Lecturecise 24: Modeling Mutable Program Heap

2013

First, a look at mutable value arrays

Simplest and easiest to reason mutable arrays are mutable *value* arrays:

- stored in fixed location in memory, denoted by a unique name
- the only way to read and write to it is through this name
- copying arrays can only be done through cloning (value semantics)

These are arrays supported in Leon currently.

```
def bubbleSort(a: Array[Int]): Array[Int] = {
  var i = a.length − 1; var j = 0; val sa = a.clone
  while(i > 0) {
    j = 0
    while(j < i) {
      if(sa(j) > sa(j+1)) {
        val tmp = sa(j);
        sa(j) = sa(j+1);
        sa(j+1) = tmp
      }
      j = j + 1
    }; i = i − 1
  }; sa
} ensuring(sorted(_, 0, a.length−1))
```

Subtlety of Array Assignment

Rule for wp of assignment of expression E to variable x, for postcondition P:

$$wp(x = E, P) =$$

Subtlety of Array Assignment

Rule for wp of assignment of expression E to variable x, for postcondition P:

$$wp(x = E, P) = P[x := E]$$

Example:

$$wp(x = y + 1, x > 5) =$$

Subtlety of Array Assignment

Rule for wp of assignment of expression E to variable x, for postcondition P:

$$wp(x = E, P) = P[x := E]$$

Example:

$$wp(x = y + 1, x > 5) = y + 1 > 5$$

Subtlety of Array Assignment

Rule for wp of assignment of expression E to variable x, for postcondition P:

$$wp(x = E, P) = P[x := E]$$

Example:

$$wp(x = y + 1, x > 5) = y + 1 > 5$$

wp of assignment to an array cell:

$$wp(a(i) = y + 1, a(i) > 5) =$$

Subtlety of Array Assignment

Rule for wp of assignment of expression E to variable x, for postcondition P:

$$wp(x = E, P) = P[x := E]$$

Example:

$$wp(x = y + 1, x > 5) = y + 1 > 5$$

wp of assignment to an array cell:

$$wp(a(i) = y + 1, a(i) > 5) = y + 1 > 5$$

Subtlety of Array Assignment

Rule for wp of assignment of expression E to variable x, for postcondition P:

$$wp(x = E, P) = P[x := E]$$

Example:

$$wp(x = y + 1, x > 5) = y + 1 > 5$$

wp of assignment to an array cell:

$$wp(a(i) = y + 1, a(i) > 5) = y + 1 > 5$$

$$wp(a(i) = y + 1, a(i) > 5 \wedge a(j) > 3)$$

$$=$$

Subtlety of Array Assignment

Rule for wp of assignment of expression E to variable x, for postcondition P:

$$wp(x = E, P) = P[x := E]$$

Example:

$$wp(x = y + 1, x > 5) = y + 1 > 5$$

wp of assignment to an array cell:

$$wp(a(i) = y + 1, a(i) > 5) = y + 1 > 5$$

$$\begin{array}{ll} & wp(a(i) = y + 1, a(i) > 5 \wedge a(j) > 3) \\ = & y + 1 > 5 \wedge a(j) > 3 \text{ ???} \end{array}$$

Subtlety of Array Assignment

Rule for wp of assignment of expression E to variable x, for postcondition P:

$$wp(x = E, P) = P[x := E]$$

Example:

$$wp(x = y + 1, x > 5) = y + 1 > 5$$

wp of assignment to an array cell:

$$wp(a(i) = y + 1, a(i) > 5) = y + 1 > 5$$

$$\begin{array}{rl} & wp(a(i) = y + 1, a(i) > 5 \land a(j) > 3) \\ = & y + 1 > 5 \land a(j) > 3 \text{ ???} \\ = & y + 1 > 5 \land ((y + 1 > 3 \land i = j) \lor (a(j) > 3 \land i \neq j)) \end{array}$$

Whether two array expressions $a(i)$ and $a(j)$ denote the same location depends on whether $i = j$.

