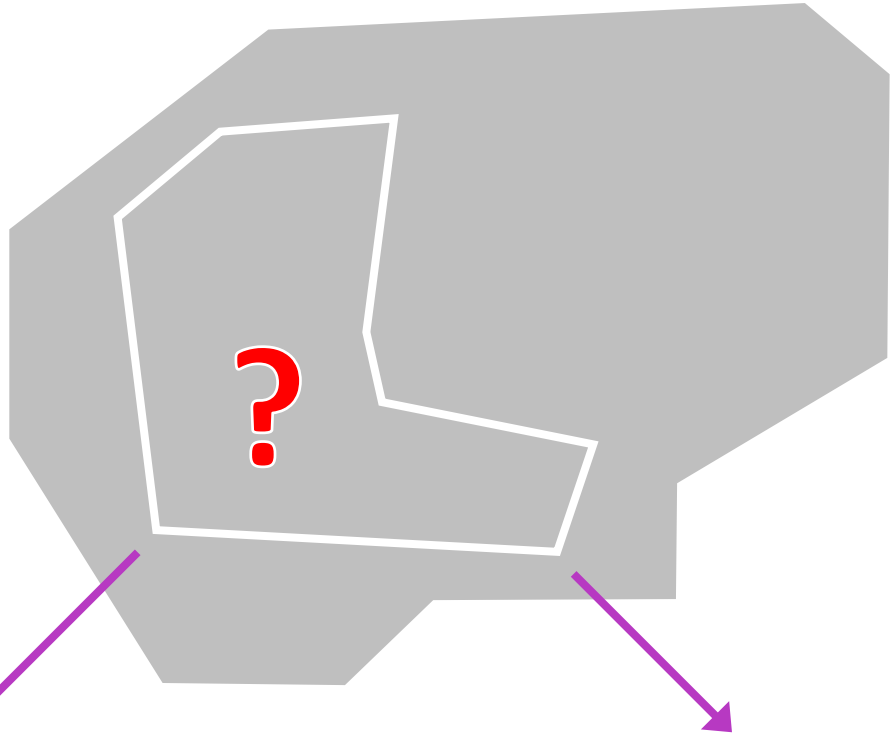
Automatic Verification of Iterated Separating Conjunctions using Symbolic Execution

ETH zürich

Peter Müller, **Malte Schwerhoff** and Alexander J. Summers

21st July 2016, Toronto

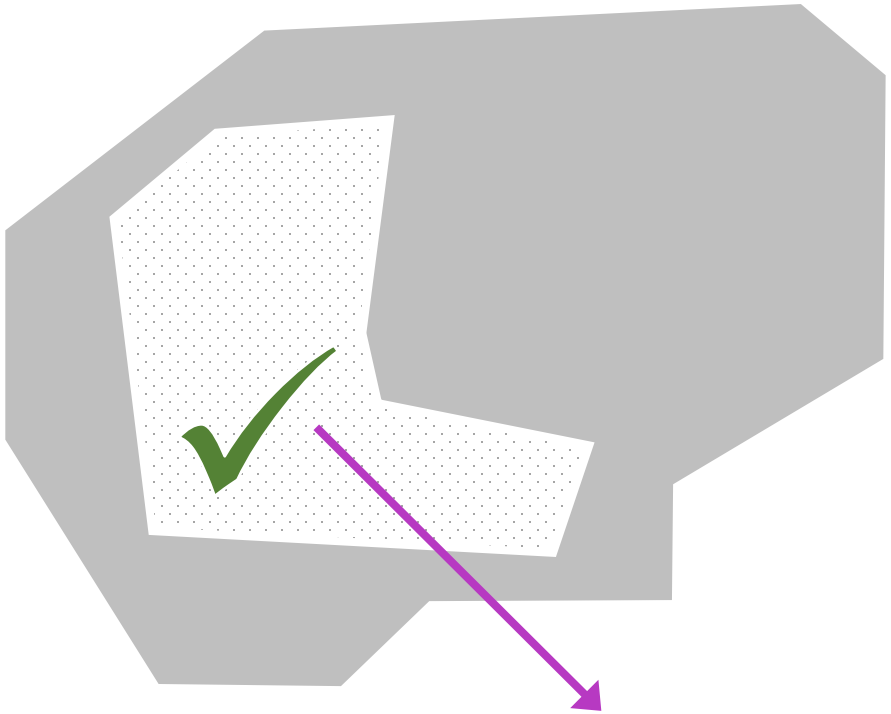
Modular Static Verification of Concurrent Programs



accessed by other thread?

