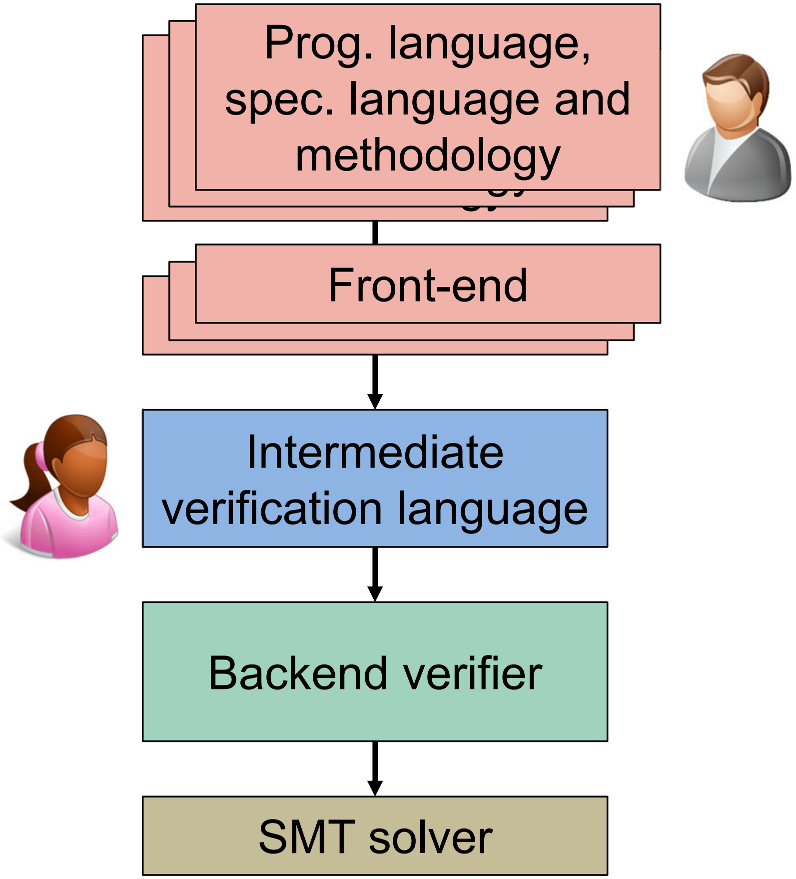
Peter Müller

OWNERSHIP IN PROGRAM VERIFICATION – FROM SEPARATION LOGIC TO RUST AND BACK

ETH zürich


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

Outline

- The Viper intermediate verification language
  - Reasoning about the heap
  - Abstraction
  - Advanced separation logic in Viper
- Verification of Rust programs
  - Ownership
  - References
- Conclusion

VIPER

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

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 explicit de-allocation (garbage collector)
  - Conceptually, objects could remain allocated

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

Aliasing

```
field val: Int

method foo(p: Ref, q: Ref)
  requires p != null && q != null
  ensures  p.val == 5
{
  p.val := 5
  q.val := 7
}
```

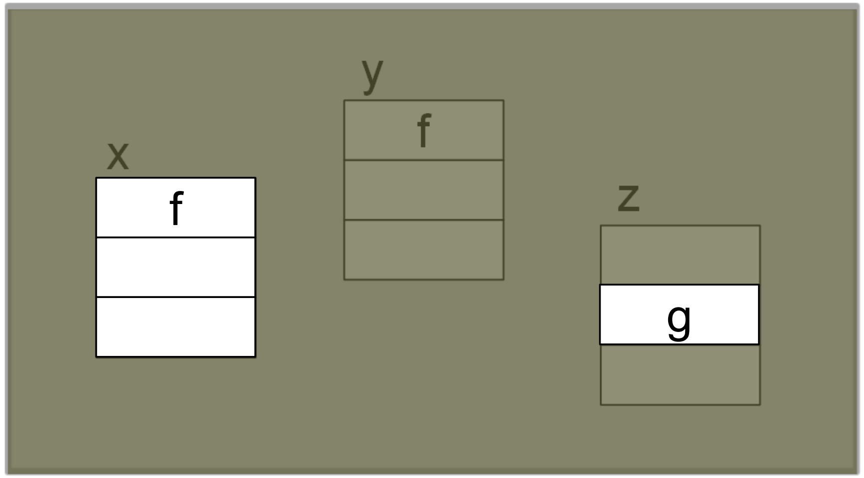
- Bad idea: require programmer to explicitly specify all non-aliasing properties
  - Does not work for unbounded data structures
  - Violates information hiding

Further challenges of heap verification

- Writing specifications that preserve information hiding
- Concurrency, e.g., data races

Access permissions

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



`x.f := 5` ✓

`y.f := 5` ✗

`z.g := x.f` ✓

`x.f := y.f` ✗

Permission assertions

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

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

$\mathtt{p.f} \mapsto \_$

- Assertions may contain both permissions and value constraints

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

$\exists V \cdot \mathtt{p.f} \mapsto V \wedge V > 0$

- 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
```

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** do not contain access predicates

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

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

- 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)
```

Framing

Frame rule

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

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 then **C** is preserved
  - Otherwise, the precondition of the conclusion 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
}
```

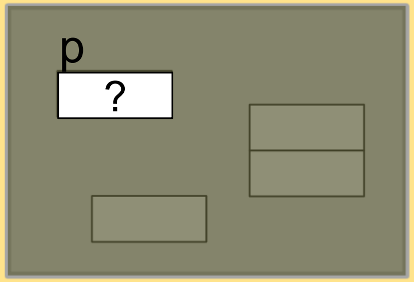
```
// assume we have acc(x.f) && acc(y.f)
assume y.f == 7
set(x, 5)
assert x.f == 5 && y.f == 7
```

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

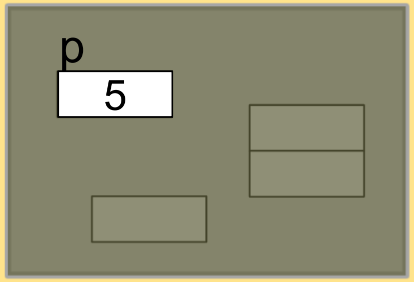
- A method modifies at most the heap locations to which it has permission

Permission transfer

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

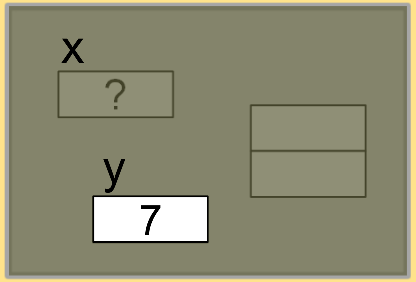