Function updates and translation of mutable value arrays

Function updates:

$$f[i := v] = f'$$

where

$$f'(j) = \begin{cases} f(j), & \text{if } j \neq i \\ v, & \text{if } j = i \end{cases}$$

Translate $var\ a : Array[Int]$ into $var\ a : Int \Rightarrow Int$ and then:

- $x = a(i) \leadsto$

Function updates and translation of mutable value arrays

Function updates:

$$f[i := v] = f'$$

where

$$f'(j) = \begin{cases} f(j), & \text{if } j \neq i \\ v, & \text{if } j = i \end{cases}$$

Translate $var\ a : Array[Int]$ into $var\ a : Int \Rightarrow Int$ and then:

- $x = a(i) \rightsquigarrow x = a(i)$

Function updates and translation of mutable value arrays

Function updates:

$$f[i := v] = f'$$

where

$$f'(j) = \begin{cases} f(j), & \text{if } j \neq i \\ v, & \text{if } j = i \end{cases}$$

Translate $var\ a : Array[Int]$ into $var\ a : Int \Rightarrow Int$ and then:

- $x = a(i) \leadsto x = a(i)$
- $a(i) = x \leadsto$

Function updates and translation of mutable value arrays

Function updates:

$$f[i := v] = f'$$

where

$$f'(j) = \begin{cases} f(j), & \text{if } j \neq i \\ v, & \text{if } j = i \end{cases}$$

Translate $var\ a : Array[Int]$ into $var\ a : Int \Rightarrow Int$ and then:

- $x = a(i) \leadsto x = a(i)$
- $a(i) = x \leadsto a = a[i := x]$

Function updates and translation of mutable value arrays

Function updates:

$$f[i := v] = f'$$

where

$$f'(j) = \begin{cases} f(j), & \text{if } j \neq i \\ v, & \text{if } j = i \end{cases}$$

Translate $var\ a : Array[Int]$ into $var\ a : Int \Rightarrow Int$ and then:

- $x = a(i) \leadsto x = a(i)$
- $a(i) = x \leadsto a = a[i := x]$
- $a = b.clone \leadsto$

Function updates and translation of mutable value arrays

Function updates:

$$f[i := v] = f'$$

where

$$f'(j) = \begin{cases} f(j), & \text{if } j \neq i \\ v, & \text{if } j = i \end{cases}$$

Translate $var\ a : Array[Int]$ into $var\ a : Int \Rightarrow Int$ and then:

- $x = a(i) \rightsquigarrow x = a(i)$
- $a(i) = x \rightsquigarrow a = a[i := x]$
- $a = b.clone \rightsquigarrow a = b$

Function updates and translation of mutable value arrays

Function updates:

$$f[i := v] = f'$$

where

$$f'(j) = \begin{cases} f(j), & \text{if } j \neq i \\ v, & \text{if } j = i \end{cases}$$

Translate $var\ a : Array[Int]$ into $var\ a : Int \Rightarrow Int$ and then:

- $x = a(i) \leadsto x = a(i)$
- $a(i) = x \leadsto a = a[i := x]$
- $a = b.clone \leadsto a = b$
- $a = b \leadsto$ not defined as assignment

Applying the desugaring of value arrays for verification

$$wp(a(i) = y + 1, a(i) > 5 \land a(j) > 3)$$

Applying the desugaring of value arrays for verification

$$
\begin{array}{ll}
 & wp(a(i) = y + 1, a(i) > 5 \wedge a(j) > 3) \\
= & wp(a = a[i := y + 1], a(i) > 5 \wedge a(j) > 3)
\end{array}
$$

Applying the desugaring of value arrays for verification

$$
\begin{array}{rl}
& wp(a(i) = y + 1, a(i) > 5 \wedge a(j) > 3) \\
= & wp(a = a[i := y + 1], a(i) > 5 \wedge a(j) > 3) \\
= & a[i := y + 1](i) > 5 \wedge a[i := y + 1](j) > 3
\end{array}
$$