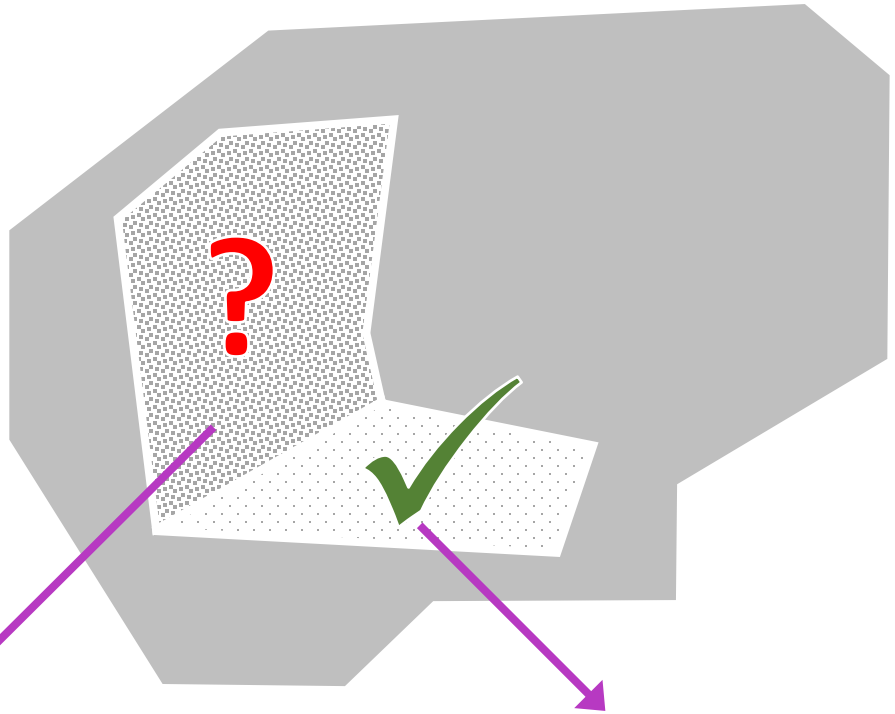
accessed by thread 1

Access Permissions (≈ Separation Logic)



exclusive
access permission
held by thread 1

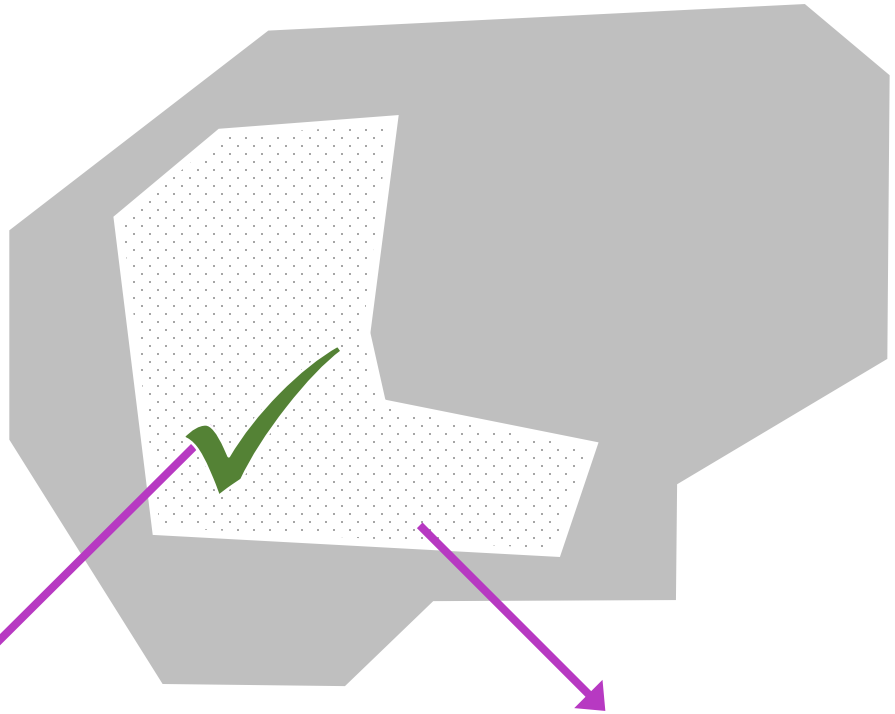
Permission Transfer



permission
transferred to
other thread

exclusive
access permission
held by thread 1

Permission Transfer

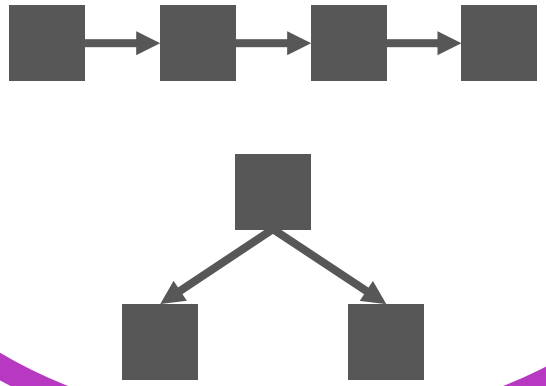


permission
transferred back
to thread 1

exclusive
access permission
held by thread 1

Unbounded Data Structures

Unidirectional



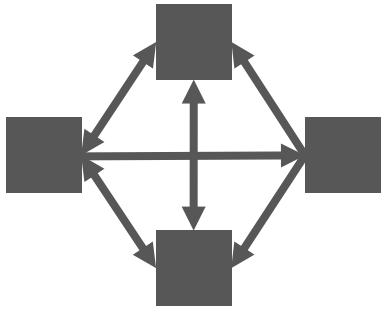
Multi-directional

Random Access

specify via

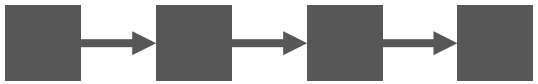
recursive predicate

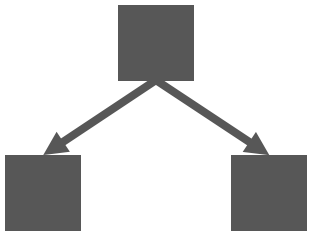
Unstructured



Unbounded Data Structures

Unidirectional

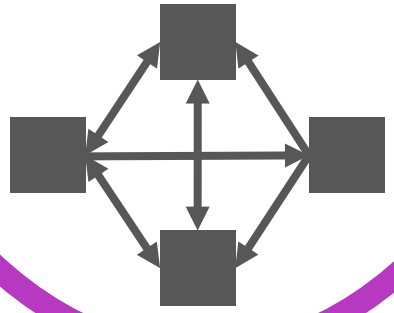




Multi-directional

Random Access

Unstructured



specify via
iterated separating
conjunction

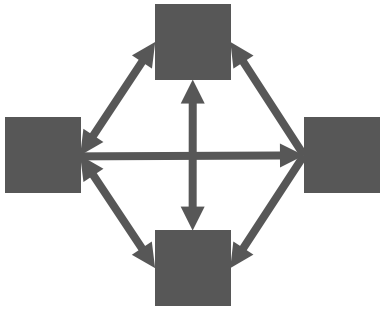
Unbounded Data Structures

Multi-directional

Random Access

specify and verify via iterated separating conjunction

Unstructured



Parallel Array Replace

```
method replace(a: Int[], left, right, from, to: Int)


{
  /* only recursive case shown */
  var mid := left + (right - left) / 2

  fork t1 := replace(a, left, mid,   from, to)
  fork t2 := replace(a, mid,  right, from, to)

  join t1
  join t2
}
```

