CSCI 740 - Programming Language Theory

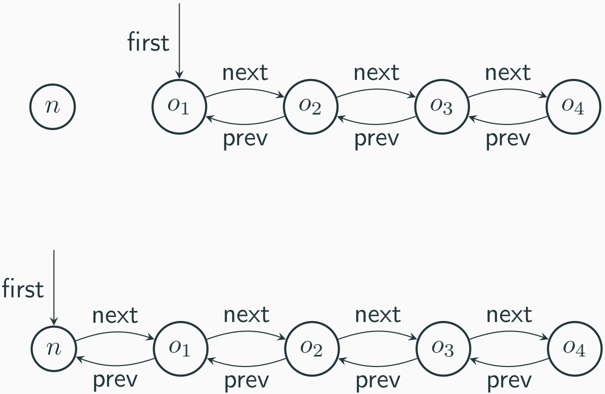
Lecture 25

Verifying Programs with Dynamic Allocation

Instructor: Hossein Hojjat

November 1, 2017

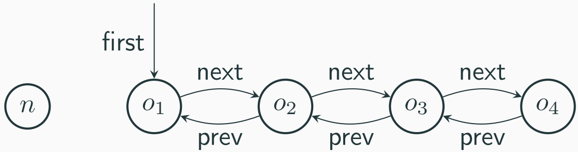
Motivating Example




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

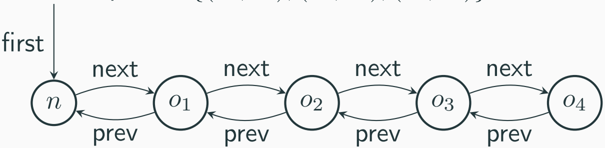
How to verify such code?

Motivating Example



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

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



$$\mathsf{next} = \{(o_1, o_2), (o_2, o_3), (o_3, o_4), \color{red}{(n, o_1)}\}$$

$$\mathsf{prev} = \{(o_2, o_1), (o_3, o_2), (o_4, o_3), \color{red}{(o_1, n)}\}$$

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

Change of relations (partial functions):

$$\mathsf{next}' = \mathsf{next} \cup \{\color{red}{(n, o_1)}\}$$

$$\mathsf{prev}' = \mathsf{prev} \cup \{\color{red}{(o_1, n)}\}$$

using assignments:

$$\mathsf{next} = \mathsf{next}[n \mapsto \mathsf{first}]$$

$$\mathsf{prev} = \mathsf{prev}[\mathsf{first} \mapsto n]$$

Reading Fields

- Statement

```
y = x.next
```

- Computes the value of `y` simply as

```
y = next(x)
```

- We should not de-reference `null`

$$\textbf{assert}(x \neq \texttt{null});$$
$$y = \texttt{next}(x)$$

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

$$\textbf{assert}(x \in \texttt{dom}(\texttt{next}));$$
$$y = \texttt{next}(x)$$

Writing Fields

- We represent each field using a global partial function
- Statement

```
x.next = y
```

- Changes heap according to this update:

$$\texttt{next}' = \texttt{next}[\texttt{x} \mapsto \texttt{y}]$$

- which is a notation that expands to:

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

- We should not assign fields of `null` so we also add this check

$$\textbf{assert}(\texttt{x} \neq \texttt{null});$$
$$\texttt{next}' = \texttt{next}[\texttt{x} \mapsto \texttt{y}]$$

Why we Need Functions?

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

```
var xf = ...
var yf = ...
xf = p
yf = q
assert(xf == p)
```

- 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 $\mathtt{x_f}$ and $\mathtt{y_f}$?
- Does this assertion hold for two distinct values `p`, `q`?

```
var xf = ...
var yf = ...
xf = p
yf = q
assert(xf == p)
```

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

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

Why we Need Functions?

- Say we have `x.f` and `y.f` in the program
- Why not replace them simply with fresh variables $\mathtt{x}_{\mathtt{f}}$ and $\mathtt{y}_{\mathtt{f}}$?
- Does this assertion hold for two distinct values `p`, `q`?

```
var xf = ...
var yf = ...
xf = p
yf = q
assert(xf == p)
```

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

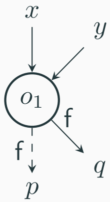
```
...
x.f = p
y.f = q
assert(x.f == p)
```

- Depends.