Applying the desugaring of value arrays for verification

$$
\begin{array}{rl}
& wp(a(i) = y + 1, a(i) > 5 \wedge a(j) > 3) \\
= & wp(a = a[i := y + 1], a(i) > 5 \wedge a(j) > 3) \\
= & a[i := y + 1](i) > 5 \wedge a[i := y + 1](j) > 3 \\
= & y + 1 > 5 \wedge a[i := y + 1](j) = v \wedge v > 3
\end{array}
$$

Applying the desugaring of value arrays for verification

$$
\begin{aligned}
& wp(a(i) = y + 1, a(i) > 5 \wedge a(j) > 3) \\
= \;& wp(a = a[i := y + 1], a(i) > 5 \wedge a(j) > 3) \\
= \;& a[i := y + 1](i) > 5 \wedge a[i := y + 1](j) > 3 \\
= \;& y + 1 > 5 \wedge a[i := y + 1](j) = v \wedge v > 3 \\
= \;& \exists v.\; y + 1 > 5 \wedge ((i = j \wedge v = y + 1) \vee (i \neq j \wedge v = a(j))) \wedge v > 3
\end{aligned}
$$

Applying the desugaring of value arrays for verification

$$\begin{array}{rl}
& wp(a(i) = y + 1, a(i) > 5 \wedge a(j) > 3) \\
= & wp(a = a[i := y + 1], a(i) > 5 \wedge a(j) > 3) \\
= & a[i := y + 1](i) > 5 \wedge a[i := y + 1](j) > 3 \\
= & y + 1 > 5 \wedge a[i := y + 1](j) = v \wedge v > 3 \\
= & \exists v.\ y + 1 > 5 \wedge ((i = j \wedge v = y + 1) \vee (i \neq j \wedge v = a(j))) \wedge v > 3 \\
= & \exists v.\ y + 1 > 5 \wedge \qquad ((i = j \wedge v = y + 1 \wedge v > 3) \\
& \qquad\qquad\qquad\qquad \vee \quad (i \neq j \wedge v = a(j) \wedge v > 3))
\end{array}$$

Applying the desugaring of value arrays for verification

$$
\begin{array}{rl}
 & wp(a(i) = y + 1, a(i) > 5 \wedge a(j) > 3) \\
= & wp(a = a[i := y + 1], a(i) > 5 \wedge a(j) > 3) \\
= & a[i := y + 1](i) > 5 \wedge a[i := y + 1](j) > 3 \\
= & y + 1 > 5 \wedge a[i := y + 1](j) = v \wedge v > 3 \\
= & \exists v.\ y + 1 > 5 \wedge ((i = j \wedge v = y + 1) \vee (i \neq j \wedge v = a(j))) \wedge v > 3 \\
= & \exists v.\ y + 1 > 5 \wedge \qquad ((i = j \wedge v = y + 1 \wedge v > 3) \\
 & \qquad\qquad\qquad \vee \quad (i \neq j \wedge v = a(j) \wedge v > 3)) \\
= & y + 1 > 5 \wedge ((y + 1 > 3 \wedge i = j) \vee (a(j) > 3 \wedge i \neq j))
\end{array}
$$

In general, we can transform array updates into if-then-else, and then into disjunctions

Array Bounds

To account for bounds checking what assertions should we add in the translation of:

1. $x = a(i)$:

Array Bounds

To account for bounds checking what assertions should we add in the translation of:

1. $x = a(i)$:

$$\begin{array}{l} assert(0 \leq i \wedge i < a_{size}) \\ x = a(i) \end{array}$$

2. $a(i) = x$:

Array Bounds

To account for bounds checking what assertions should we add in the translation of:

1. $x = a(i)$:
$$\begin{array}{l} assert(0 \leq i \wedge i < a_{size}) \\ x = a(i) \end{array}$$
2. $a(i) = x$:
$$\begin{array}{l} assert(0 \leq i \wedge i < a_{size}) \\ a = a[i := x] \end{array}$$