```
  p.f := v
```




```
}
```

```
// 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

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

- 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

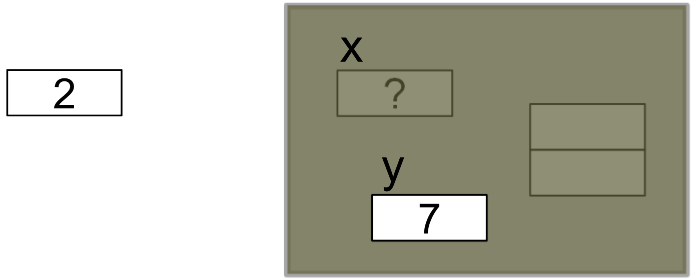
```
// assume we have acc(x.f) && acc(y.f)
x.f := 0
y.f := 7
while (x.f < 10)
  invariant acc(x.f)
{
  x.f := x.f + 1
}
assert y.f == 7
```

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

Permission transfer: inhale and exhale operations

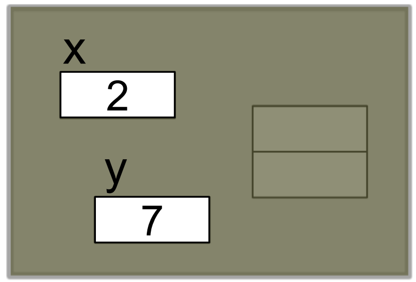
- **`inhale A`** means:
  - obtain all permissions required by assertion **A**
  - assume all logical constraints

```
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
```



Simplified proof obligations for 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)
```

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 yet

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

- Permissions are an excellent basis

Outline

- The Viper intermediate verification language
  - Reasoning about the heap
  - Abstraction
  - Advanced separation logic in Viper
- Verification of Rust programs
  - Ownership
  - References
- Conclusion

Limitations of technique as introduced so far

```
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(elem, next)
    n.next := null
    this.elem := e
    this.next := n
  } else {
    append(this.next, e)
  }
}
```

- Specification reveals implementation details
- Neither permissions nor behavior can be expressed

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))
}
```

```
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 fold and unfold statements in the program

- 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)
```

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

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

```
method head(this: Ref) returns (res: Int)
  requires list(this)
  ensures  list(this)
  ensures  res ==
           old(unfolding list(this) in this.elem)
```

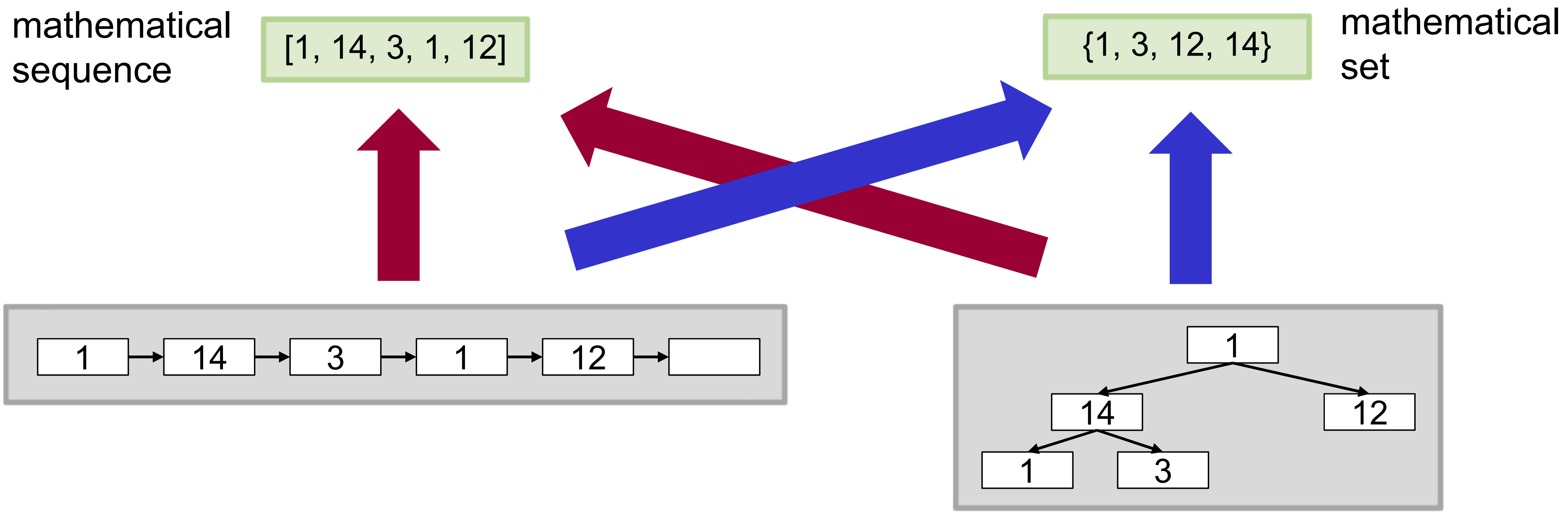
Challenges revisited

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

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

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

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

- The Viper intermediate verification language
  - Reasoning about the heap
  - Abstraction
  - Advanced separation logic in Viper

