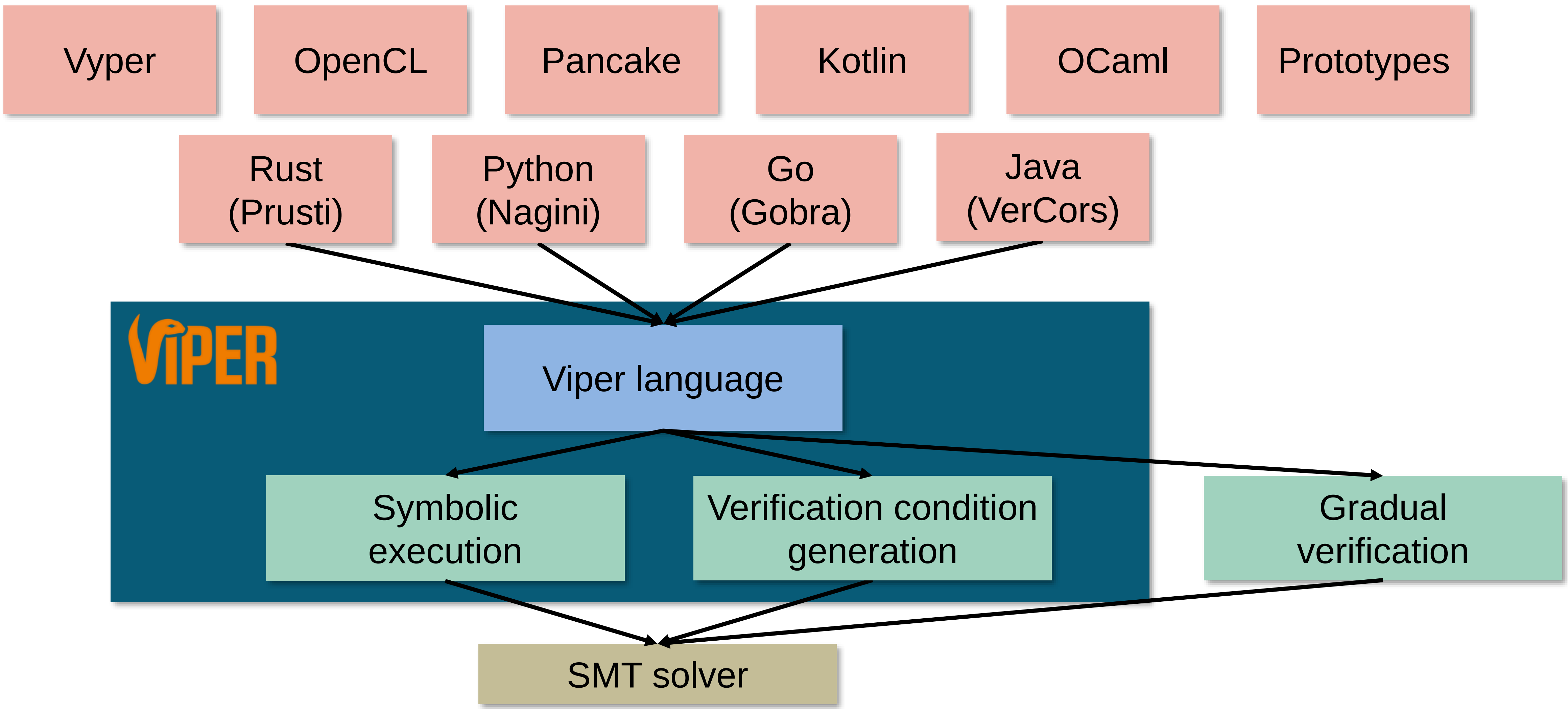
Peter Müller and Thibault Dardinier

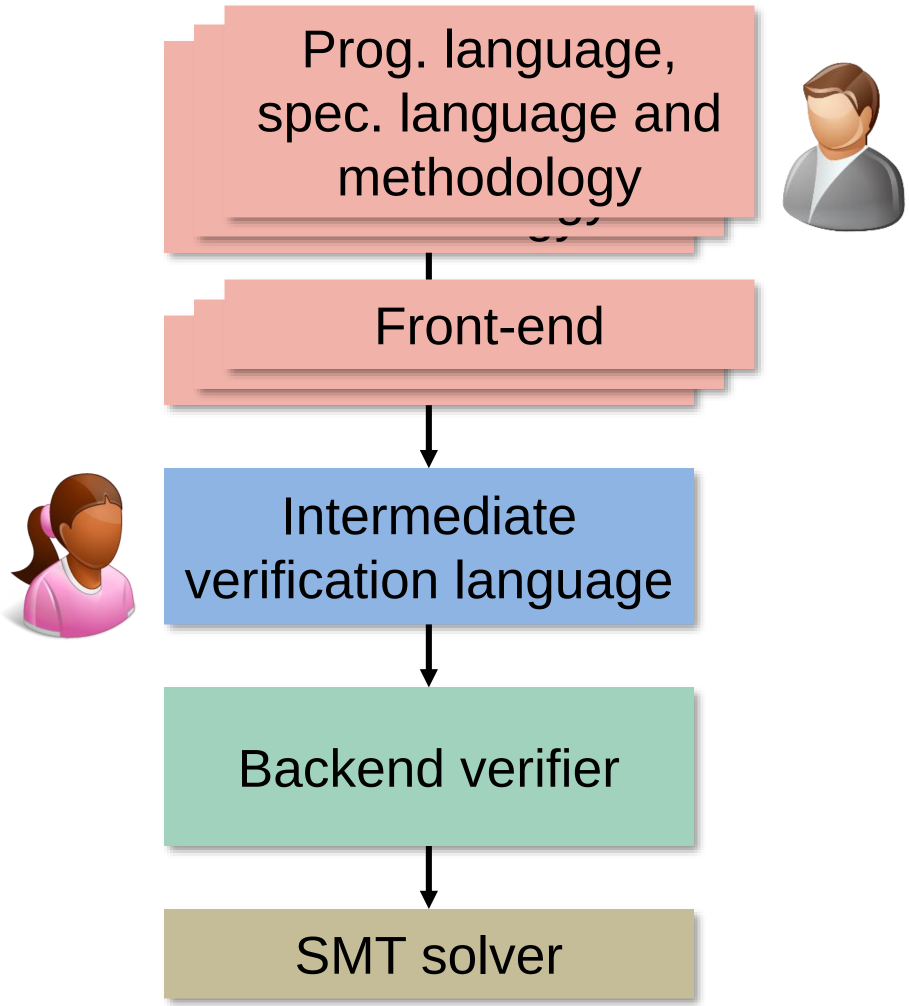
VIPER:
AN INFRASTRUCTURE FOR AUTOMATED VERIFICATION IN SEPARATION LOGIC

ETH zürich


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


Vyper
OpenCL
Pancake
Kotlin
OCaml
Prototypes
Rust (Prusti)
Python (Nagini)
Go (Gobra)
Java (VerCors)
VIPER
Viper language
Symbolic execution
Verification condition generation
Gradual verification
SMT solver

- viper.ethz.ch
- Try online: http://viper.ethz.ch/tutorial
- Install as VS Code extension
- Tutorial: https://sites.google.com/view/vipertutorialpopl2025/home