Parallel Array Replace: Verification Challenges

```
method replace(a: Int[], left, right, from, to: Int)
  requires permission to a[left..right)
  ensures  permission to a[left..right)
{
  /* only recursive case shown */
  var mid := left + (right - left) / 2

  fork t1 := replace(a, left, mid,   from, to)
  fork t2 := replace(a, mid,  right, from, to)

  join t1
  join t2
}
```

verification state representation

Parallel Array Replace: Verification Challenges

```
method replace(a: Int[], left, right, from, to: Int)
  requires permission to a[left..right)
  ensures  permission to a[left..right)
{
  /* only recursive case shown */
  var mid := left + (right - left) / 2

  fork t1 := replace(a, left, mid,   from, to)
  fork t2 := replace(a, mid,  right, from, to)

  join t1
  join t2
}
```

verification state representation

lose permissions

Parallel Array Replace: Verification Challenges

```
method replace(a: Int[], left, right, from, to: Int)
  requires permission to a[left..right)
  ensures  permission to a[left..right)
{
  /* only recursive case shown */
  var mid := left + (right - left) / 2

  fork t1 := replace(a, left, mid,   from, to)
  fork t2 := replace(a, mid,  right, from, to)

  join t1
  join t2
}
```

verification state representation

lose permissions

gain permissions

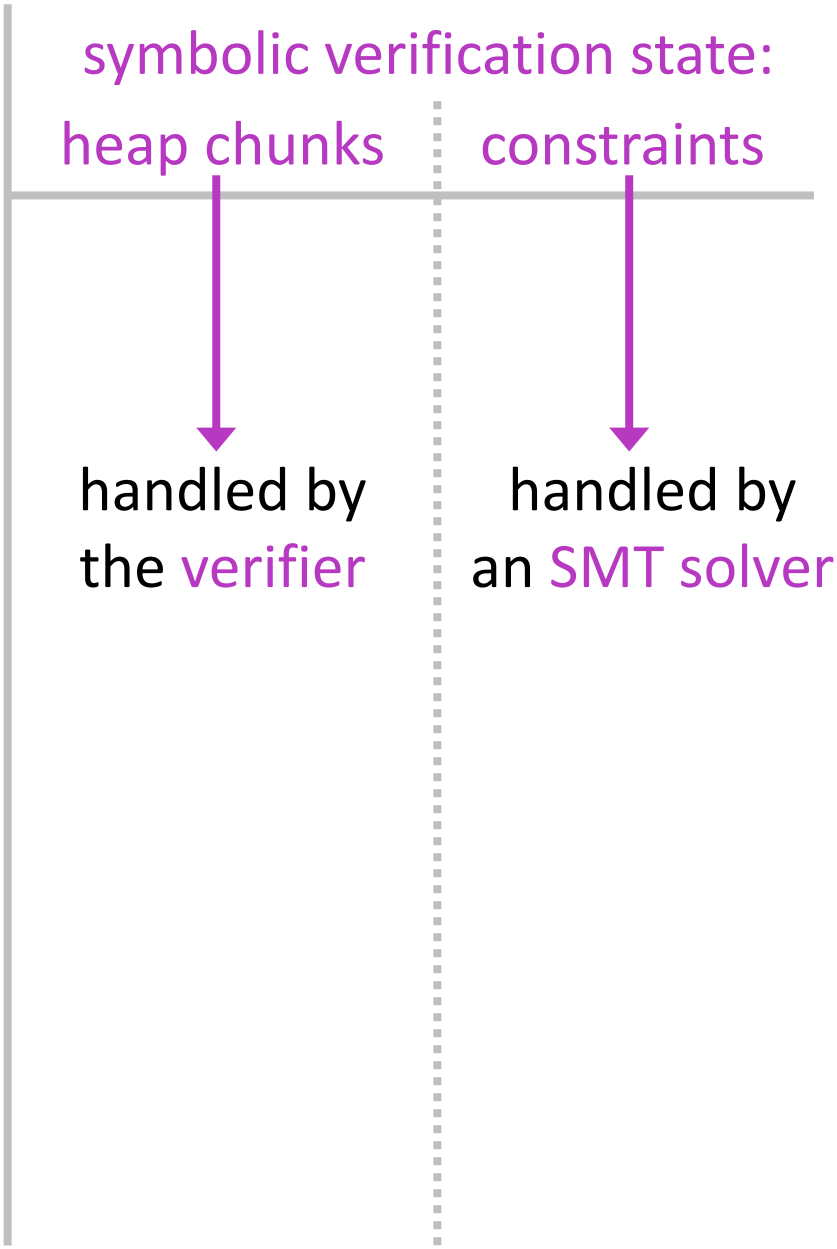
Smallfoot-style Symbolic Execution

```
gain permission to x.f, y.f
assume x.f == y.f

var z := x

fork t := modify_f(z)
join t

assert x.f == y.f
```