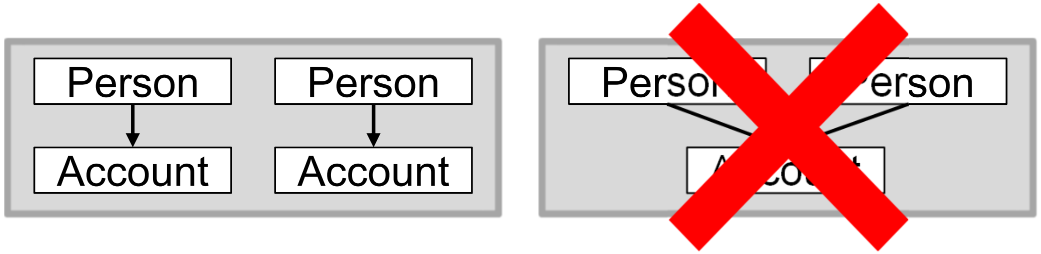
- Verification of Rust programs
  - Ownership
  - References

- Conclusion

Limitations of technique as introduced so far

```
field val: Int

method clone(this: Ref) returns (res: Ref)
  requires acc(this.val)  // read only
  ensures  acc(this.val) && acc(res.val)
  ensures  this.val == old(this.val)
{
  res := new(val)
  res.val := this.val
}
```

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



- To enable framing, methods that only read a data structure must specify that each abstraction remains unchanged
- Sharing in data structures cannot be expressed (all structures are trees)

Fractional permissions

- To distinguish read and write access, permissions can be split and re-combined
  - The permission amount π is a rational number in [0;1]
  - Viper syntax allows fractions `n/d`, **`write`** for 1, **`none`** for 0, and **`wildcard`** for existentially-quantified positive amounts
  - **`acc`**`(E.f)` is a shorthand for **`acc`**`(E.f, `**`write`**`)`, and `P(E)` for **`acc`**`(P(E), `**`write`**`)`

```
acc(p.f, π)
```

```
acc(P(e), π)
```

- Field read requires some non-zero permission

```
inhale acc(x.f, 1/2)
v := x.f
```

- Field write requires full (write) permission

```
inhale acc(x.f, 1/2)
x.f := v
```

Manipulating fractional permissions

- 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)`

- Inhale adds permissions
- Exhale subtracts permissions and havocs only when *all* permission to a location or predicate instance is removed
- Values are framed as long as *some* permission is held

```
method clone(this: Ref) returns (res: Ref)
  requires acc(this.val, 1/2)  // read only
  ensures  acc(this.val, 1/2) && acc(res.val)
{ … }
```

```
method demo(this: Ref) returns (c: Ref)
  requires acc(this.val)
{
  var tmp: Int
  tmp := this.val
  c := clone(this)  // no havoc of this.val
  assert tmp == this.val
}
```

Predicates and fractional permissions

- Predicates may contain fractional permissions, for instance, to permit sharing

```
predicate readCell(this: Ref) {
  acc(this.cell) && acc(this.cell.val, 1/2)
}
```

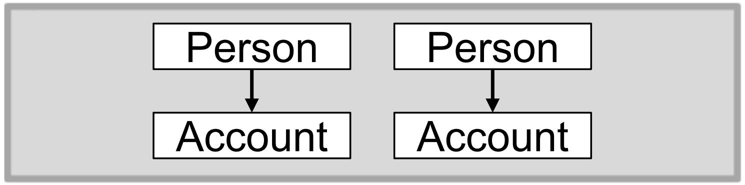
- Unfold and fold multiply the fraction of the predicate with the fractions in the predicate body

```
inhale acc(readCell(x), 1/4)
unfold acc(readCell(x), 1/4)
exhale acc(x.cell.val, 1/8)
```

Sharing in data structures

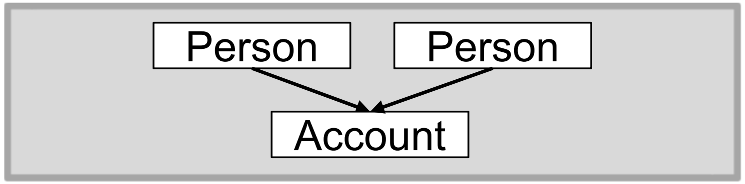
- Full permissions can describe tree-shaped data structures only

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



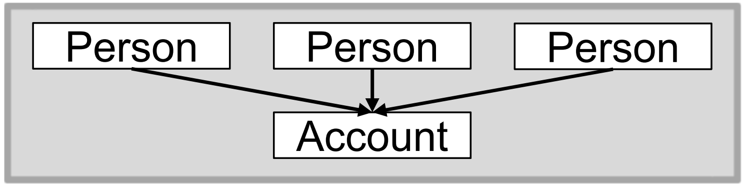
- Fractional permissions allow sharing

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



- including unbounded (immutable) sharing

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



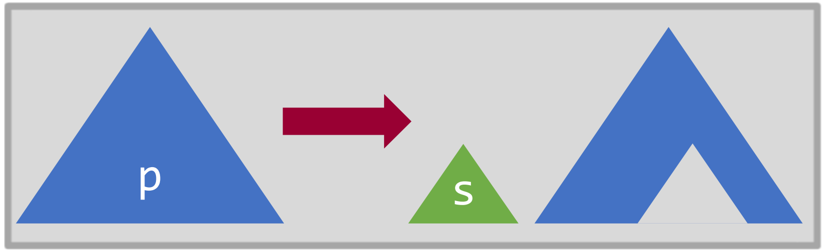
Partial data structures

```
field savings: Ref

predicate account(this: Ref)

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

```
method get(p: Ref) returns (s: Ref)
  requires person(p)
  ensures  account(s)
{
  unfold person(p)
  s := p.savings
}
```