We view an array as a pair (a, a_{size})

Array Bounds

To account for bounds checking what assertions should we add in the translation of:

1. $x = a(i)$:
$$\begin{array}{l} assert(0 \leq i \wedge i < a_{size}) \\ x = a(i) \end{array}$$
2. $a(i) = x$:
$$\begin{array}{l} assert(0 \leq i \wedge i < a_{size}) \\ a = a[i := x] \end{array}$$

We view an array as a pair (a, a_{size})

Each array comes with its size. That's a good thing.

- C-like arrays whose size cannot be determined either at compile-time or run-time belong to the assembly language, and have been the cause of buffer overflows
- Microsoft introduced static checking tools to check array bounds as part of their build process, dramatically reducing such errors.
- To make this feasible, tools require each array to be passed with arguments that store its bounds.

Exercise

Translate into checks and updates:

```
if (a(i) > 0) {
  b(a(k))= b(k) + a(a(i))
}
```

Mutable Value Maps

Everything we said about mutable arrays holds for mutable value maps, and for 'var'-s storing any other immutable data structure

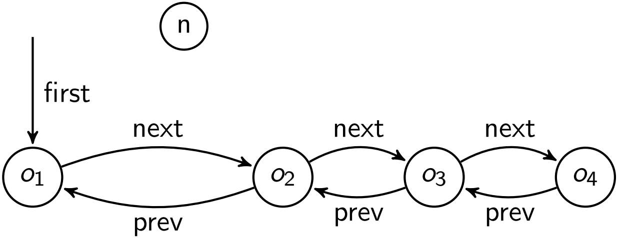
Imperative update to a component is a global update with functional update on the right-hand side

$$m(key) = value \quad \rightsquigarrow \quad m = m[key := value]$$

After this transformation, we can treat maps just as we treat integers

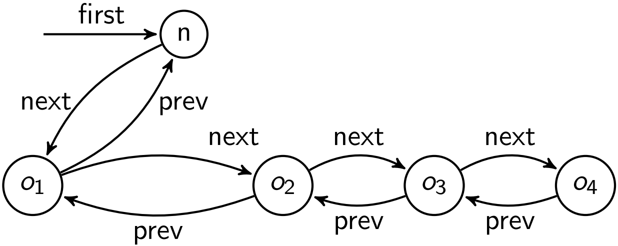
This does **not** work for data structures whose internal components can be accessed and changed from outside.

Linked List Insertion



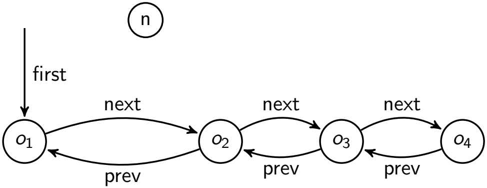

```
if (first == null)
   first = n;
else {
   n.next = first;
   first.prev = n;
   first = n;
}
```

insert(first,n):



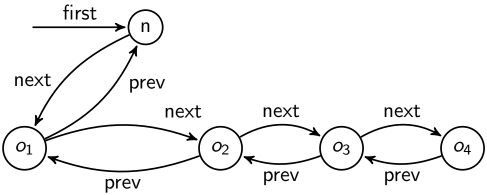
How to verify such code?