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

Outline

- Separation logic proofs in Viper
  - Hoare-style verification
  - Permission-based reasoning
  - Abstraction
  - Advanced separation logic
- Viper as target language
- Conclusion

Basics of the Viper language

```
method indexOf(s: Seq[Int], e: Int) returns (res: Int)
  requires 0 < |s|
  ensures res < 0 ==> !(e in s)
  ensures 0 <= res ==> res < |s| && s[res] == e
{
  if(s[0] == e)  { res := 0 }
  else {
    if(|s| == 1) { res := -1 }
    else {
      res := indexOf(s[1..], e)
      if(res != -1) { res := res + 1 }
    }
  }
}
```

- Viper is an imperative, statically-typed, sequential language
- Programs include a sequence of method declarations
- Methods have specifications
- Method bodies contain statements
  - Structured and unstructured control flow

Type system

- Viper has built-in primitive types with the usual operations

```
Bool, Int, …
```

  and built-in generic datatypes

```
Seq[T], Set[T], Multiset[T], Map[S,T]
```

- Programs may declare generic ADTs and uninterpreted sorts (as part of custom theories)

```
adt List[T] {
  Nil()
  Cons(value: T, tail: List[T])
}
```

```
domain List[T] {
  function length(l: List[T]): Int

  axiom nonneg {
    forall l: List[T] :: 0 <= length(l)
  }
}
```

Method specifications

```
method indexOf(s: Seq[Int], e: Int) returns (res: Int)
  requires 0 < |s|
  ensures res < 0 ==> !(e in s)
  ensures 0 <= res ==> res < |s| && s[res] == e
  decreases s
```

- Method specifications may include
  - Preconditions
  - Postconditions
  - A termination measure
- Viper verifies modularly that for all method executions
  - If the preconditions hold in the initial state then the execution will not abort and if the method terminates, the postconditions will hold in the final state
  - That the execution terminates, if a decreases clause is given

Loop annotations

```
method indexOf(s: Seq[Int], e: Int) returns (res: Int)
  ensures res < 0 ==> !(e in s)
  ensures 0 <= res ==> res < |s| && s[res] == e
  decreases s
{
  var i: Int := 0

  while(i < |s| && s[i] != e)
    invariant 0 <= i <= |s|
    invariant forall j: Int :: 0 <= j < i ==> s[j] != e
    decreases |s| - i
  { i := i + 1 }

  res := (i == |s| ? -1 : i)
}
```

- Verification of loops requires invariants
- Termination is verified if a decreases-clause is provided

Outline

- Separation logic proofs in Viper
  - Hoare-style verification
  - Permission-based reasoning
  - Abstraction
  - Advanced separation logic
- Viper as target language
- Conclusion

Heap model: an object-based language

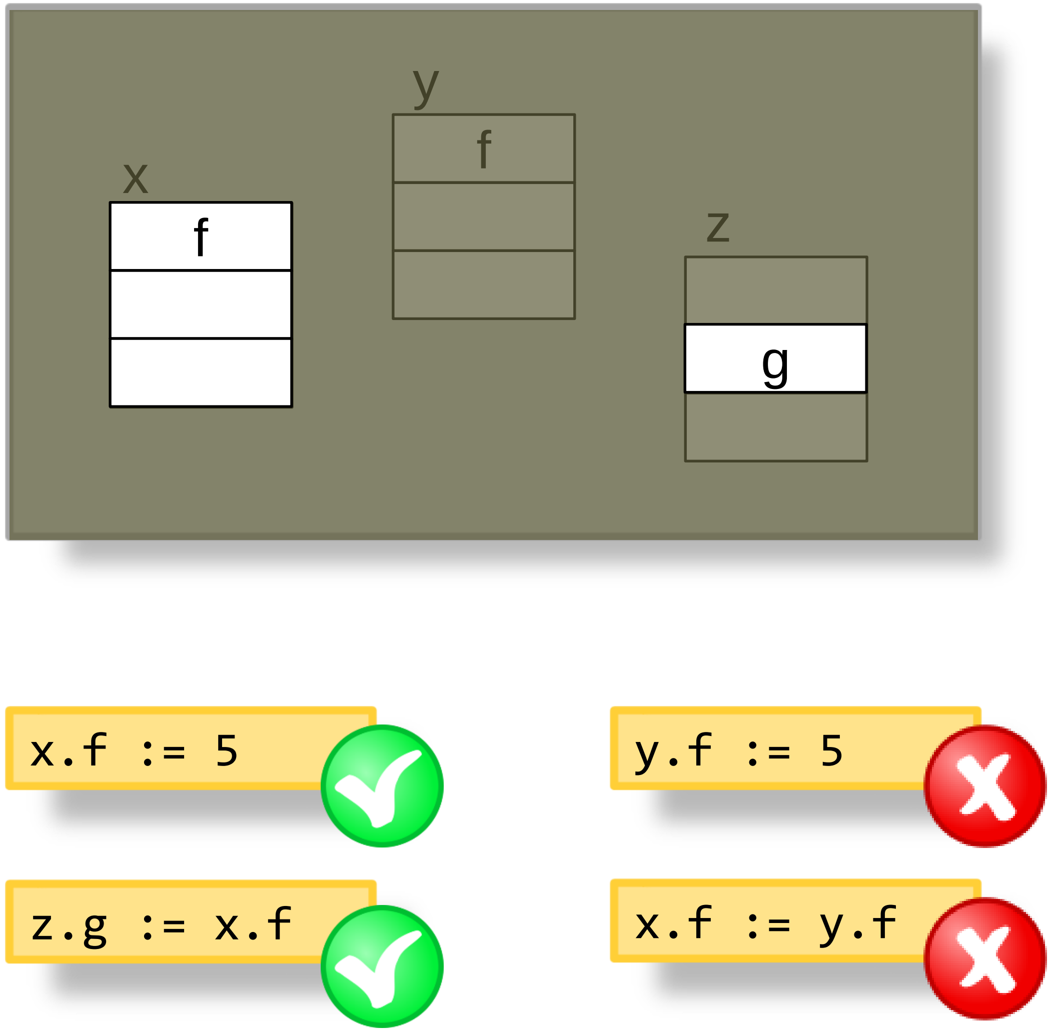
```
field val: Int

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

- A heap maps object-field pairs to values
- 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
  - Conceptually, objects could remain allocated

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

Separation logic

- Separation logic denotes permissions by points-to predicates

```
p.f ↦ _
```

- Disjointness of permissions is expressed by separating conjunction

```
p.f ↦ _ * q.f ↦ _ ⇒ p ≠ q
```

Viper

- Viper's logic uses access predicates
  - Access predicates are not permitted under negations, disjunctions, and on the left of implications

```
acc(p.f)
```

- Viper's && acts like separating conjunction

```
acc(p.f) && acc(q.f) ⇒ p ≠ q
```

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)
```

Implicit dynamic frames

- Viper uses a variation of separation logic called implicit dynamic frames, which specify permissions and value constraints separately

- Assertions may contain both permissions and value constraints

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

```
∃v :: p.f ↦ v * v > 0
```

- Most 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
```

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

Implicit dynamic frames: example