```
s := get(p)
s.deposit(100)
p.move() // requires person(p)
```

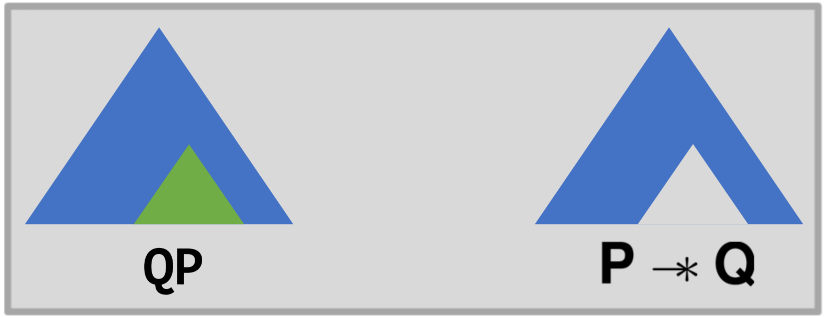
- To allow clients of `get` to use the person object later, `get` needs to return permissions to the remainder of the object



- Bad idea: return permissions to fields explicitly
  - Violates information hiding
  - Does not work for unbounded structures
- Bad idea: 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 `get` method

```
method get(p: Ref) returns (s: Ref)
  requires person(p)
  ensures  account(s) && (account(s) --* person(p))
```

- Intuition: permissions are held by method executions, loop iterations, 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
```

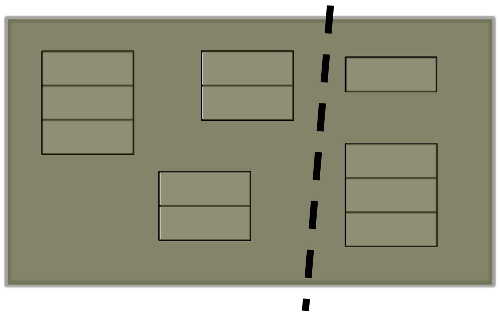
$$\mathbf{P} * (\mathbf{P} \mathrel{-\!\!*} \mathbf{Q}) \Rightarrow \mathbf{Q}$$

- 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

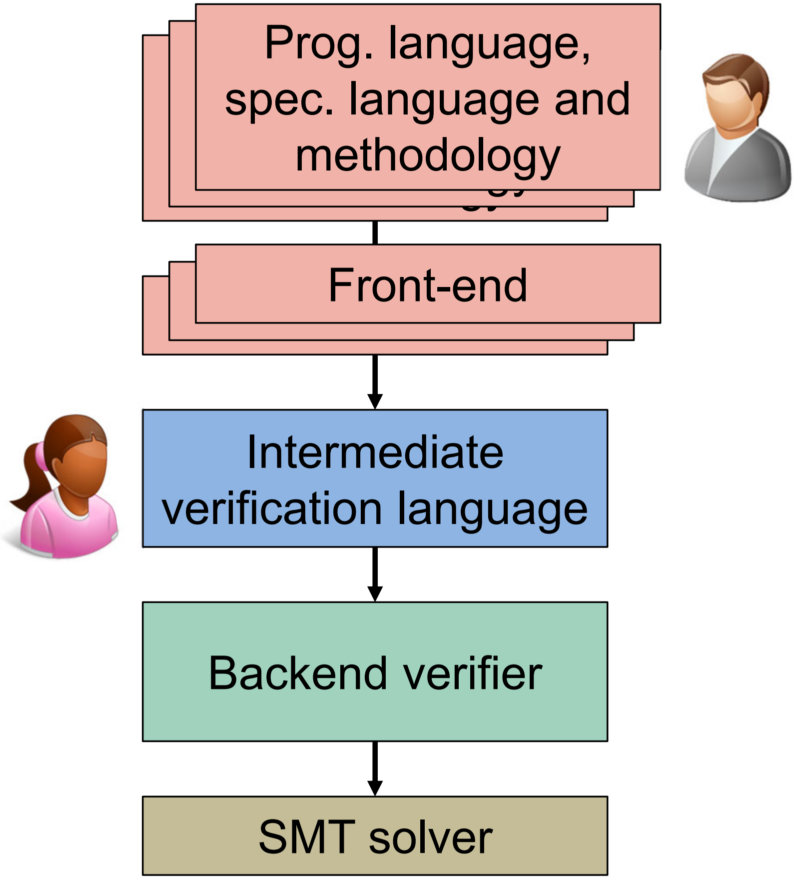


$$* \mathbf{P} \Rightarrow \mathbf{Q}$$

Example revisited

```
method get(p: Ref) returns (s: Ref)
  requires person(p)
  ensures  account(s) && (account(s) --* person(p))
{
  unfold person(p)
  s := p.savings
  package account(s) --* person(p) {
    fold person(p)
  }
}
```

```
s := get(p)
s.deposit(100)
apply account(s) --* person(p)
p.move() // requires person(p)
```



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

gobra

VIPER

Go verification in Gobra

```
requires isList(a) && isList(b)
func client(a *list, b *list) {
  tmp := b.length()
  a.appendBack(100)
  assert b.length() == tmp
}
```

Functional properties
Memory errors
Aliasing
Data races

- To handle these challenges, Gobra uses a verification technique similar to Viper's
- Programmers get exposed to the verification logic
- The overhead is substantial (both amount and complexity of annotations)

Rust and its type system

```rust
fn client(a: &mut List, b: &mut List)
{
  let tmp = b.length();
  a.appendBack(100);
  assert!(b.length() == tmp);
}
```

