# Advanced Webinar – Quantifiers

Jochen Hoenicke

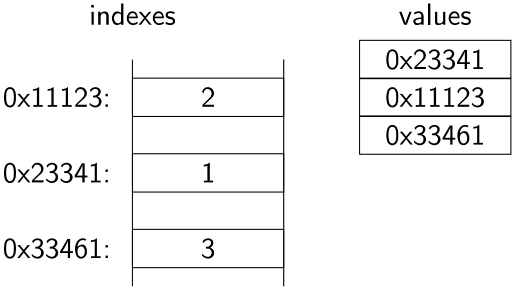
CERTORA

12-July-2023

# Motivating Example: EnumerableSet (from OpenZeppelin)

```
contract EnumerableSet {

    struct Set {
        // Storage of set values
        bytes32[] _values;
        // Position of the value in the
        // ‘values‘ array, plus 1 because
        // index 0 means a value is not in
        // the set.
        mapping(bytes32 => uint256)
            _indexes;
    }
```

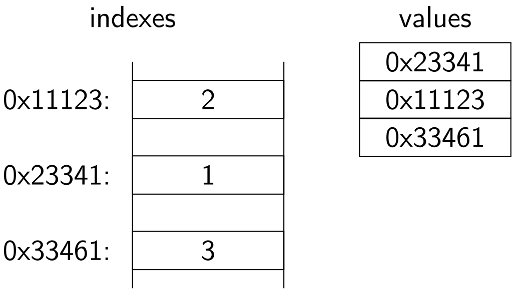



EnumerableSet combines an (unordered) set (implemented by a map) with an array.

- Many operations are $O(1)$: add, remove, contains.
- Values can be enumerated.

```
contract EnumerableSet {

    struct Set {
        // Storage of set values
        bytes32[] _values;
        // Position of the value in the
        // `values` array, plus 1 because
        // index 0 means a value is not in
        // the set.
        mapping(bytes32 => uint256)
            _indexes;
    }
```




We show that the two datastructures are always consistent:

$$\forall i.\ 0 \leq i < \texttt{values.length} \Rightarrow \texttt{indexes}[\texttt{values}[i]] = i + 1$$

$$\forall v.\ \texttt{indexes}[v] \neq 0 \Rightarrow \texttt{indexes}[v] \leq \texttt{values.length} \wedge \texttt{values}[\texttt{indexes}[v] - 1] = v$$

# Old way: Parametrized Invariants

```
invariant indexInv(uint256 index)
    index < values.length =>
     indexes[values[index]] = index + 1 {
    preserved { ... }
}

invariant valueInv(bytes32 values)
    indexes[value] != 0 =>
     indexes[value] <= values.length &&
     values[indexes[value] - 1] = value {
    preserved { ... }
}
```

# Old way: Parametrized Invariants

```
invariant indexInv(uint256 index)
    index < values.length =>
     indexes[values[index]] = index + 1 {
    preserved { ... }
}

invariant valueInv(bytes32 values)
    indexes[value] != 0 =>
     indexes[value] <= values.length &&
     values[indexes[value] - 1] = value {
    preserved { ... }
}
```

```
rule removePreservesOther {
    bytes32 value;
    bytes32 other;

    requireInvariant valuesInv(value);
    requireInvariant indexInv(values.
        length - 1);

    require value != other;
    bool otherInSetBefore = inSet(other);
    remove(value);
    assert inSet(other) ==
        otherInSetBefore;
}
```

User has to figure out all needed invariants.

# New way: Quantified Invariants

```
invariant indexValueInv()
   (forall uint256 index.
    index < values.length =>
     indexes[values[index]] = index + 1)
      &&
   (forall bytes32 value.
    indexes[value] != 0 =>
     indexes[value] <= values.length &&
     values[indexes[value] - 1] = value);
```

```
rule removePreservesOther {
    bytes32 value;
    bytes32 other;

    requireInvariant indexValueInv();

    require value != other;
    bool otherInSetBefore = inSet(other);
    remove(value);
    assert inSet(other) ==
        otherInSetBefore;
}
```

- no `preserved` block needed for invariant.
- user adds full invariant and the prover figures out needed instances.

# Limitations

- Quantified formulas can't invoke solidity view functions.
- No access to storage inside quantifers!
- Prover may not find all necessary instances.

# No Access to Storage

- CVL cannot currently access storage.
- View function cannot be called in quantifiers.

- CVL cannot currently access storage.
- View function cannot be called in quantifiers.

Solution: Use ghost copy and update it in hooks.

```
ghost mapping(bytes32 => uint256) indexes {
    init_state axiom forall bytes32 x. indexes[x] == 0;
}

hook Sstore currentContract.set._inner._indexes[KEY bytes32 value] uint256 newIndex
    STORAGE {
    indexes[value] = newIndex;
}

hook Sload uint256 index currentContract.set._inner._indexes[KEY bytes32 value]
    STORAGE {
    require indexes[value] == index;
}
```

# Heuristic Instantiation 1

$$\forall i.\ a[i] \leq a[i+1]$$

- Needs a lot of instances to prove $a[0] \leq a[100]$.
- Heuristic instantiation will only add a few to ensure termination.

# Heuristic Instantiation 1

$$\forall i.\ a[i] \leq a[i+1]$$

- Needs a lot of instances to prove $a[0] \leq a[100]$.
- Heuristic instantiation will only add a few to ensure termination.

Instead, use the following equivalent formula:

$$\forall i.\ \forall j. i \leq j \Rightarrow a[i] \leq a[j]$$

With this $a[0] \leq a[100]$ can be proved with our heuristic.