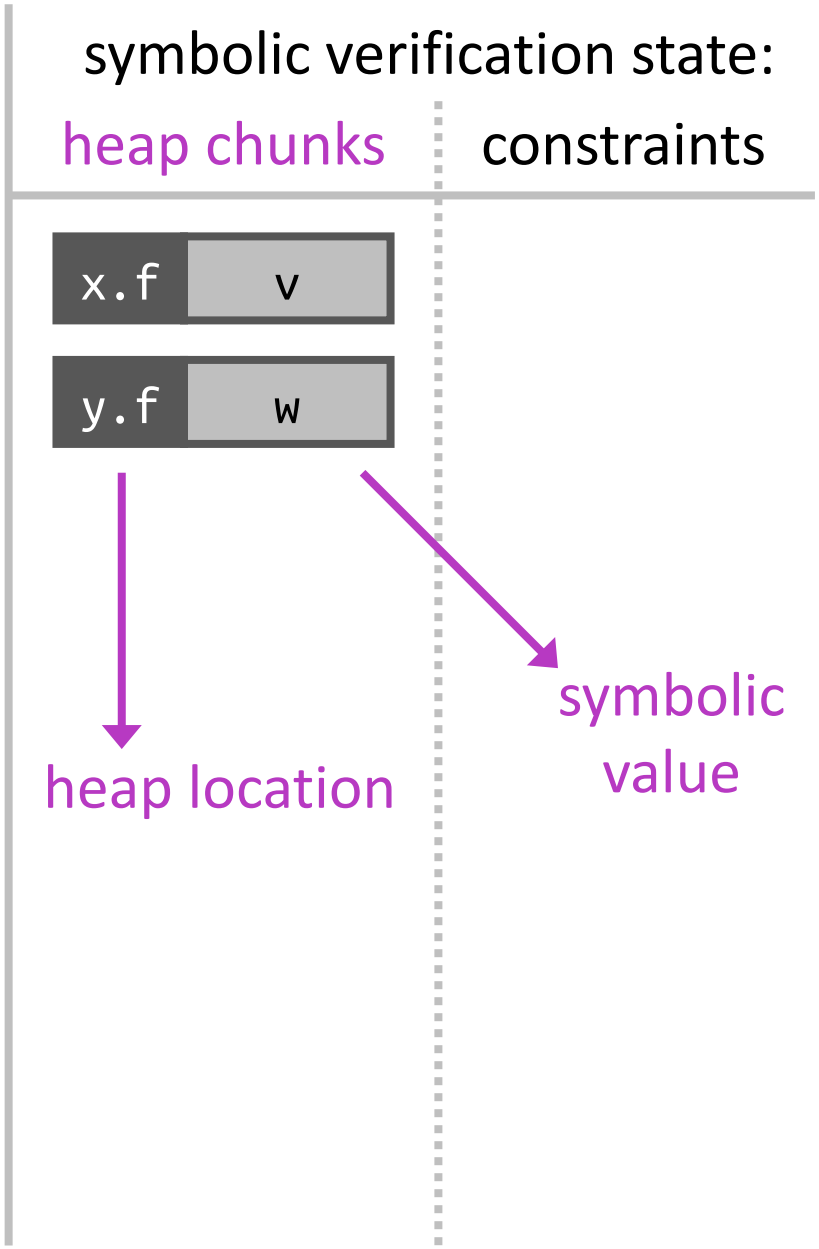
Representing Permissions as Heap Chunks

```
gain permission to x.f, y.f
assume x.f == y.f

var z := x

fork t := modify_f(z)
join t

assert x.f == y.f
```



Representing Value Constraints

```
gain permission to x.f, y.f
assume x.f == y.f

var z := x

fork t := modify_f(z)
join t

assert x.f == y.f
```

symbolic verification state:

heap chunks	constraints
x.f | v	$v = w$
y.f | w	$z = x$

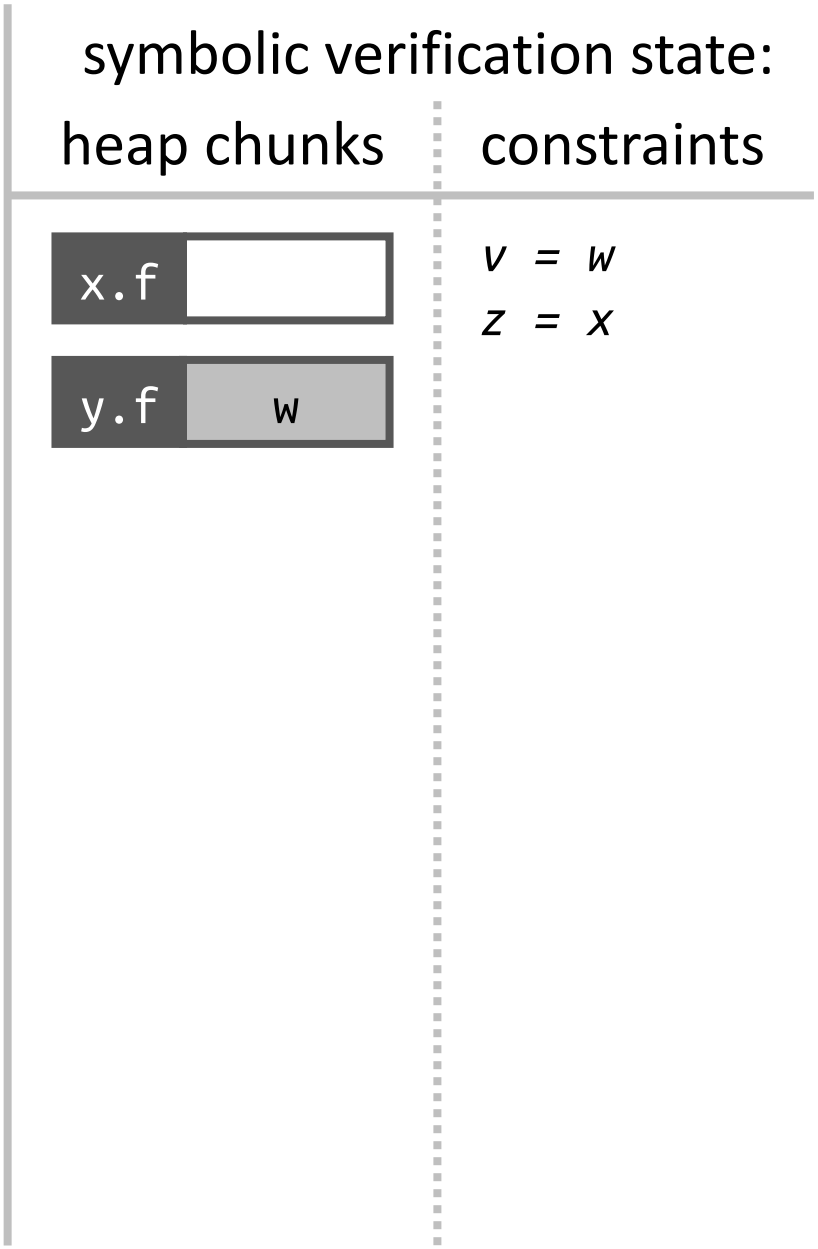
Losing Permissions

```
gain permission to x.f, y.f
assume x.f == y.f

var z := x

fork t := modify_f(z)
join t

assert x.f == y.f
```