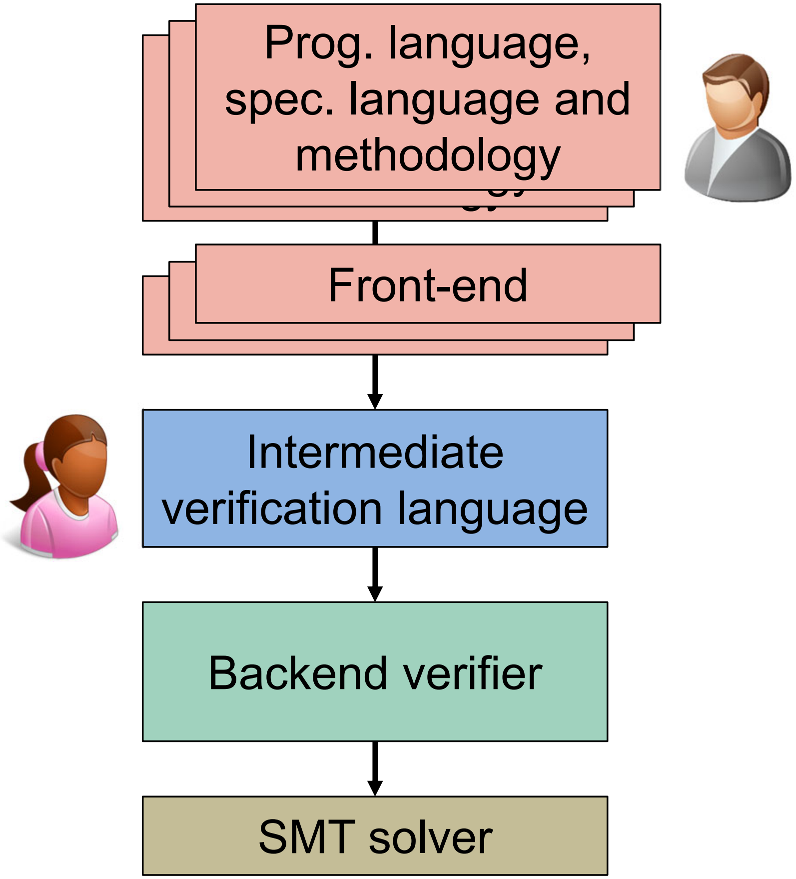
Functional properties

No memory errors

Controlled aliasing

No data races

Can we leverage this type system to simplify verification?



Prusti

VIPER

Outline

- The Viper intermediate verification language
  - Reasoning about the heap
  - Abstraction
  - Advanced separation logic in Viper
- Verification of Rust programs
  - Ownership
  - References
- Conclusion

Ownership in Rust

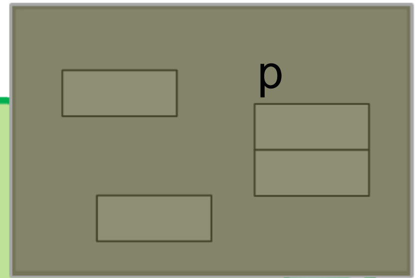
```
struct Pair {
  first: i32,
  second: i32,
}

fn demo(mut p: Pair, mut q: Pair) {
  p.first = 5;
  q.first = 7;
  assert!(p.first == 5)
}
```

- Rust supports ownership via its type system
- An expression representing a memory location (a place) of a type T owns the value in that location
- Owned values cannot be aliased
- Rust leverages this guarantee:
  - To ensure data race freedom
  - To perform automatic memory management (without a garbage collector)

Move assignments

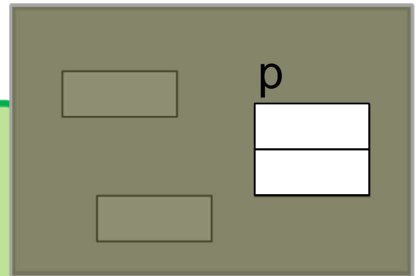
- When reading from a place, ownership of the read value is moved to the receiving place
  - Except for types that implement the `Copy` trait, such as **`i32`**

```
fn foo(mut p: Pair) {
  let mut q = p;
  q.first = 5;
}
```



- When a value is moved out of a place, the place becomes unusable until a new value is assigned

```
fn bar(mut p: Pair) {
  foo(p);
  p.second = 7;
}
```



From Rust ownership to permissions

- The notions of ownership in Rust and separation logic are extremely similar

"Rust ownership ≈ write permission"

Exclusive ownership
No access through aliases
Ownership transfer upon calls and returns

- Approach: generate a memory safety proof in Viper automatically from Rust types

Rust		Viper
Types	→	Predicates
Signatures		Core method specs
Compiler analyses		Auxiliary annotations

Viper encoding: primitive types

- We model values of primitive types as objects with one field (so that we can later take a reference to the memory location)

```
field val: Int

predicate i32(this: Ref) {
  acc(this.val)
}
```

- Auxiliary operations yield the integer value and create the object for an integer

```
function getInt(this: Ref): Int
  requires i32(this)
{
  unfolding i32(this) in this.val
}
```

```
method makeInt(x: Int) returns (res: Ref)
  ensures i32(res)
  ensures getInt(res) == x
{
  res := new(val);
  res.val := x
  fold i32(res)
}
```

Viper encoding: user-defined types

- The predicate for a struct includes the predicate instances for its fields because ownership in Rust is deep

```
struct Pair {
  first: i32,
  second: i32,
}
```

```
field first: Ref
field second: Ref

predicate Pair(this: Ref) {
  acc(this.first) && i32(this.first) &&
  acc(this.second) && i32(this.second)
}
```