```
method swap(a: Ref, b: Ref)
  requires a.f ↦ v * b.f ↦ w
  ensures  a.f ↦ w * b.f ↦ v
{
  var tmp: Int
  tmp := a.f
  a.f := b.f
  b.f := tmp
}
```

```
method swap(a: Ref, b: Ref)
  requires acc(a.f) && acc(b.f)
  ensures  acc(a.f) && acc(b.f)
  ensures  a.f == old(b.f) && b.f == old(a.f)
{
  var tmp: Int
  tmp := a.f
  a.f := b.f
  b.f := tmp
}
```

- `old`-expressions are evaluated in the pre-state of a method
- Labeled `old`-expressions allow one to relate arbitrary states within a method

Exercise

Implement a method

```
method gauss(n: Int, res: Ref)
```

that sums up the first `n` natural numbers and stores the result in the `val`-field of reference `res`.

Tasks:

- Verify memory safety.
- Specify and verify functional correctness.
- Verify termination.

Hints:

- See template `Exercise1.vpr`.
- Use a while loop.
- Store intermediate results directly in `res.val`, not in a local.

Outline

- Separation logic proofs in Viper
  - Hoare-style verification
  - Permission-based reasoning
  - Abstraction
  - Advanced separation logic
- Viper as target language
- Conclusion

Predicates

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

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

- 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))
}
```

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))
}
```

```
method client(x: Ref, y: Ref)
  requires 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))
}
```

```
method client(x: Ref)
  requires list(x)
{
  x.next := null  // no permission
}
```

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

Folding and unfolding predicates

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

- An unfold statement exchanges a predicate instance for its body

```
method client(x: Ref)
  requires list(x)
{
  unfold list(x)
  x.next := null
}
```

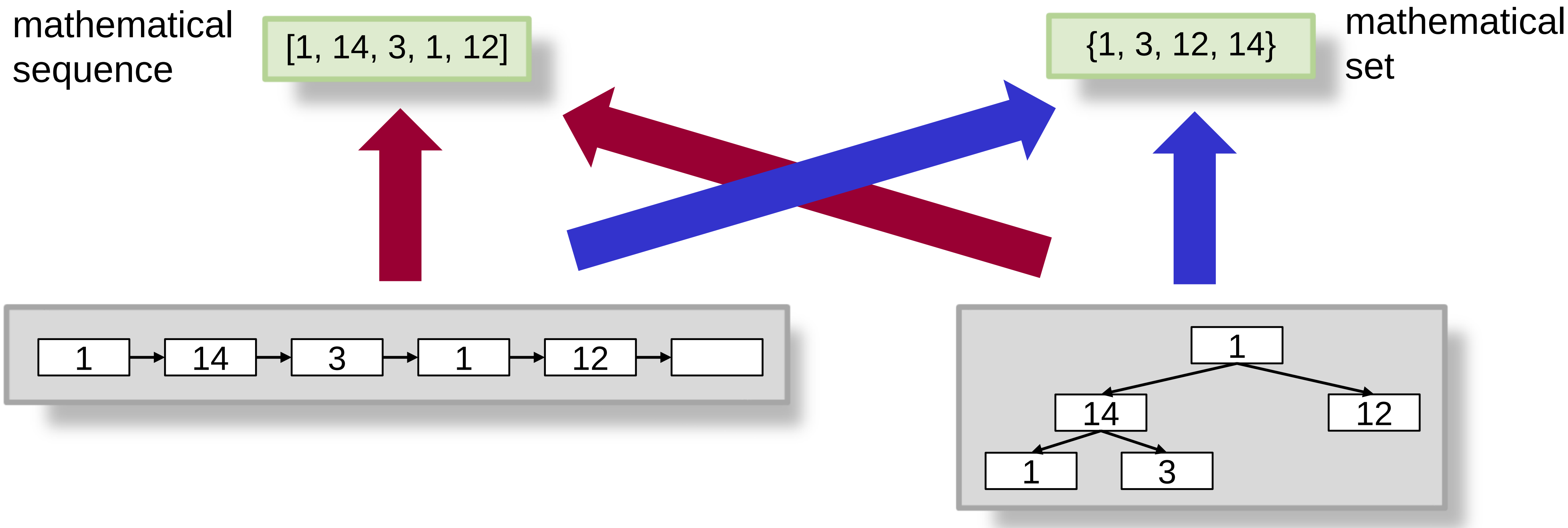
- A fold statement exchanges a predicate body for a predicate instance

```
method client(x: Ref)
  requires list(x)
  ensures  list(x)
{
  unfold list(x)
  x.next := null
  fold list(x)
}
```

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

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 and function calls are expressions

- Functions may be recursive

- Functions must have a precondition that frames the function body, that is, provides all permissions to evaluate the body

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

Idiomatic abstraction in Viper

```
predicate list(this: Ref, cont: Seq[Int]) {
  ∃e,n :: this.elem ↦ e * this.next ↦ n *
  (n == null ==> 0 == |cont|) *
  (n != null ==> ∃c ::
                 cont == Seq(e) ++ c *
                 list(n, c))
}
```

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

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

- Implicit dynamic frames specify permissions and value constraints separately and separate inputs (`this`) from outputs (`cont`)
  - Facilitates incremental specification and verification
  - Enables using deterministic, side-effect free code functions in specifications (e.g., equals)

Exercise

Separation logic often uses predicates for list segments, for instance, to describe cyclic lists.

Tasks:

- Define a predicate `lseg` for non-empty list segments.
- Define a function `segcont` that yields the sequence of integers stored in a list segment.
- Implement and verify a method

```
method single(e: Int) returns (res: Ref)
```

  that creates a cyclic list with one element, e.

Hint:

- See template `Exercise2.vpr`.

Outline

- Separation logic proofs in Viper
  - Hoare-style verification
  - Permission-based reasoning
  - Abstraction
  - Advanced separation logic
- Viper as target language
- Conclusion

Fractional permissions

- To distinguish read and write access, permissions can be split and re-combined
  - A permission amount π is a rational number in [0;1]

    `acc(x.f, π)`

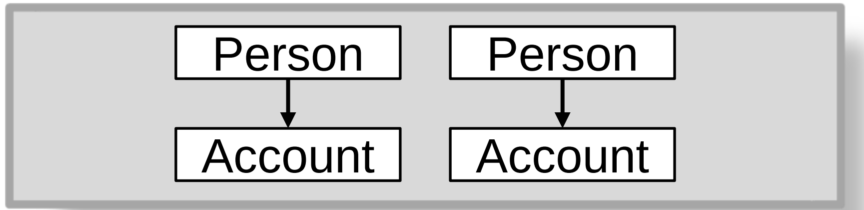
  - Viper syntax allows fractions `n/d`, as well as **`write`** for 1, **none** for 0, and **wildcard** for an arbitrary positive permission amount
  - `acc(E.f)` is a shorthand for `acc(E.f, write)`, and `P(E)` for `acc(P(E), write)`
- Field read requires some non-zero permission, field write requires full (write) permission
- Separating conjunction sums up permissions of the conjuncts

  `acc(x.f, 1/2) && acc(x.f, 1/2)` is equivalent to `acc(x.f, write)`

Sharing in data structures

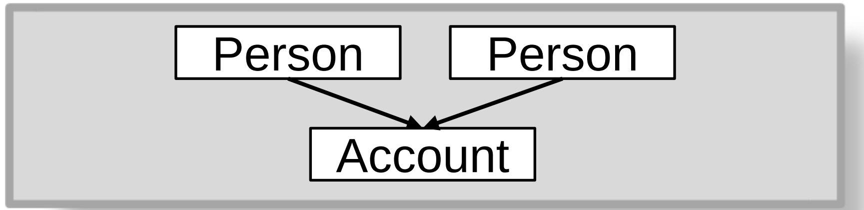
- Full permissions can describe tree-shaped data structures only