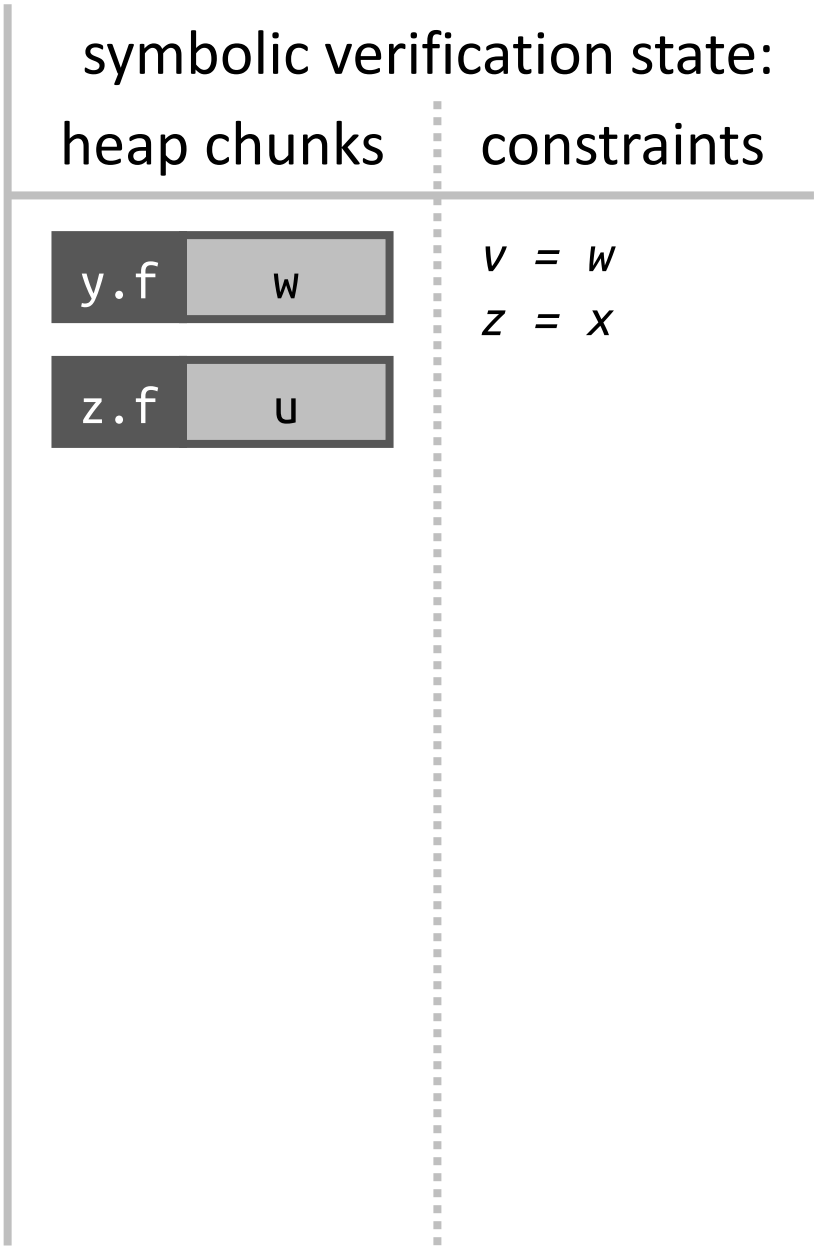
Gaining Permissions

```
gain permission to x.f, y.f
assume x.f == y.f

var z := x

fork t := modify_f(z)
join t

assert x.f == y.f
```



Asserting Value Constraints

```
gain permission to x.f, y.f
assume x.f == y.f

var z := x

fork t := modify_f(z)
join t

assert x.f == y.f
```

SMT query:

is $u = w$ entailed?

symbolic verification state:

heap chunks	constraints
y.f ↦ w	$v = w$
z.f ↦ u	$z = x$

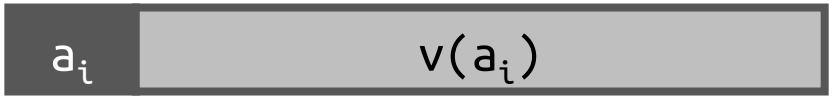
Representing ISCs as Quantified Heap Chunks

```
method replace(a, left, right, from, to)
  requires permission to a[left..right)
  ensures  permission to a[left..right)
{
  var mid := left + (right - left) / 2

  fork t1 := replace(a, left, mid,   from, to)
  fork t2 := replace(a, mid,  right, from, to)

  join t1
  join t2
}
```