Aliasing

Does the assertion hold in this case:

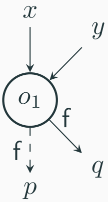
```
x = y
x.f = p
y.f = q
assert(x.f == p)
```



Aliasing

Does the assertion hold in this case:

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

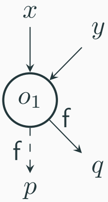


- No! `y` and `x` are aliased references, denote the same object
- Even though left hand sides `x.f` and `y.f` look different, they interfere

Aliasing

Does the assertion hold in this case:

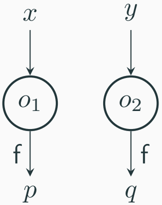
```
x = y
x.f = p
y.f = q
assert(x.f == p)
```



- 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?

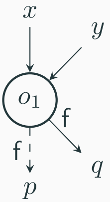
```
assume(x≠y)
x.f = p
y.f = q
assert(x.f == p)
```



Aliasing

Does the assertion hold in this case:

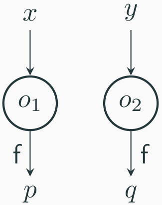
```
x = y
x.f = p
y.f = q
assert(x.f == p)
```



- 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≠y)
x.f = p
y.f = q
assert(x.f == p)
```



Yes!

Example: wp Computation

- Recall $\mathsf{wp}(x := e, Q) = Q[x \mapsto e]$ (substitution)
- Ignoring null checks, we have the following:

$$\begin{aligned}
&\mathsf{wp}(x.f := p; y.f := q\ ,\ x.f = p) = \\
&\mathsf{wp}(f = f[x \mapsto p]; f = f[y \mapsto q]\ ,\ f(x) = p) = \\
&\mathsf{wp}(f = f[x \mapsto p]\ ,\ (f[y \mapsto q])(x) = p) = \\
&((f[x \mapsto p])[y \mapsto q])(x) = p
\end{aligned}$$

- If h is a function then

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

- Thus

$$\begin{aligned}
&((f[x \mapsto p])[y \mapsto q])(x) = p \\
\Leftrightarrow &(x = y \wedge p = q) \vee (x \neq y \wedge p = (f[x \mapsto p])(x)) \\
\Leftrightarrow &(x = y \wedge p = q) \vee (x \neq y \wedge p = p) \\
\Leftrightarrow &(x = y \wedge p = q) \vee x \neq y
\end{aligned}$$

Characterizes precisely the weakest condition under which assertion holds

Exercise

```
class C {
  var f: C
}
```

- Translate into checks and function updates

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

Exercise

```
class C {
  var f: C
}
```

- Translate into checks and function updates

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

Solution.

$$
\begin{array}{l}
\texttt{assume}(\texttt{z} \neq \texttt{null}) \\
\texttt{assume}(\texttt{y} \neq \texttt{null}) \\
\texttt{assume}(\texttt{f(y)} \neq \texttt{null}) \\
\texttt{assume}(\texttt{f(f(y))} \neq \texttt{null}) \\
\texttt{assume}(\texttt{f(x)} \neq \texttt{null}) \\
\texttt{f := f}\Big[\ \texttt{f(x)} \mapsto \big(\texttt{f(z) + f(f(f(y)))}\big)\ \Big]
\end{array}
$$

Modeling Dynamic Allocation

- Can we prove this?

```
x = new  C();
y = new  C();
assert(x≠y);
```

Modeling Dynamic Allocation

- Can we prove this?

```
x = new  C();
y = new  C();
assert(x≠y);
```

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

Modeling Dynamic Allocation

- Can we prove this?

```
x = new  C();
y = new  C();
assert(x≠y);
```

- 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):

```
havoc(x);
assume(x ∉ alloc);
alloc = alloc ∪ {x}
```

`alloc` Set

Original program

```
x = new C();
y = new C();
assert(x≠y);
```

Becomes

```
havoc(x);
assume(x ∉ alloc)
alloc = alloc ∪ {x};
havoc(y);
assume(y ∉ alloc);
alloc = alloc ∪ {y};
assert(x≠y);
```

Renaming variables we obtain:

```
havoc(x);
assume(x ∉ alloc)
alloc₁ = alloc ∪ {x};
havoc(y);
assume(y ∉ alloc₁);
alloc₂ = alloc₁ ∪ {y};
assert(x≠y);
```

Assertion holds because

$$(\mathtt{alloc}_1 = \mathtt{alloc} \cup \{\mathtt{x}\}) \wedge (\mathtt{y} \notin \mathtt{alloc}_1) \Rightarrow \mathtt{x} \neq \mathtt{y}$$