Viper encoding: methods

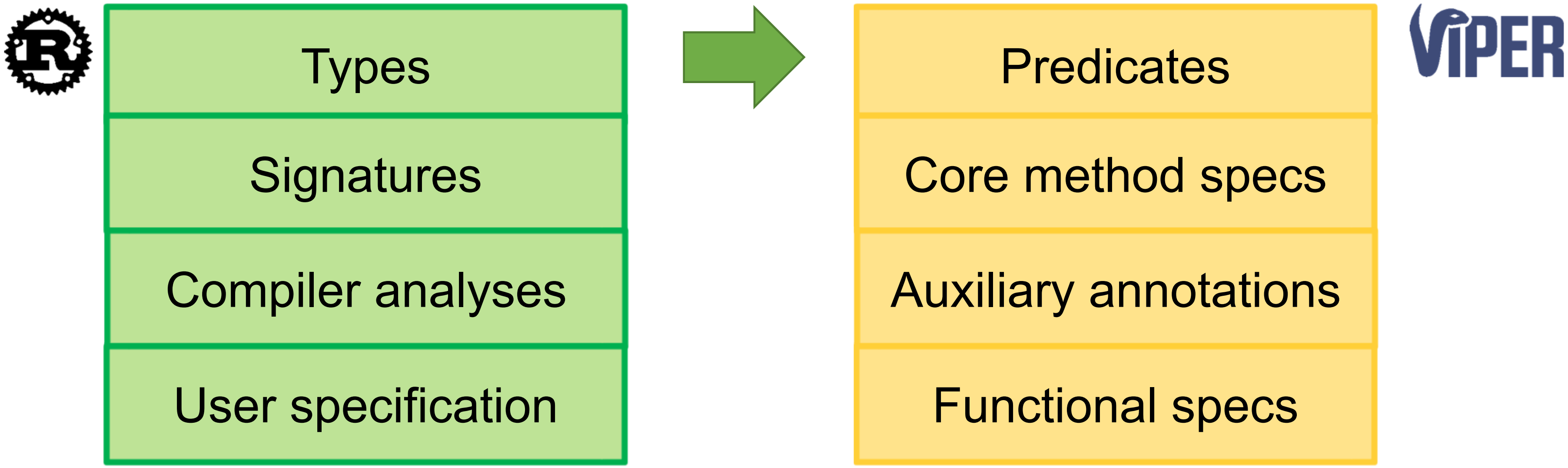
```
fn demo(mut p: Pair, mut q: Pair) {
  p.first = 5;
  q.first = 7;
  assert!(p.first == 5)
}
```

```
method demo(p: Ref, q: Ref)
  requires Pair(p) && Pair(q)
{
  unfold Pair(p)
  p.first := makeInt(5)
  unfold Pair(q)
  q.first := makeInt(7)
  fold Pair(q)
  assert getInt(p.first) == 5
  fold Pair(p)
}
```

- The encoding generates all annotations necessary to verify the method in Viper

Functional correctness

- The automatically-generated core proof re-proves memory safety
- The contained permission specifications enable reasoning about aliasing and framing (and race freedom in concurrent code)
- Functional specifications can be layered on top easily



Viper encoding: functions and contracts

```
#[pure]
fn max(a: i32, b: i32) -> i32 {
    if a > b { a } else { b }
}
```

```
function max(a: Ref, b: Ref): Ref
  requires i32(a) && i32(b)
{ getInt(a) > getInt(b) ? a : b }
```

```
#[ensures(result.first == old(max(p.first, q.first)))]
fn pmax(p: Pair, q: Pair) -> Pair
{ … }
```

```
method pmax(p: Ref, q: Ref) returns (res: Ref)
  requires Pair(p) && Pair(q)
  ensures  Pair(res)
  ensures  unfolding Pair(res) in
           getInt(res.first) ==
           old(unfolding Pair(p) in
               unfolding Pair(q) in
               getInt(max(p.first, q.first)))
{ … }
```

Outline

- The Viper intermediate verification language
  - Reasoning about the heap
  - Abstraction
  - Advanced separation logic in Viper
- Verification of Rust programs
  - Ownership
  - References
- Conclusion

Borrowing

- With move assignments, functions need to return their arguments in order to transfer ownership back to the caller

```
fn swap(mut p: Pair) -> Pair {
  let tmp = p.first;
  p.first = p.second;
  p.second = tmp;
  p
}
```

```
let mut p = Pair{ first: 5, second: 7 };
p = swap(p);
println!("{}", p.first);
```

- To avoid transferring ownership back and forth, Rust supports borrowing, which provides temporary access to a value through a reference

```
fn swap(p: &mut Pair) {
  let tmp = p.first;
  p.first = p.second;
  p.second = tmp;
}
```

```
let mut p = Pair{ first: 5, second: 7 };
swap(&mut p);
println!("{}", p.first);
```

Borrow checking

- Borrowing creates a temporary alias

- The borrowed-from place is unusable as long as the borrow exists

Borrow for q is created
Type error
p is unusable
Borrow for q expires
p is usable again

```
let mut p = Pair{ first: 5, second: 7 };
let q = &mut p;
swap(&mut p);
q.first = 9;
swap(&mut p);
```

- The Rust compiler uses a static analysis to determine when borrows expire

- Borrow checking across functions may require lifetime annotations (not discussed here)

Viper encoding: references

```
struct Pair {
  first: i32,
  second: i32,
}
```

```
field ref: Ref

predicate RefMutPair(this: Ref) {
  acc(this.ref) && Pair(this.ref)
}
```

```
let mut p;
p = Pair{ first: 5, second: 7 };
let q = &mut p;
q.first = 9;
// borrow for q expires
swap(&mut p);
```

```
…
q := new(ref)
q.ref := p
fold RefMutPair(q)
…
unfold RefMutPair(q)
unfold Pair(q.ref)
q.ref.first := makeInt(9)
fold Pair(q.ref)
fold RefMutPair(q)
…
unfold RefMutPair(q)
…
```

Functional correctness

- The automatically-generated core proof re-proves memory safety
- The contained permission specifications enable reasoning about aliasing and framing (and race freedom in concurrent code)
- Functional specifications can be layered on top easily

```
#[ensures(p.first == old(p.second))]
#[ensures(p.second == old(p.first))]
fn swap(p: &mut Pair) {
  let tmp = p.first;
  p.first = p.second;
  p.second = tmp;
}
```

Reborrowing

- Reborrowing occurs when a function borrows from a borrowed value

```
fn getFirst(p: &mut Pair) -> &mut i32 {
  &mut p.first
}
```

```
method getFirst(p: Ref) returns (res: Ref)
  requires RefMutPair(p)
  ensures  RefMuti32(res)
{ … }
```

Naïve encoding does not return permissions to the remainder of the `Pair p`

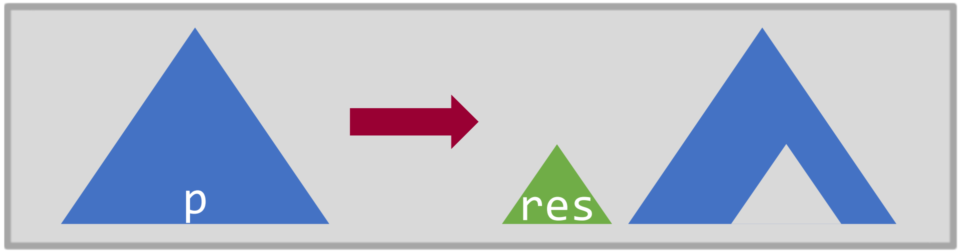
```
fn demo(mut p: Pair) {
  p.second = 0;
  let f = getFirst(&mut p);
  *f = *f + 1;
  // borrow for f expires
  assert!(p.second == 0);
}
```

How to re-construct `RefMutPair(p)`?