```
predicate person(this: Ref) {
  acc(this.savings) &&
  acc(this.savings.bal)      }
```



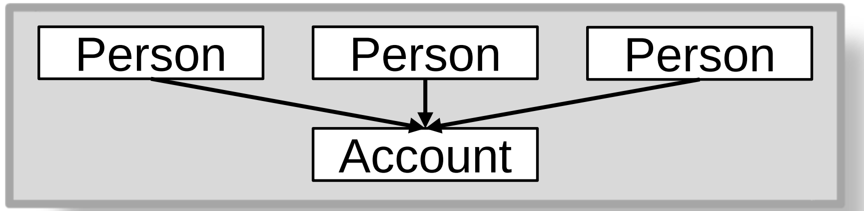
- Fractional permissions allow sharing

```
predicate person(this: Ref) {
  acc(this.savings) &&
  acc(this.savings.bal, 1/2) }
```



- including unbounded (immutable) sharing

```
predicate person(this: Ref) {
  acc(this.savings &&
  acc(this.savings.bal, wildcard)    }
```



Predicates and fractional permissions

- Predicates may contain fractions of permissions

```
predicate P(x: Ref)
{ acc(x.f, 1/2) }
```

- It is also possible to own fractions of a predicate instance

`acc(P(x), 1/2) && acc(P(x), 1/2)` is equivalent to `acc(P(x), write)`

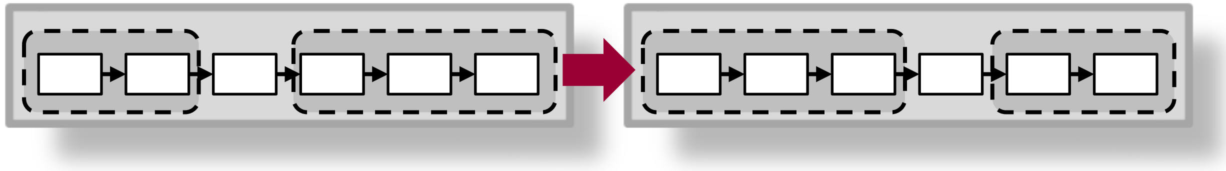
- Unfold and fold multiply the fraction of the predicate with the fractions in the predicate body

```
method client(x: Ref)
  requires acc(P(x), 1/4)
  ensures  acc(x.f, 1/8)
{
  unfold acc(P(x), 1/4)
}
```

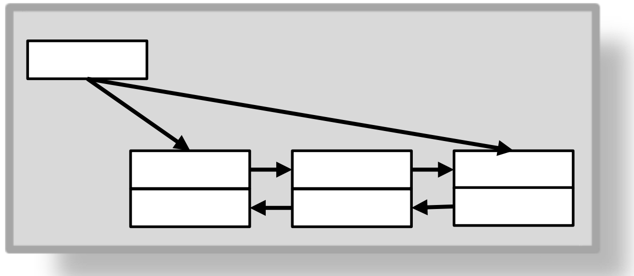
Limitations of recursive predicates

- Recursive predicates allow one to specify unbounded data structures
  - Traversals happen in the order in which the predicate needs to be unfolded

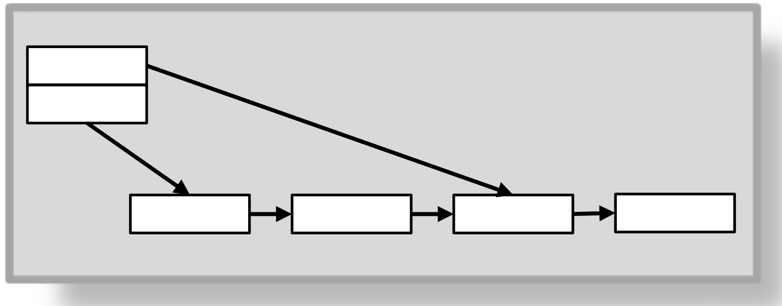
- Predicates are not ideal for many other use cases



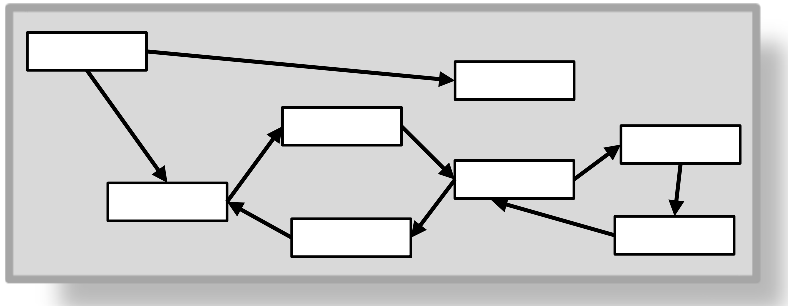
Iterative traversals



Other traversal orders



Random-access data structures



Arbitrary cyclic data structures

Quantified permissions

- To denote permission to an unbounded set of locations without prescribing a traversal order, we allow permissions and predicates to occur under universal quantifiers

```
forall x: T :: A
```

- Viper's `forall` quantifiers can be thought of as a possibly-infinite iterated separating conjunction

$$\texttt{forall x: T :: A} \equiv \texttt{A}[x_1/x] \;\&\&\; \texttt{A}[x_2/x] \;\&\&\; \ldots$$

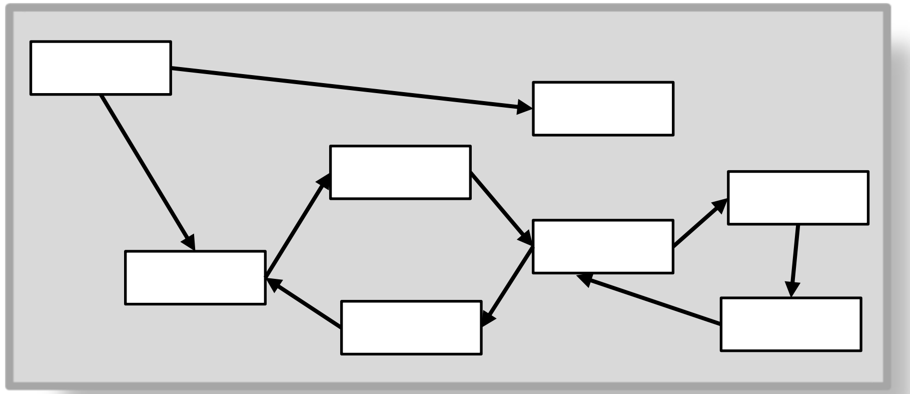
- Viper requires for each assertion **`acc`**`(E.f)` under a **`forall`** `x:T` that `E` is injective, that is:

$$x_1 \neq x_2 \Rightarrow E[x_1/x] \neq E[x_2/x]$$

  - The analogous rule applies to predicates (for parameter tuples)

Explicit footprints

- As alternative to predicates, we can specify permission to an unbounded set of locations by
  - Maintaining an explicit set of references as ghost state (the explicit footprint)
  - Quantifying over the set elements in specifications



- We represent a graph as a set of nodes, each node stores a (possibly empty) set of successors