# Heuristic Instantiation 2

Instantiation is based on searching for values that build existing terms:

$$\forall i.\ 1 \leq i \leq \texttt{values.length} \Rightarrow \texttt{indexes}[\texttt{values}[i-1]] = i$$

- If program reads $\texttt{values}[j]$, then useful instance is $i := j + 1$.
- More complicated if there are possible overflows.
- Even more complicated if variables are added or multiplication is used.
- The heuristic currently does not support this.

# Heuristic Instantiation 2

Instantiation is based on searching for values that build existing terms:

$$\forall i.\ 1 \leq i \leq \texttt{values.length} \Rightarrow \texttt{indexes}[\texttt{values}[i-1]] = i$$

- If program reads $\texttt{values}[j]$, then useful instance is $i := j + 1$.
- More complicated if there are possible overflows.
- Even more complicated if variables are added or multiplication is used.
- The heuristic currently does not support this.

Instead, use the following quantified invariant:

$$\forall i.\ 0 \leq i < \texttt{values.length} \Rightarrow \texttt{indexes}[\texttt{values}[i]] = i + 1$$

DEMO

# Linked List Example

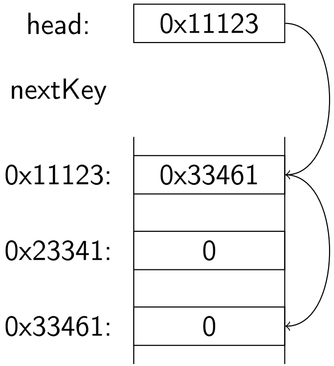
```
contract LinkedList {
    struct Element {
        bytes32 _nextKey;
        uint256 _valid;
    }

    struct List {
        bytes32 _head;
        mapping(bytes32 => bytes32) _elems;
    }
```

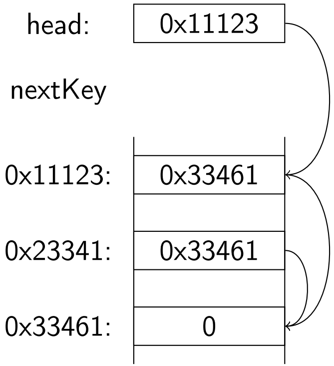
```
    function insertAfter(bytes32 key,
        bytes32 afterKey) {
        require...
        _elems[key].nextKey =
            _elem[afterKey].nextKey;
        _elems[key].valid = 1
        _elems[afterKey].nextKey = key;
    }
```

Show that key is in the list after calling insertAfter.

# Linked List: Insert

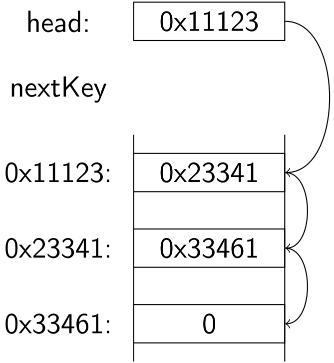



`insertAfter(key=0x23341, afterKey=0x11123):`

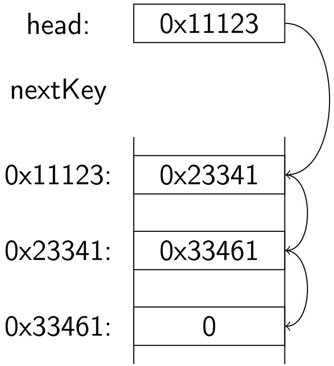



```
insertAfter(key=0x23341, afterKey=0x11123):
        _elements[key].nextKey = _elements[afterKey].nextKey;
```

# Linked List: Insert




```
insertAfter(key=0x23341, afterKey=0x11123):
        _elements[key].nextKey = _elements[afterKey].nextKey;
        _elements[afterkey].nextKey = key;
```

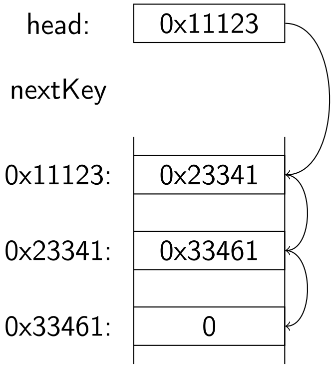



```
insertAfter(key=0x23341, afterKey=0x11123):
        _elements[key].nextKey = _elements[afterKey].nextKey;
        _elements[afterkey].nextKey = key;
```

Can we express that key is in the list?

$$key = head \lor key = nextKey[head] \lor key = nextKey[nextKey[head]] \lor \dots$$

# Linked List: Insert




```
insertAfter(key=0x23341, afterKey=0x11123):
        _elements[key].nextKey = _elements[afterKey].nextKey;
        _elements[afterkey].nextKey = key;
```

Can we express that key is in the list?

$$key = head \lor key = nextKey[head] \lor key = nextKey[nextKey[head]] \lor \dots$$

Not first-order expressible (using only *head* and *nextKey*).

# Linked List: Reachability

Add another ghost for reachability!

```
ghost reach(bytes32, bytes32) returns bool {
    init_state axiom forall bytes32 X. forall bytes32 Y.
        reach(X, Y) == (X == Y || Y == to_bytes32(0));
}
```

`reach`$(x, y)$: there is a path from $x$ to $y$ of length $\geq 0$ using `nextKey`.

- Initialize `reach` as above.
- Update `reach` in the store hook for `nextKey`.
- Use `reach` to express reachability in spec.

# Updating Reachability

This is how `reach` changes on `nextKey`$[a] := b$:

$$\forall x, y.\ \texttt{reach@new}(x, y) = (\texttt{reach@old}(x, y) \land (\neg\texttt{reach@old}(x, a) \lor \texttt{reach@old}(y, a)))$$
$$\lor\ (\texttt{reach@old}(x, a) \land \texttt{reach@old}(b, y))$$

DEMO