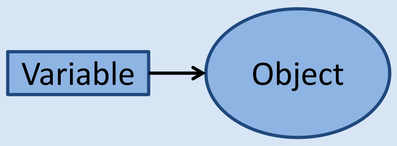
Aliasing Contracts – Untangling Spaghetti References

Janina Voigt, Alan Mycroft
Computer Laboratory, University of Cambridge

The Problem: Spaghetti References

In modern object-oriented programming, a variable stores a reference to an object (a particular memory location).



This way, we can have ***multiple*** variables pointing to the ***same*** object!

Example

At the start of the piece of code on the right, we have the variables x, y, z and a all pointing to some memory location. As the code executes, some variables start to point to the same location as each other; we say they are ***aliased***.

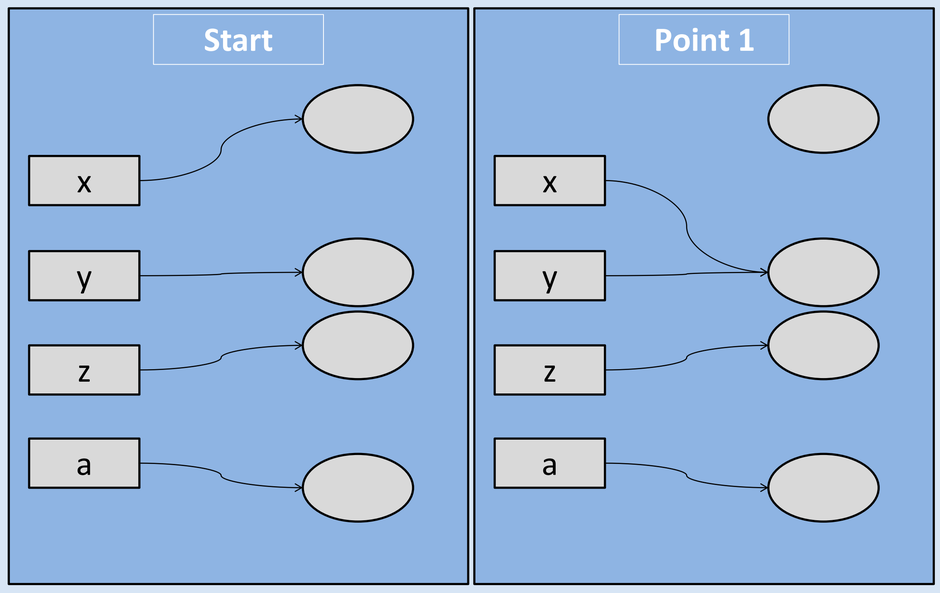
At the end of the code, variables x, z and a all point to the same object.