```
field next: Set[Ref]

predicate graph(nodes: Set[Ref]) {
  forall n: Ref :: n in nodes ==> acc(n.next) && (n.next subset nodes)
}
```

- This idiom supports arbitrary traversals, random accesses, and arbitrary sharing

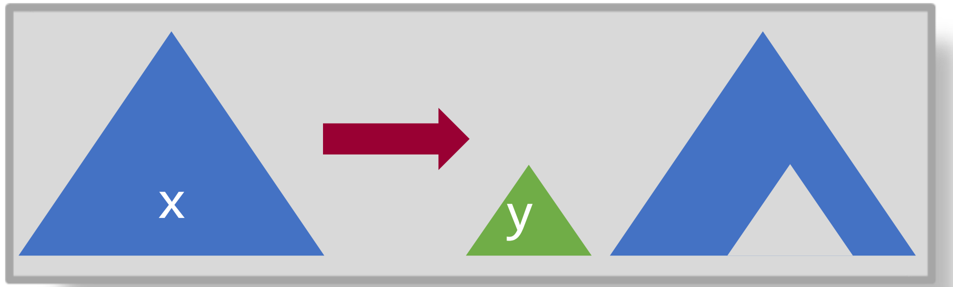
Partial data structures

```
field left: Ref
field right: Ref

predicate tree(x: Ref) {
  acc(x.left) && acc(x.right)
  && (x.left != null ==> tree(x.left))
  && (x.right != null ==> tree(x.right))
}
```

```
method getLeft(x:Ref) returns (y:Ref)
  requires tree(x)
  ensures tree(y) && ...
{
  y := x
  while (y.left != null)
    invariant tree(y) && ...
  {
    unfold tree(y)
    y := y.left
  }
}
```

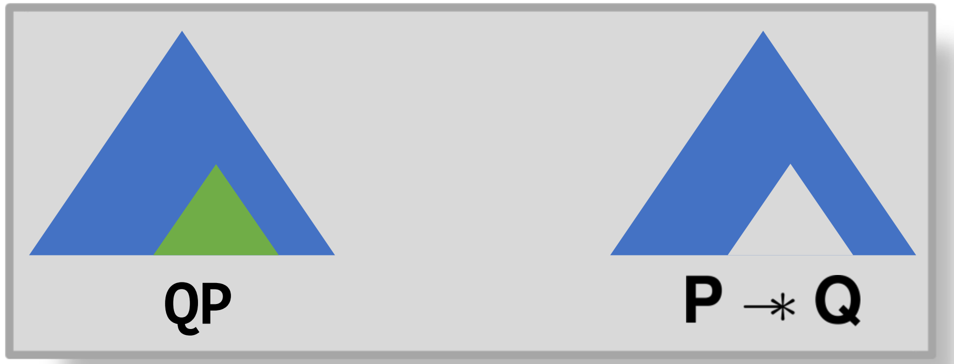
- To allow clients of `getLeft` to use the predicate `tree(x)` later, `getLeft` needs to return permissions to the rest of the tree



- Not ideal: define a dedicated predicate
  - Requires a way to identify the hole
  - Requires ghost code to plug the hole

Separating implication: magic wands

- A magic wand **P** $-\!\!*$ **Q** represents the difference between **Q** and **P**



- This allows us to specify our `getLeft` method

```
method getLeft(x: Ref) returns (y: Ref)
  requires tree(x)
  ensures  tree(y) && (tree(y) --* tree(x))
```

- Intuition: permissions are held by method executions, loop iterations, either directly or inside predicate instances or magic wands

Reasoning with magic wands

- Applying a magic wand
  - Viper has a designated statement to apply modus ponens for magic wands

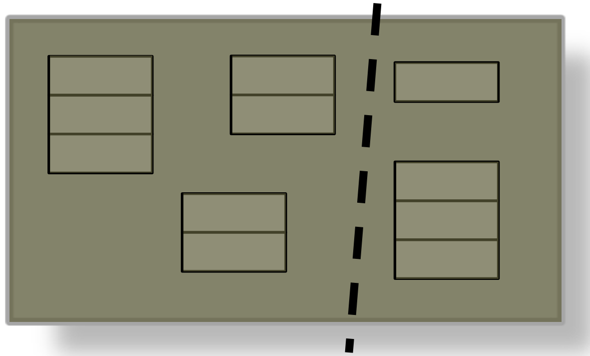
```
apply P --* Q
```

$$\mathbf{P} * (\mathbf{P} \mathrel{-\!\!*} \mathbf{Q}) \Rightarrow \mathbf{Q}$$

- Creating a magic wand
  - Viper needs to determine which permissions from the current state need to be moved into the wand such that the wand, together with **P**, yields **Q**

```
package P --* Q
```

split current state



such that

$$* \; \mathbf{P} \Rightarrow \mathbf{Q}$$

Example revisited

```
method getLeft(x: Ref) returns (y: Ref)
    requires tree(x)
    ensures tree(y) && (tree(y) --* tree(x))
{
   y := x
   package tree(x) --* tree(x)
   while (unfolding tree(y) in y.left != null)
      invariant tree(y) && (tree(y) --* tree(x))
   {
      unfold tree(y)
      var y_left: Ref := y.left
      package tree(y_left) --* tree(x)
      {
         fold tree(y)
         apply tree(y) --* tree(x)
      }
      y := y_left
   }
}
```