Modeling Linked Structures using Relations




```
if (first == null)
  first = n;
else {
  n.next = first;
  first.prev = n;
  first = n;
}
```

$$next = \{(o_1, o_2), (o_2, o_3), (o_3, o_4)\}$$
$$prev = \{(o_2, o_1), (o_3, o_2), (o_4, o_3)\}$$



$$next' = \{(o_1, o_2), (o_2, o_3), (o_3, o_4), (n, o_1)\}$$
$$prev' = \{(o_2, o_1), (o_3, o_2), (o_4, o_3), (o_1, n)\}$$

Change of relations (partial functions):

$$next' = next \cup \{(n, o_1)\}$$
$$prev' = prev \cup \{(o_1, n)\}$$

using assignments:

next = next[n:=first]
prev = prev[first:=n]

Reading Fields

Statement

$$y = x.next$$

computes the value of y simply as

$$y = next(x)$$

We should not de-reference null, so we add this check.
$y = x.next$ translates into

$$\begin{array}{l} assert(x \neq null); \\ y = next(y) \end{array}$$

We assume that the type system ensures that if x is not null then the value $next(y)$ is defined. Otherwise, we could add the corresponding check:

$$\begin{array}{l} assert(x \in dom(next)); \\ y = next(y) \end{array}$$

where $dom(r) = \{x | \exists y.\ (x, y) \in r\}$

Writing Fields

We represent each field using a global partial function
Statement

$$x.next = y$$

changes heap according to this update:

$$next' = next[x := y]$$

which is a notation that expands to:

$$next' = \{(u, v) | (u = x \wedge v = y) \vee (u \neq v \wedge (u, v) \in next)\}$$

We should not assign fields of 'null', so we also add this check.
$x.next = y$ can translate into an imperative language with global maps:

$$\begin{array}{ll} assert(x \neq null); & \\ next = next[x := y] & \quad \text{shorthand assignment : } \; next(x) = y \end{array}$$

Why we need functions

Say we have $x.f$ and $y.f$ in the program.
Why not replace them simply with fresh variables x_f and y_f?
Does this assertion hold for two distinct values p, q?

$$
\begin{aligned}
&var\ xf = ...\\
&var\ yf = ...\\
&xf = p\\
&yf = q\\
&assert(xf == p)
\end{aligned}
$$

Why we need functions

Say we have $x.f$ and $y.f$ in the program.
Why not replace them simply with fresh variables x_f and y_f?
Does this assertion hold for two distinct values p, q?

$$\begin{aligned}
&var\ xf = ...\\
&var\ yf = ...\\
&xf = p\\
&yf = q\\
&assert(xf == p)
\end{aligned}$$

Yes. The value of xf is still p

Why we need functions

Say we have $x.f$ and $y.f$ in the program.
Why not replace them simply with fresh variables x_f and y_f?
Does this assertion hold for two distinct values p, q?

$$
\begin{aligned}
&var\ xf = ...\\
&var\ yf = ...\\
&xf = p\\
&yf = q\\
&assert(xf == p)
\end{aligned}
$$

Yes. The value of xf is still p
Does this assertion hold?

$$
\begin{aligned}
&...\\
&x.f = p\\
&y.f = q\\
&assert(x.f == p)
\end{aligned}
$$

Why we need functions

Say we have $x.f$ and $y.f$ in the program.
Why not replace them simply with fresh variables x_f and y_f?
Does this assertion hold for two distinct values p, q?

$$
\begin{aligned}
&var\ xf = ...\\
&var\ yf = ...\\
&xf = p\\
&yf = q\\
&assert(xf == p)
\end{aligned}
$$

Yes. The value of xf is still p
Does this assertion hold?

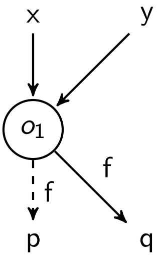
$$
\begin{aligned}
&...\\
&x.f = p\\
&y.f = q\\
&assert(x.f == p)
\end{aligned}
$$

Depends.