Viper encoding: reborrowing

- Reborrowing creates a partial data structure

```
fn getFirst(p: &mut Pair) -> &mut i32 {
  &mut p.first
}
```



- Partial data structures can be described using magic wands

```
method getFirst(p: Ref) returns (res: Ref)
  requires RefMutPair(p)
  ensures  RefMuti32(res)
  ensures  RefMuti32(res) --* RefMutPair(p)
{ … }
```

- Creating a reborrow packages a magic wand, expiring a reborrow applies it

Functional correctness: pledges

```
#[after_expiry(p.second == old(p.second))]
fn getFirst(p: &mut Pair) -> &mut i32 {
  &mut p.first
}
```

```
fn demo(mut p: Pair) {
  p.second = 0;
  let f = getFirst(&mut p);
  *f = *f + 1;
  // borrow for f expires
  assert!(p.second == 0);
}
```

- Lender function could modify its argument before creating the reborrow
- Postconditions must not access unusable borrowed-from value
  - All assertions follow Rust's type rules
- Pledges specify properties that will be true when the reborrow expires
- Here, getFirst can predict the value of p.second at expiry because the caller cannot modify this unusable place before the reborrow expires

Fun with pledges

```
#[ensures(*result == old(p.first))]
#[after_expiry(p.first ==
                before_expiry(*result))]
fn getFirst(p: &mut Pair) -> &mut i32 {
  &mut p.first
}
```

```
fn demo(mut p: Pair) {
  p.first = 0;
  let f = getFirst(&mut p);
  *f = *f + 1;
  // borrow for f expires
  assert!(p.first == 1);
}
```

- This pledge does not allow the client to modify the (usable) reborrowed value
- The postcondition can specify side effects on the reborrowed result
- Pledges specify properties of the borrowed-from value parametric in the reborrowed value

Viper encoding: pledges

- Pledges are conjoined to the rhs of the magic wand for the reborrow
  - Analogous to other functional specifications

```
#[after_expiry(p.second == old(p.second))]
fn getFirst(p: &mut Pair) -> &mut i32 {
  &mut p.first
}
```

```
method getFirst(p: Ref) returns (res: Ref)
  requires RefMutPair(p)
  ensures  RefMuti32(res)
  ensures  RefMuti32(res) --* RefMutPair(p) &&
           p.ref.second == old(p.ref.second)
{ … }
```

- When the reborrow expires, applying the wand provides the properties from the pledge

Shared references

- Rust distinguishes between exclusive, mutable references and shared, immutable references
  - Allows programmers to express design intent
  - Enables race-free concurrent reads

```
fn getFirst(p: &Pair) -> &i32 {
  &p.first
}
```

- Shared borrowing creates a temporary alias

- The borrowed-from place remains usable, but immutable as long as a shared borrow exists

Shared borrow for q1 is created

Creating another shared reference

p remains usable, but immutable

Type error: q1 is immutable

p is mutable

Borrows expire again

```
let mut p = Pair{ first: 5, second: 7 };
let q1 = &p;
let q2 = &p;
let v = p.first;
q1.first = 9;
swap(&mut p);
```

Aliasing XOR mutability

```
fn getFirst(p: &Pair) -> &i32 {
  &p.first
}
```

```
fn demo(mut p: Pair) {
  p.first = 0;
  let f = getFirst(&p);
  // borrow for f expires
  assert!(p.first == 0);
}
```

- Rust guarantees that, in each state, there is either at most one usable place that can access a value, or the value is immutable
- This guarantee allows us to frame values automatically
- No value can change while a shared reference exists

Viper encoding: shared references

- We reuse the type encoding for mutable references

```
struct Pair {
  first: i32,
  second: i32,
}
```

```
field ref: Ref

predicate RefMutPair(this: Ref) {
  acc(this.ref) && Pair(this.ref)
}
```

- and use fractional permissions to express shared access
  - read is a symbolic positive permission amount

```
fn isSame(p: &Pair) -> bool {
  p.first == p.second
}
```

```
method isSame(p: Ref) returns (res: Bool)
  requires acc(RefMutPair(p), read)
  ensures  pBool(res)
{ … }
```

Viper encoding: creating and expiring shared references

```
let mut p;
p = Pair{ first: 5, second: 7 };
let q = &p;
let v = q.first;
// borrow for q expires
swap(&mut p);
```

```
…
q := new()
exhale acc(Pair(p), write-read)
inhale acc(RefMutPair(q), read)
assume unfolding acc(RefMutPair(q), read) in
  q.ref == p
…
v := unfolding acc(RefMutPair(q), read) in
     unfolding acc(Pair(q.ref), read) in
     q.ref.first
…
exhale acc(RefMutPair(q), read)
inhale acc(Pair(p), write-read)
…
```

- This encoding trusts the borrow checker to determine when all shared borrows have expired, which simplifies permission accounting

Comparison of annotation overhead

```rust
#![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)
      }
    }
  }
```

```rust
  #[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)
}
```