```
y := getLeft(x)
y.update() // requires tree(y)
apply tree(y) --* tree(x)
x.update() // requires tree(x)
```

The Viper language

Program code

- Sequential, imperative language
- Standard control structures
- Basic type system
- Built-in heap
- Explicit permission manipulation

Assertion language

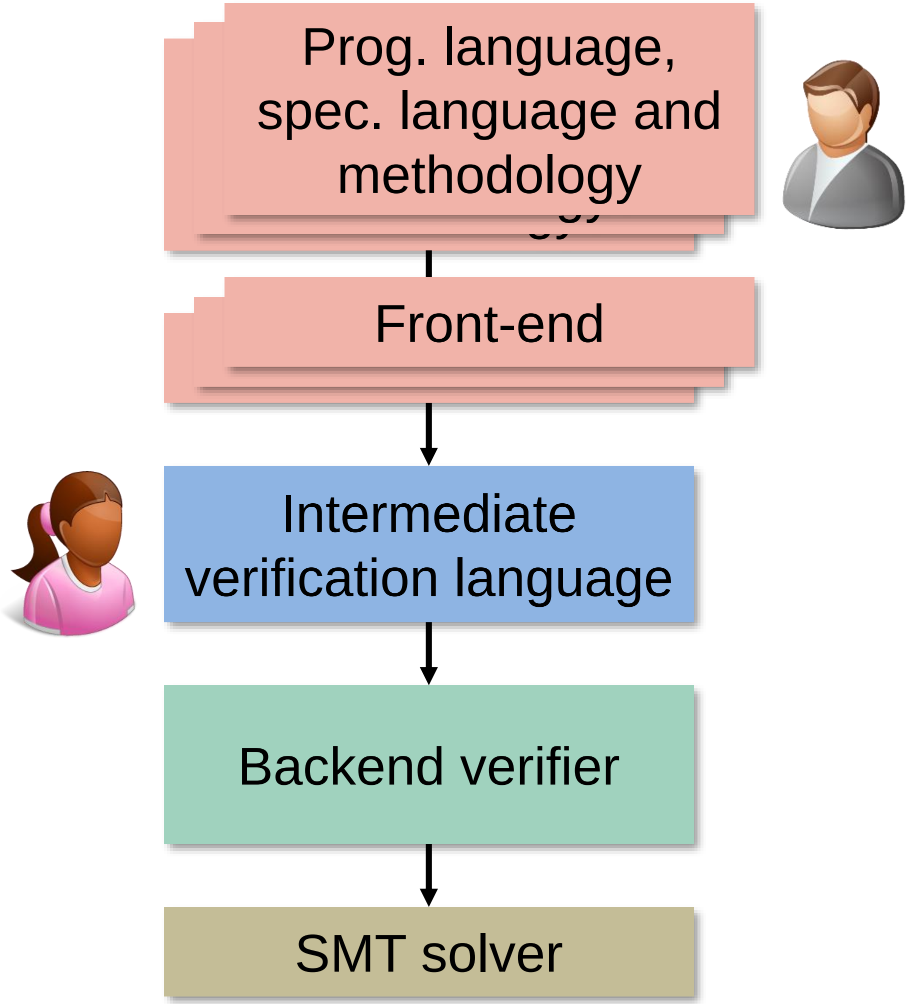
- Inductive predicates
- Abstraction functions
- Fractional permissions
- Iterated separating conjunction
- Magic wands

Verification

- Absence of run-time errors
- Memory safety
- User-provided assertions
- Termination

Mathematical theories

- Predefined datatypes
- User-defined datatypes
- Uninterpreted functions
- Axioms


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

gobra

Outline

- Separation logic proofs in Viper
- Viper as target language
  - Encoding source languages
  - Encoding program logics
- Conclusion

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
}
```

```
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
  tmp := 0

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

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

Arrays

- Viper does not have built-in arrays
- In contrast to sequences, arrays are mutable heap data structures
- We model arrays by a set of disjoint references that can be accessed via an index
- `loc(a, i).val` models `a[i]`
- More-dimensional arrays can be encoded analogously

```
field val: Int // for integer arrays

domain Array {
  function loc(a: Array, i: Int): Ref
  function len(a: Array): Int
  function first(r: Ref): Array
  function second(r: Ref): Int

  axiom injectivity {
    forall a: Array, i: Int :: {loc(a, i)}
      first(loc(a, i)) == a &&
      second(loc(a, i)) == i
  }

  axiom length_nonneg {
    forall a: Array :: len(a) >= 0
  }
}
```

Accessing array locations

- Arrays are random-access data structures

- We can express permissions using quantified permissions

```
forall i: Int :: 0 <= i < len(a) ==> acc(loc(a, i).val)
```

  - Similarly for sub-ranges of the array

- We define macros for convenient access

```
define lookup(a, i)
  loc(a, i).val
```

```
define update(a, i, e) {
  loc(a, i).val := e
}
```

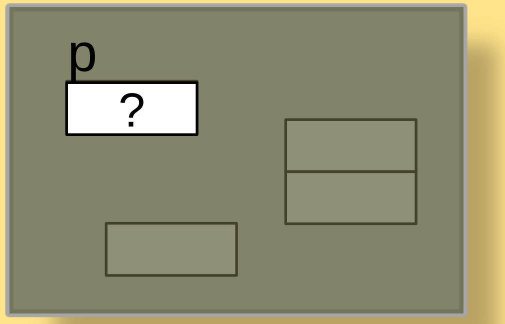
  - Bounds are checked implicitly via permissions

Outline

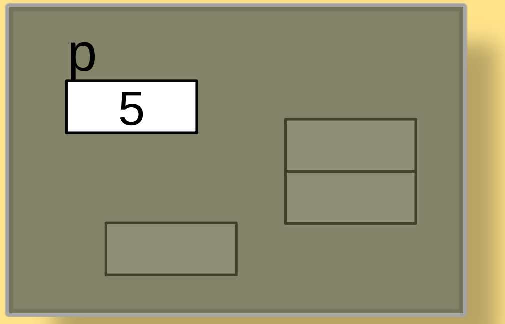
- Separation logic proofs in Viper
- Viper as target language
  - Encoding source languages
  - Encoding program logics
- Conclusion

Permission transfer

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

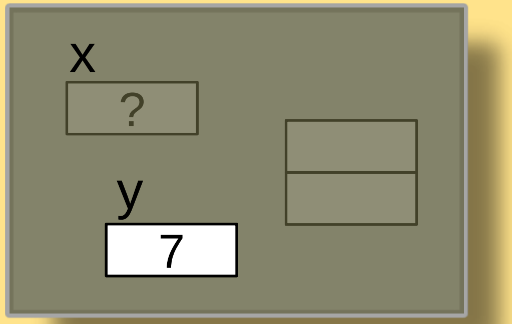



```
  p.f := v
```




```
}
```

```
method client(x: Ref, y: Ref)
  requires acc(x.f) && acc(y.f)
  requires x.f == 2 && y.f == 7 {
```




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

Framing!

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

Permission transfer for method calls

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

- 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

Permission transfer for loops and concurrency

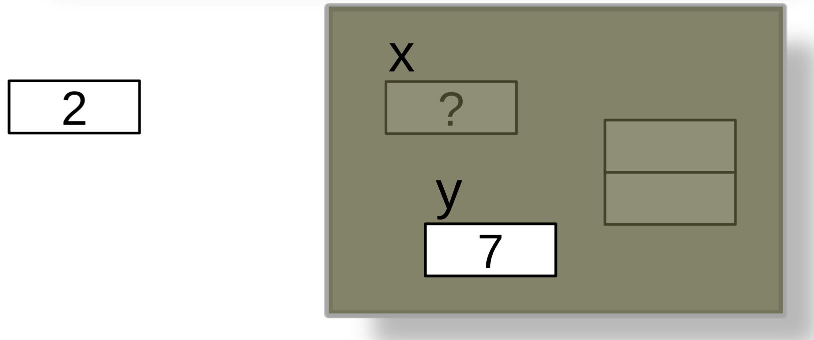
$$\dfrac{\dfrac{\{\ \boxed{\mathtt{acc}(\mathtt{x.f})} * \mathtt{x.f} < 10\ \}\ \ \mathtt{x.f := x.f + 1}\ \ \{\ \boxed{\mathtt{acc}(\mathtt{x.f})}\ \}}{\{\ \mathtt{acc}(\mathtt{x.f})\ \}\ \ \mathtt{while}(\mathtt{x.f < 10})\ \{\ \ldots\ \}\ \ \{\ \mathtt{acc}(\mathtt{x.f}) * \neg \mathtt{x.f} < 10\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

$$\dfrac{\dfrac{\{\ \boxed{\mathbf{A}_1}\ \}\ \mathtt{S}_1\ \{\ \boxed{\mathbf{B}_1}\ \} \qquad \{\ \boxed{\mathbf{A}_2}\ \}\ \mathtt{S}_2\ \{\ \boxed{\mathbf{B}_2}\ \}}{\{\ \mathbf{A}_1 * \mathbf{A}_2\ \}\ \mathtt{S}_1 \parallel \mathtt{S}_2\ \{\ \mathbf{B}_1 * \mathbf{B}_2\ \}}}{\{\ \boxed{\mathbf{A}_1} * \boxed{\mathbf{A}_2} * \boxed{\mathbf{C}}\ \}\ \mathtt{S}_1 \parallel \mathtt{S}_2\ \{\ \boxed{\mathbf{B}_1} * \boxed{\mathbf{B}_2} * \boxed{\mathbf{C}}\ \}}$$

Permission transfer: inhale and exhale operations

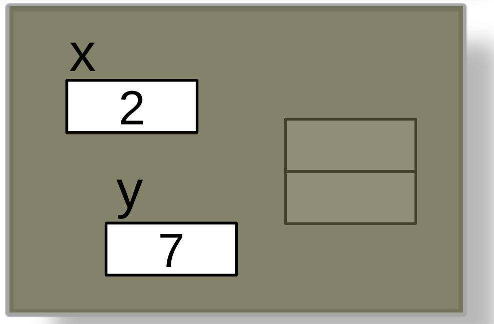
- `inhale A` means:
  - obtain all permissions required by assertion **A**
  - assume all logical constraints