Aliasing

Does the assertion hold in this case:

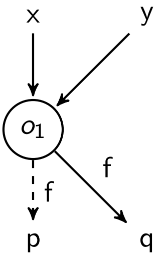
$x = y$
$x.f = p$
$y.f = q$
$assert(x.f == p)$



Aliasing

Does the assertion hold in this case:

$x = y$
$x.f = p$
$y.f = q$
$assert(x.f == p)$

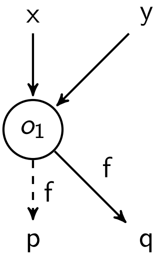


No! y and x are **aliased** references, denote the same object
Even though left hand sides $x.f$ and $y.f$ look different, they interfere

Aliasing

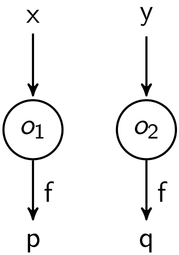
Does the assertion hold in this case:

$x = y$
$x.f = p$
$y.f = q$
$assert(x.f == p)$



No! y and x are **aliased** references, denote the same object
Even though left hand sides $x.f$ and $y.f$ look different, they interfere
Does it hold in this case:

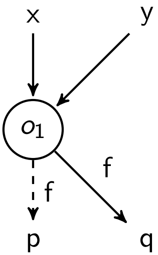
$assume(x \neq y)$
$x.f = p$
$y.f = q$
$assert(x.f == p)$



Aliasing

Does the assertion hold in this case:

$x = y$
$x.f = p$
$y.f = q$
$assert(x.f == p)$

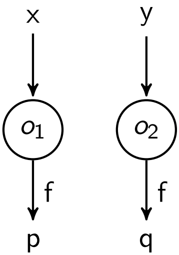


No! y and x are **aliased** references, denote the same object

Even though left hand sides $x.f$ and $y.f$ look different, they interfere

Does it hold in this case:

$assume(x \neq y)$
$x.f = p$
$y.f = q$
$assert(x.f == p)$

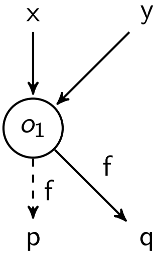


Yes.

Fields as functions demystify aliasing

Does the assertion hold in this case:

$x = y$
$x.f = p$
$y.f = q$
$assert(x.f == p)$

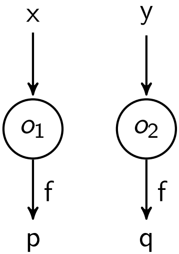


$x = y$
$f(x) = p$
$f(y) = q$
$assert(f(x) == p)$

Does not hold. Indices x, y are the same

Does it hold in this case:

$assume(x \neq y)$
$x.f = p$
$y.f = q$
$assert(x.f == p)$



$assume(x \neq y)$
$f(x) = p$
$f(y) = q$
$assert(f(x) == p)$

Holds. Indices are distinct

Example weakest precondition computation

Recall $wp(v = e, P) = P[v := e]$ (substitution)
Ignoring null checks, we have the following:

$$\begin{array}{l} wp(x.f = p; y.f = q, x.f == p) = \\ wp(f = f[x := p]; f = f[y := q], f(x) == p) = \\ wp(f = f[x := p], (f[y := q])(x) = p) = \\ ((f[x := p])[y := q])(x) = p \end{array}$$

If h is a function then

$$h[a := b](u) \;=\; v \quad \Leftrightarrow \quad (u = a \wedge v = b) \vee (u \neq a \wedge v = h(u))$$

Thus

$$\begin{array}{rl} & ((f[x := p])[y := q])(x) = p \\ \Leftrightarrow & (x = y \wedge p = q) \;\vee\; (x \neq y \wedge p = (f[x := p])(x)) \\ \Leftrightarrow & (x = y \wedge p = q) \;\vee\; (x \neq y \wedge p = p) \\ \Leftrightarrow & (x = y \wedge p = q) \;\vee\; x \neq y \end{array}$$

Characterizes precisely the weakest condition under which assertion holds

Exercise: translate into checks and function updates

$$\text{class } C \ \{\text{var } f : C\}$$

Statement:

x.f.f= z.f + y.f.f.f

Exponentially many cases in aliasing

Note that each write introduces an update, which later creates case analysis.

This creates either exponentially many cases

To handle array updates efficiently, SMT solver support theory of functions with updates, which are called theories of arrays.