A heap-free encoding: purification

- The behavior of a Rust function can be described entirely via its parameter and result values
- This suggests an alternative encoding (purification) that does not use the heap

```
struct Pair {
  first: i32,
  second: i32,
}
```

```
adt Pair {
  MkPair(first: Int, second: Int)
}
```

```
#[ensures(p.first == old(p.second))]
#[ensures(p.second == old(p.first))]
fn swap(p: &mut Pair) {
  let tmp = p.first;
  p.first = p.second;
  p.second = tmp;
}
```

```
method swap(p: Pair) returns (res: Pair)
  ensures res.first == p.second
  ensures res.second == p.first
{
  res := p
  var tmp: Int := res.first
  res := MkPair(res.second, res.second)
  res := MkPair(res.first, tmp)
}
```

Purification

- Purification simplifies the verification problem substantially
  - No heap and, thus, no need for permissions, predicates, magic wands
  - Much faster verification times (Creusot and Verus verifiers)
  - But treatment of reborrows is not trivial (based on prophecy variables)

- However, Rust allows unsafe code to work around the limitations of the strict ownership type system
  - Unsafe code may introduce aliasing among mutable pointers

```
let mut p;
p = Pair{ first: 5, second: 7 };
let ptr1: *mut i32 = &mut p.first;
let ptr2 = ptr1;
unsafe { *ptr1 = *ptr1 + 1; }
unsafe { *ptr2 = *ptr2 + 1; }
println!("{}", p.first);
```

- `ptr1` and `ptr2` and mutable aliases

- Verification requires a logic that can handle general heaps

Summary: 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;
}
```

- 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)

Outline

- The Viper intermediate verification language
  - Reasoning about the heap
  - Abstraction
  - Advanced separation logic in Viper
- Verification of Rust programs
  - Ownership
  - References
- Conclusion

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

Ownership in program verification

Ownership types
[Clarke, Potter, Noble ‘98]

Universe types
[Müller et al. ‘99]

Ownership-based verification
[Müller ‘01]

Separation Logic
[O’Hearn, Reynolds, Yang ‘01]

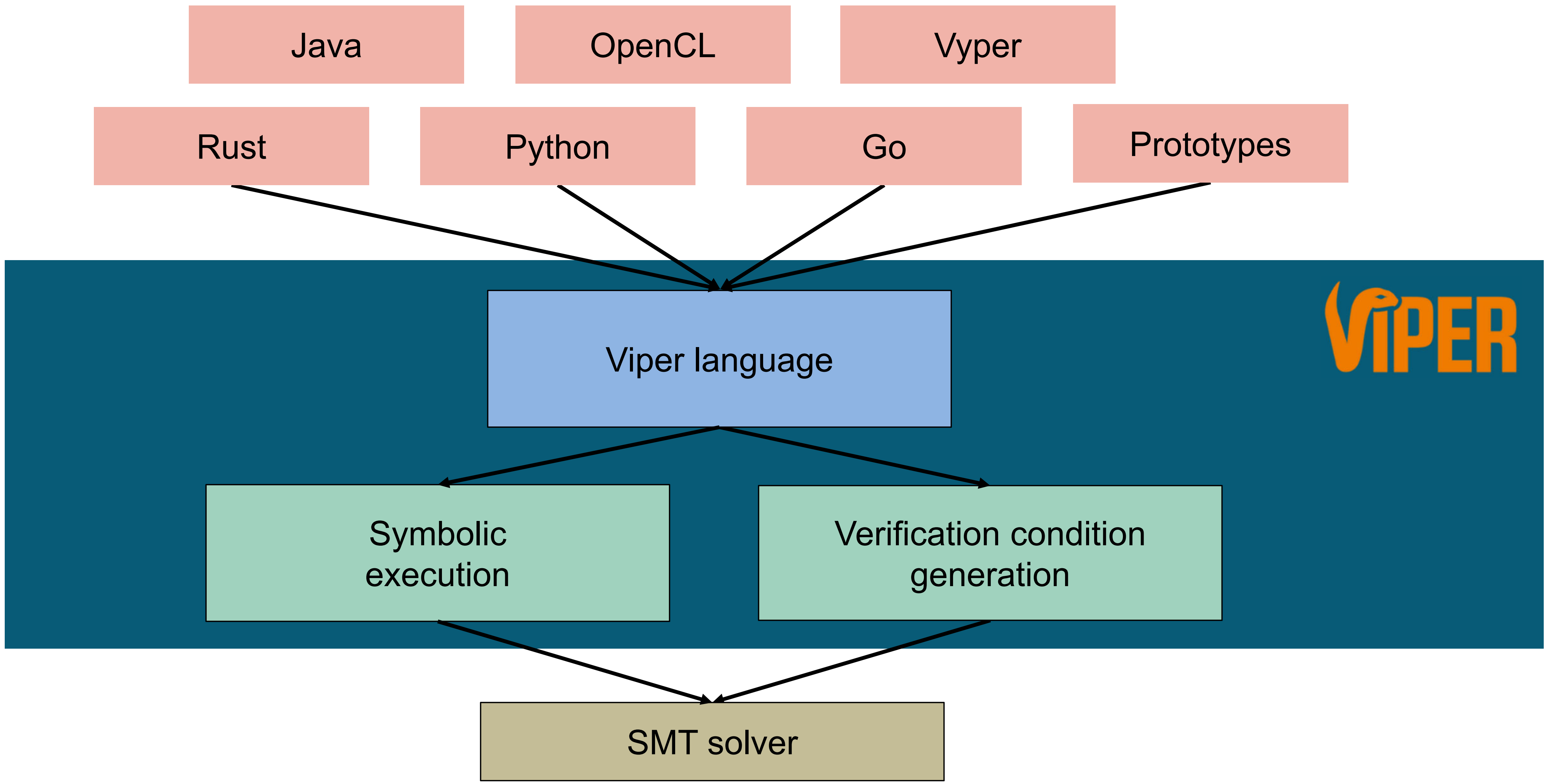
Spec# [Leino, Müller ‘04]

Rust [Matsakis, Klock ‘14]

Prusti [Astrauskas et al. ‘19]
Creusot [Denis et al. ‘22]
Verus [Latuada et al. ‘23]

Move [Blackshear et al. ‘19]

Move Prover [Zhong et al. ‘20]


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