```
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 and globals
  - 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`**

Encoding structured parallelism

- The proof rule employs the familiar permission transfer

$$\dfrac{\dfrac{\{\ \mathbf{A}_1\ \}\ \ S_1\ \ \{\ \mathbf{B}_1\ \} \qquad \{\ \mathbf{A}_2\ \}\ \ S_2\ \ \{\ \mathbf{B}_2\ \}}{\{\ \mathbf{A}_1 * \mathbf{A}_2\ \}\ \ S_1 \parallel S_2\ \ \{\ \mathbf{B}_1 * \mathbf{B}_2\ \}}}{\{\ \mathbf{A}_1 * \mathbf{A}_2 * \mathbf{C}\ \}\ \ S_1 \parallel S_2\ \ \{\ \mathbf{B}_1 * \mathbf{B}_2 * \mathbf{C}\ \}}$$

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

```
method S1(…) 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

Encoding locks

$$\frac{}{\{\,\mathbf{R}\,\}\ \mathtt{x := new\ Lock()}\ \{\,\mathtt{isLock(x,}\mathbf{R}\mathtt{)}\,\}}$$

$$\frac{}{\{\,\mathtt{isLock(x,}\mathbf{R}\mathtt{)}\,\}\ \mathtt{acquire\ x}\ \{\,\mathtt{locked(x,}\mathbf{R}\mathtt{)} * \mathbf{R}\,\}}$$

$$\frac{}{\{\,\mathtt{locked(x,}\mathbf{R}\mathtt{)} * \mathbf{R}\,\}\ \mathtt{release\ x}\ \{\,\mathtt{isLock(x,}\mathbf{R}\mathtt{)}\,\}}$$

$$\forall x :: \mathtt{isLock(x,}\mathbf{R}\mathtt{)} \Leftrightarrow \mathtt{isLock(x,}\mathbf{R}\mathtt{)} * \mathtt{isLock(x,}\mathbf{R}\mathtt{)}$$

Encode custom resources as abstract predicates

```
predicate isLock(x: Ref, …)

predicate locked(x: Ref, …)
```

Encode duplicable resources as arbitrary positive fractions

```
acc(isLock(x, …), wildcard)
```

Encoding lock invariants

$$\frac{}{\{\, \mathbf{R} \,\}\ x := \mathbf{new}\ \mathtt{Lock}()\ \{\, \mathtt{isLock}(x, \mathbf{R}) \,\}}$$

$$\frac{}{\{\, \mathtt{isLock}(x, \mathbf{R}) \,\}\ \mathbf{acquire}\ x\ \{\, \mathtt{locked}(x, \mathbf{R}) * \mathbf{R} \,\}}$$

$$\frac{}{\{\, \mathtt{locked}(x, \mathbf{R}) * \mathbf{R} \,\}\ \mathbf{release}\ x\ \{\, \mathtt{isLock}(x, \mathbf{R}) \,\}}$$

$$\forall x :: \mathtt{isLock}(x, \mathbf{R}) \Leftrightarrow \mathtt{isLock}(x, \mathbf{R}) * \mathtt{isLock}(x, \mathbf{R})$$

- Specify invariant in a program annotation

```
share x inv R
```

- Apply defunctionalization
  - Assign unique constant c to each invariant
  - Parameterize predicates

```
predicate isLock(x: Ref, inv: Int)

predicate locked(x: Ref, inv: Int)
```

  - Declare macro or predicate for all invariants

```
define Inv(x, n) (
  (n == c ==> R) &&
  …
)
```

Encoding lock operations

$$\frac{}{\{\, \mathbf{R} \,\}\ \ x := \textbf{new}\ \texttt{Lock}()\ \ \{\, \texttt{isLock}(x, \mathbf{R}) \,\}}$$

$$\frac{}{\{\, \texttt{isLock}(x, \mathbf{R}) \,\}\ \ \textbf{acquire}\ x\ \ \{\, \texttt{locked}(x, \mathbf{R}) * \mathbf{R} \,\}}$$

$$\frac{}{\{\, \texttt{locked}(x, \mathbf{R}) * \mathbf{R} \,\}\ \ \textbf{release}\ x\ \ \{\, \texttt{isLock}(x, \mathbf{R}) \,\}}$$

$$\forall x :: \texttt{isLock}(x, \mathbf{R}) \Leftrightarrow \texttt{isLock}(x, \mathbf{R}) * \texttt{isLock}(x, \mathbf{R})$$

```
exhale Inv(x,c)
inhale acc(isLock(x, c), wildcard)
```

```
exhale acc(isLock(x, c), wildcard)
inhale locked(x, c) && Inv(x,c)
```

```
exhale locked(x, c) && Inv(x,c)
inhale acc(isLock(x, c), wildcard)
```

Exercise

The following rules are adapted from Relaxed Separation Logic.

$$\frac{}{\{\mathsf{true}\}\ l := \mathtt{alloc}_{\mathsf{na}}()\ \{\mathsf{Uninit}(l)\}}$$

$$\frac{}{\{l \overset{1}{\mapsto} \_ \vee \mathsf{Uninit}(l)\}\ [l]_{\mathsf{na}} := e\ \{l \overset{1}{\mapsto} e\}}$$

$$\frac{}{\{l \overset{k}{\mapsto} e\}\ x := [l]_{\mathsf{na}}\ \{x = e * l \overset{k}{\mapsto} e\}}$$

$$(l \overset{k}{\mapsto} e \ * \ l \overset{k'}{\mapsto} e') \Leftrightarrow (e = e' * l \overset{k+k'}{\mapsto} e)$$

Encode them in Viper and verify the client code on the right.

- Assume that l is an integer location.
- Recall that Viper does not support disjunction of impure assertions.

Hints:

- See template `Exercise3.vpr`.
- Track initialization in the state.