Array theories (or simply disjunctions) allow verification conditions to remain polynomial

Simple theories of arrays can be eliminated using if-then-else, which reduces to fresh variables and disjunctions

Modeling dynamic allocation (new, fresh objects)

Now can we prove this:

$$\begin{array}{l} x = new\ C() \\ y = new\ C() \\ assert(x \neq y) \end{array}$$

Modeling dynamic allocation (new, fresh objects)

Now can we prove this:

$$\begin{aligned} &x = new\ \ C() \\ &y = new\ \ C() \\ &assert(x \neq y) \end{aligned}$$

Can we introduce global variables and assumptions that correctly describe fresh objects?

Modeling dynamic allocation (new, fresh objects)

Now can we prove this:

$$\begin{array}{l} x = new\ \ C() \\ y = new\ \ C() \\ assert(x \neq y) \end{array}$$

Can we introduce global variables and assumptions that correctly describe fresh objects?
Global set *alloc* denotes objects allocated so far

$$x = new\ \ C()$$

denotes (for now):

$$\begin{array}{l} havoc(x) \\ assume(x \notin alloc) \\ alloc = alloc \cup \{x\} \end{array}$$

How allocated set models fresh objects

Original program

$$
\begin{aligned}
&x = \mathit{new}\ C()\\
&y = \mathit{new}\ C()\\
&\mathit{assert}(x \neq y)
\end{aligned}
$$

becomes

$$
\begin{aligned}
&\mathit{havoc}(x)\\
&\mathit{assume}(x \notin \mathit{alloc})\\
&\mathit{alloc} = \mathit{alloc} \cup \{x\}\\
&\mathit{havoc}(y)\\
&\mathit{assume}(y \notin \mathit{alloc})\\
&\mathit{alloc} = \mathit{alloc} \cup \{y\}\\
&\mathit{assert}(x \neq y)
\end{aligned}
$$

How allocated set models fresh objects

Original program

$$\begin{aligned}&x = new\ \ C()\\&y = new\ \ C()\\&assert(x \neq y)\end{aligned}$$

becomes

$$\begin{aligned}&havoc(x)\\&assume(x \notin alloc)\\&alloc = alloc \cup \{x\}\\&havoc(y)\\&assume(y \notin alloc)\\&alloc = alloc \cup \{y\}\\&assert(x \neq y)\end{aligned}$$

Renaming variables we obtain:

$$\begin{aligned}&havoc(x)\\&assume(x \notin alloc)\\&alloc_1 = alloc \cup \{x\}\\&havoc(y)\\&assume(y \notin alloc_1)\\&alloc_2 = alloc_1 \cup \{y\}\\&assert(x \neq y)\end{aligned}$$

How allocated set models fresh objects

Original program

$$x = new\ C()$$
$$y = new\ C()$$
$$assert(x \neq y)$$

becomes

$$havoc(x)$$
$$assume(x \notin alloc)$$
$$alloc = alloc \cup \{x\}$$
$$havoc(y)$$
$$assume(y \notin alloc)$$
$$alloc = alloc \cup \{y\}$$
$$assert(x \neq y)$$

Renaming variables we obtain:

$$havoc(x)$$
$$assume(x \notin alloc)$$
$$alloc_1 = alloc \cup \{x\}$$
$$havoc(y)$$
$$assume(y \notin alloc_1)$$
$$alloc_2 = alloc_1 \cup \{y\}$$
$$assert(x \neq y)$$

Assertion holds because

$$alloc_1 = alloc \cup \{x\} \wedge y \notin alloc_1 \ \Rightarrow\ x \neq y$$

Find loop invariant and prove assertion

assume$(\mathsf{N} > 0 \wedge \mathsf{p} > 0 \wedge \mathsf{q} > 0 \wedge \mathsf{p} \neq \mathsf{q})$
a = **new** Array[Object](N)
i = 0
while (i < N) {
   a(i) = **new** Object()
   i = i + 1
}
assert(a(p) != a(q))