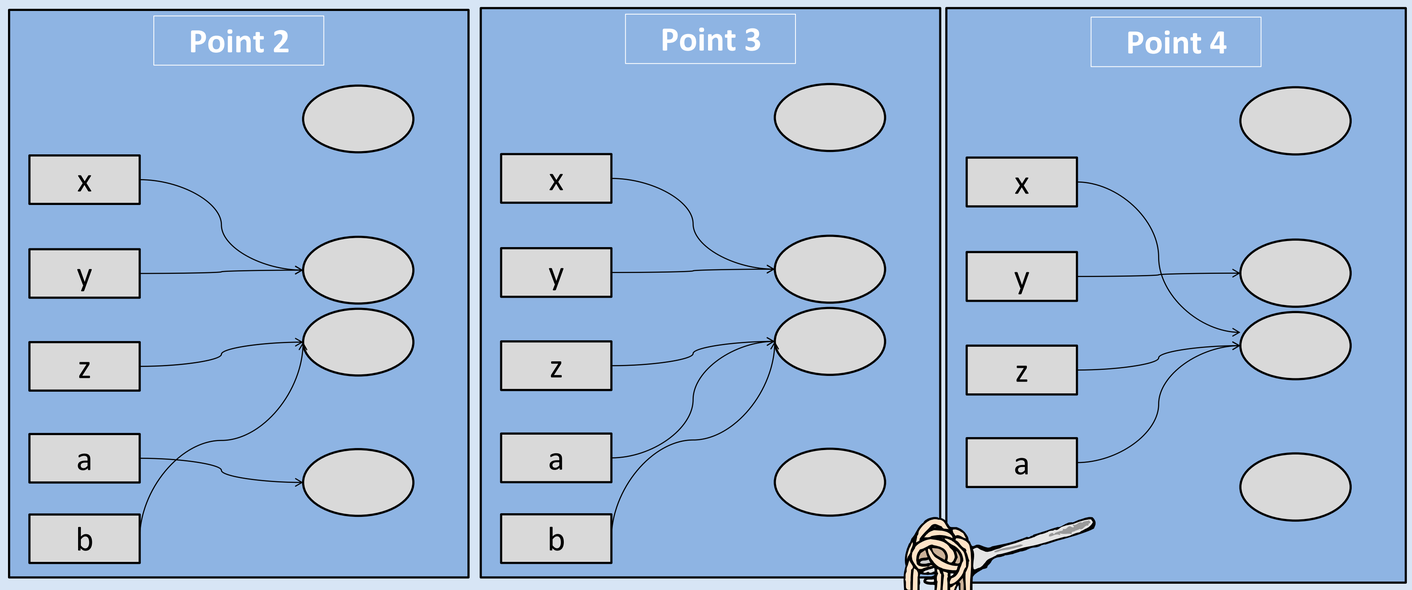
If we now store a value into x, the values in z and a will be affected as well; after all, they all point to the same thing in memory!

```
var x, y, z, a;

void foo() {
   x = y;    (1)
   bar(z);   (2)
   x = z;    (4)
}

void bar(var b) {
   a = b;    (3)
}
```

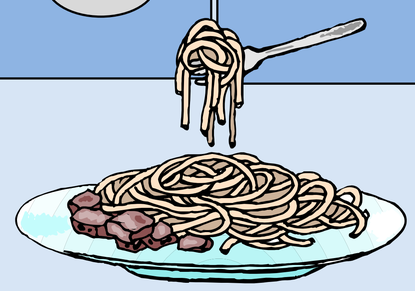




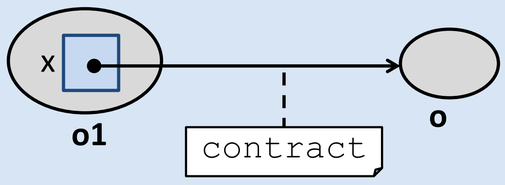
What's the problem?

- It's really ***difficult to see from the code*** which variables point to the same thing, even for our simple example. Imagine what this is like in a complex program!
- Aliasing ***changes all the time*** while the program is executing.
- If two variables are aliased but shouldn't be, we get ***unexpected changes*** because they are connected. This is a common cause of ***bugs***.

We get a very complex aliasing structure, **spaghetti references**!



Proposed Solution: Aliasing Contracts



Contract	Meaning
accessor == this	Only the object holding reference x (object **o1**) can access object **o**.
true	There are no restrictions on accessing **o**: it can always be accessed (if all other contracts for it evaluate to true as well).
false	**o** can never be accessed.
accessor == accessed	**o** can only be accessed by **o** itself.
accessor canaccess this	**o** can only be accessed if the accessor also has the right to access the object holding reference x (object **o1**).

We annotate each variable with a ***boolean*** expression (called an aliasing contract) which we can evaluate to true or false at runtime.

Contracts specify under which circumstances the object ***to which the variable points*** can be accessed.

When an object is accessed, the contracts of ***all variables currently pointing to it*** must be evaluated. If any of these evaluates to false, the access fails.

We use the special variables ***accessor*** and ***accessed*** to refer to the object making the access and the object being accessed.

Contracts don't restrict aliasing itself but ***mitigate the effects of aliasing***. We can specify when an object should be accessible; if we try to illegally access it through an alias, this will give a contract error!

Example: Linked List

In this example, we have a linked list, where each node holds a link to the `next` node. The linked list itself only has a link to the `head` node.

The nodes should only be accessible to the linked list. To enforce this, we use aliasing contracts.

Even if another part of the program now has an alias to a node, it cannot use it for accesses, since this would cause a contract error.

The nodes in our list are now fully protected from the effects of aliasing!