$\forall a_i \in a[l..r)$



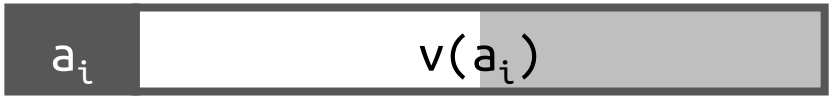
ISCs: Partially Losing Permissions

```
method replace(a, left, right, from, to)
  requires permission to a[left..right)
  ensures  permission to a[left..right)
{
  var mid := left + (right - left) / 2

  fork t1 := replace(a, left, mid,   from, to)
  fork t2 := replace(a, mid,  right, from, to)

  join t1
  join t2
}
```

$\forall a_i \in a[l..r)$



ISCs: Fully Losing Permissions

```
method replace(a, left, right, from, to)
  requires permission to a[left..right)
  ensures  permission to a[left..right)
{
  var mid := left + (right - left) / 2

  fork t1 := replace(a, left, mid,   from, to)
  fork t2 := replace(a, mid,  right, from, to)

  join t1
  join t2
}
```

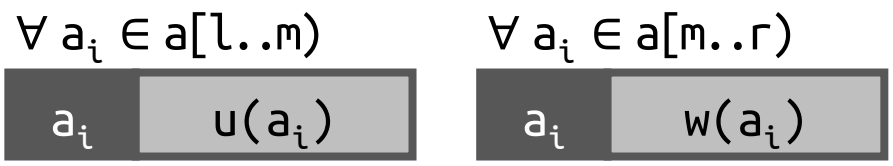
$\forall a_i \in a[l..r)$

ISCs: Gaining Permissions

```
method replace(a, left, right, from, to)
  requires permission to a[left..right)
  ensures  permission to a[left..right)
{
  var mid := left + (right - left) / 2

  fork t1 := replace(a, left, mid,   from, to)
  fork t2 := replace(a, mid,  right, from, to)

  join t1
  join t2
}
```

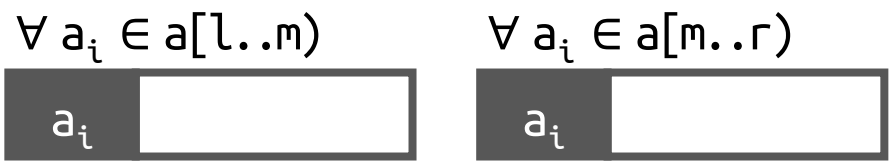


ISCs: Losing Permissions Across Multiple Chunks

```
method replace(a, left, right, from, to)
  requires permission to a[left..right)
  ensures  permission to a[left..right)
{
  var mid := left + (right - left) / 2

  fork t1 := replace(a, left, mid,   from, to)
  fork t2 := replace(a, mid,  right, from, to)

  join t1
  join t2
}
```



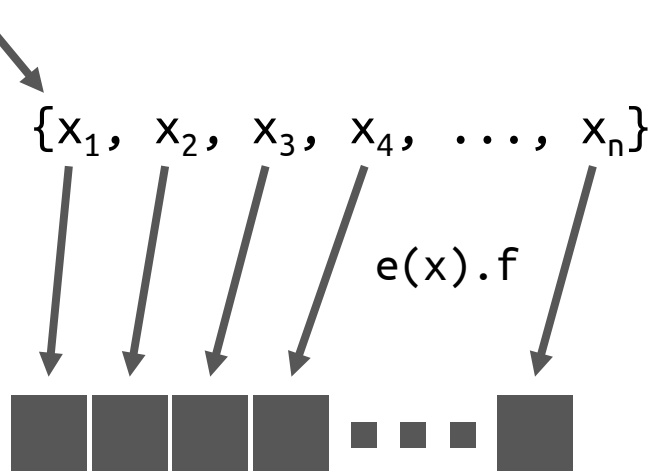
Supporting General Receiver Expressions

```
gain ∀ x ∈ S :: permission to e(x).f
```

```
lose ∀ y ∈ R :: permission to y.f
```