```
method sum(p, q)
  requires p ½↦ v * q ½↦ w
  ensures  p ½↦ v * q ½↦ w
  ensures  result == v + w
{
  a := [p]
  b := [q]
  return a + b
}

method client() {
  x := alloc()
  y := alloc()
  [x] := 2
  [y] := 3
  s := sum(x, y)
  assert s == 5
}
```

Scope of existing Viper encodings

Language features

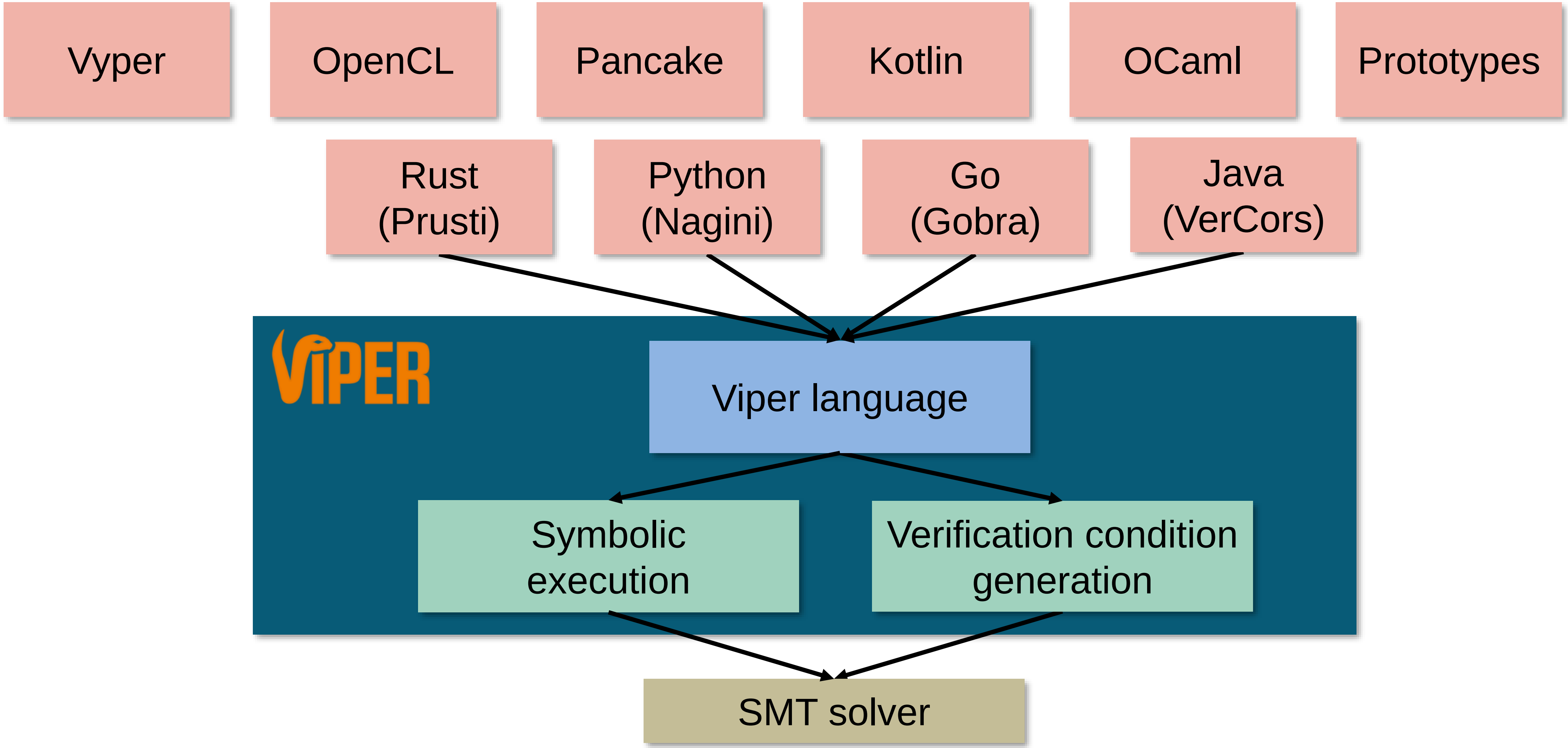
- Imperative code
- Object-oriented code
- Nominal and structural typing
- Closures
- Multithreading with shared state and message passing
- Weak-memory concurrency

Properties

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

Outline

- Separation logic proofs in Viper
- Viper as target language
- Conclusion


Vyper
OpenCL
Pancake
Kotlin
OCaml
Prototypes
Rust (Prusti)
Python (Nagini)
Go (Gobra)
Java (VerCors)
VIPER
Viper language
Symbolic execution
Verification condition generation
SMT solver

Main limitations

Inherited from SMT solver

- First-order logic
- Undecidable theories may lead to spurious errors
- Verification time for large methods

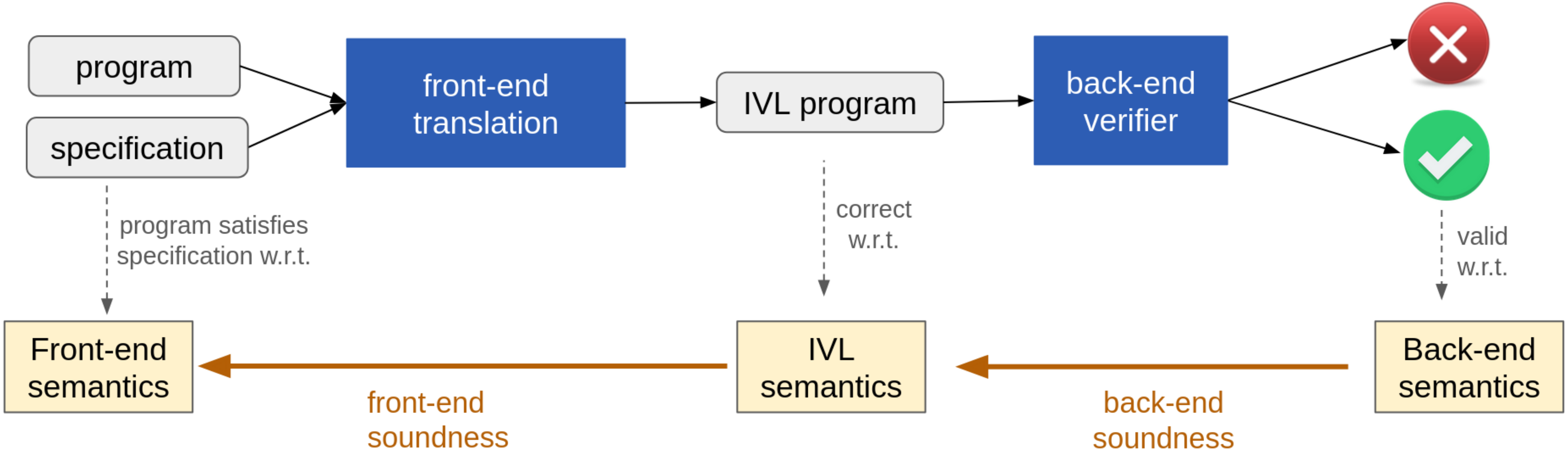
Annotation overhead

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

Trust assumptions

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

Formal Foundations for Translational Separation Logic Verifiers



Talk on Wednesday at 5pm in Peek-A-Boo!

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