General Receiver Expressions: Challenge

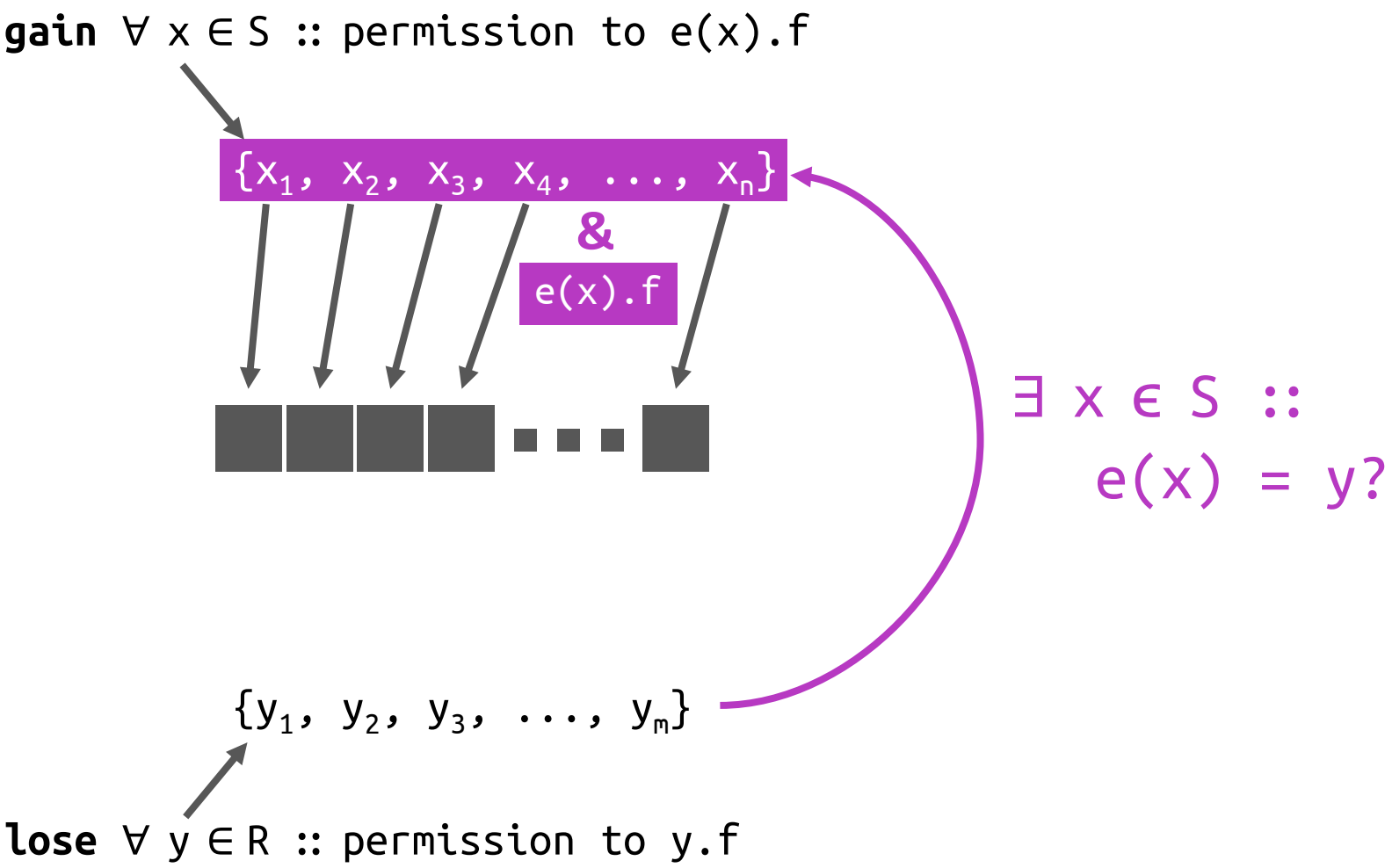
gain ∀ x ∈ S :: permission to e(x).f



{y1, y2, y3, ..., ym} → permission to y.f?

lose ∀ y ∈ R :: permission to y.f

General Receiver Expressions: Challenge



General Receiver Expressions: Injectivity

gain ∀ x ∈ S :: permission to e(x).f

lose ∀ y ∈ R :: permission to y.f

1. Require e(x) to be injective

   (naturally satisfied by e.g. arrays and graphs)

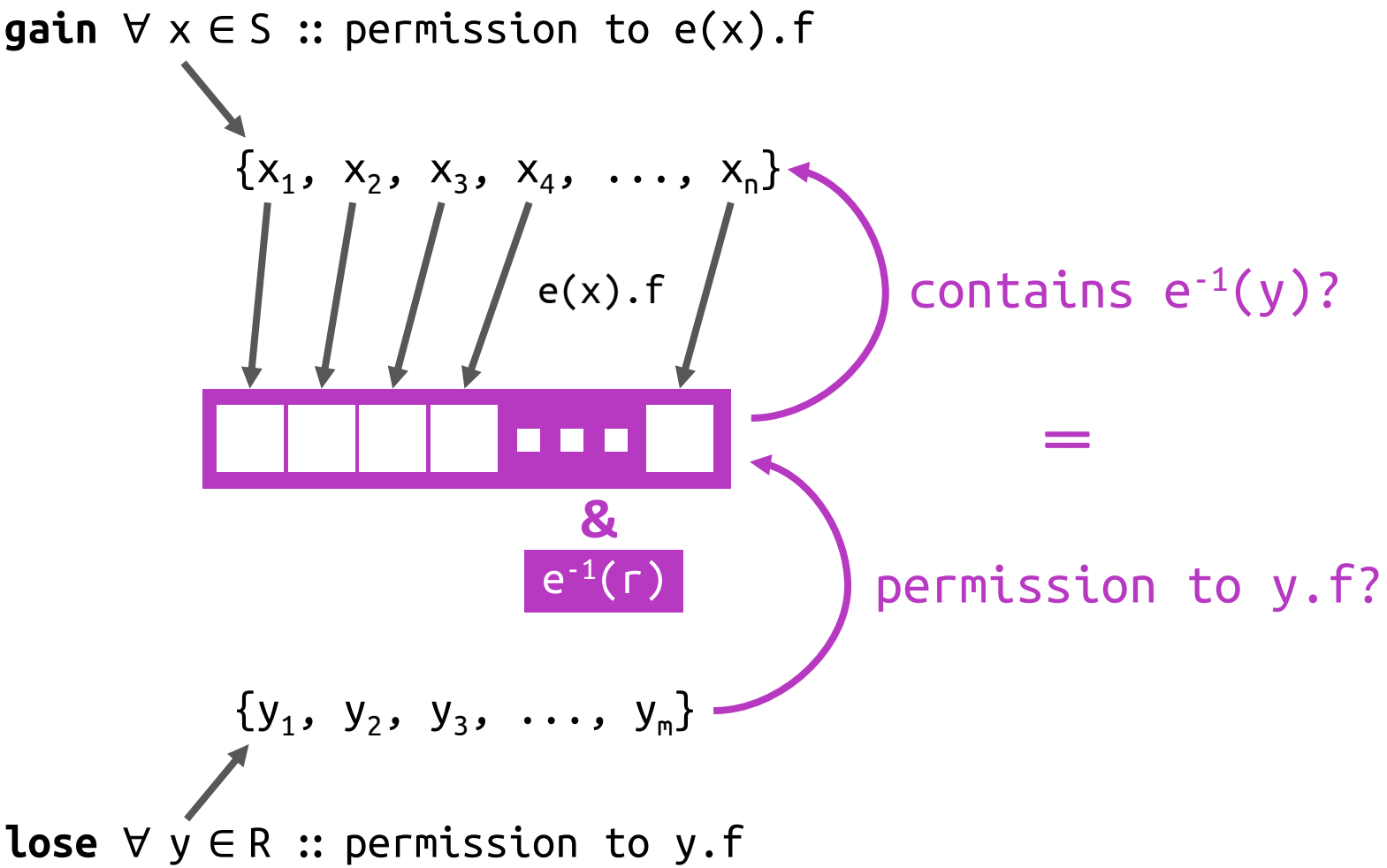
General Receiver Expressions: Inverse Functions

gain ∀ x ∈ S :: permission to e(x).f

lose ∀ y ∈ R :: permission to y.f

1. Require e(x) to be injective
2. Axiomatise inverse function $e^{-1}(x)$ to SMT solver

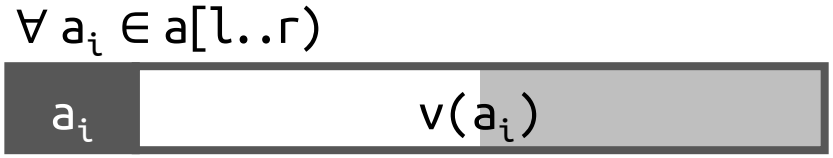
General Receiver Expressions: Efficient Solution



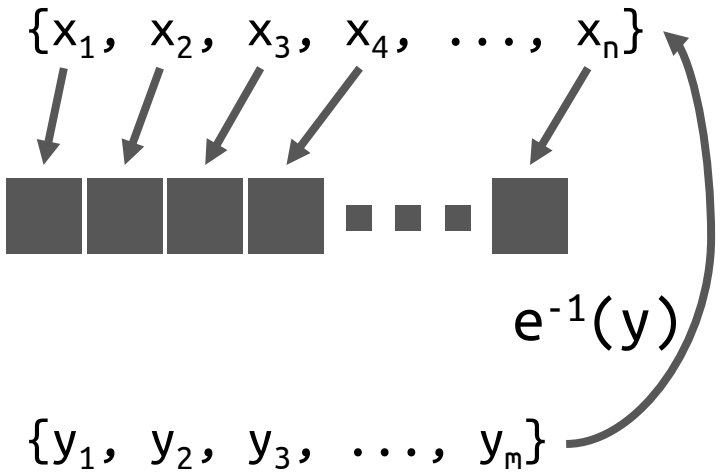
Summary: Presented in this Talk

Representation of ISCs in the verification state

Manipulation thereof: gaining and losing permissions



Injective receiver expressions to avoid existential queries



Overview: Additionally in our Paper

Fractional permissions: shared read access

Complex permission expressions: e.g. for handling non-injective receiver expressions

Integration of ISCs with recursive predicates

Heap-dependent abstraction functions and functional properties in general

SMT challenges: encoding higher-order functions and partial functions; controlling quantifier instantiations

Evaluation: challenging examples, incl. arrays and graphs

Implementation is Open Source

Implemented as part of the

Verification Infrastructure

Support for iterated separating conjunction won the

Distinguished User-Assistance Tool Feature Award

at VerifyThis@ETAPS'16

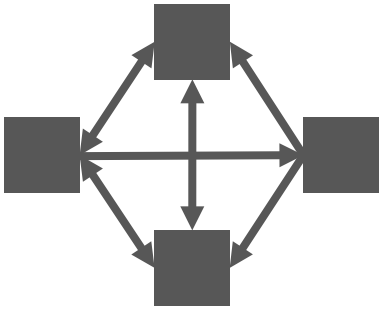
Implementation is Open Source

Implemented as part of the

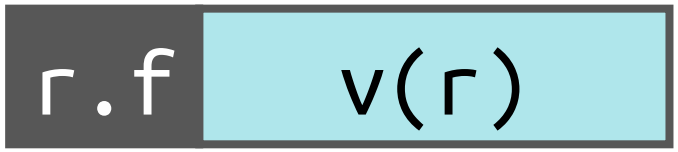
Verification Infrastructure

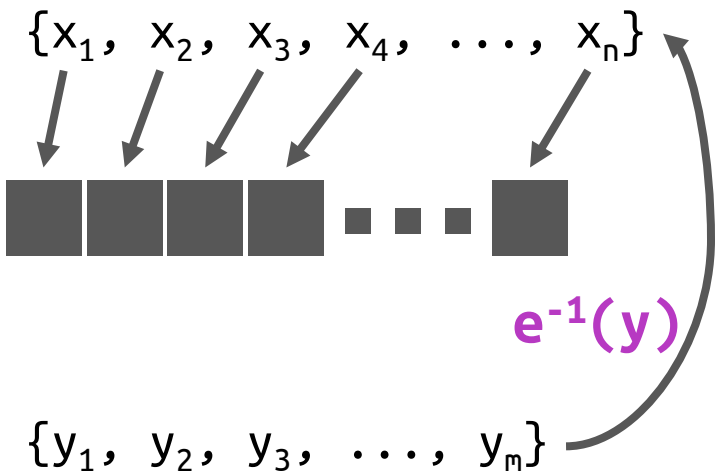
Demo

Automatic Verification of ISCs using Symbolic Execution



$\forall r \in S$





http://viper.ethz.ch