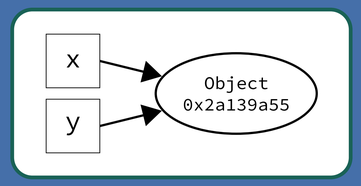
Latte
Lightweight Alias Tracking for Java

Conrad Zimmerman (Brown University)
Joint work with Catarina Gamboa (CMU, ULisboa)
Advised by Jonathan Aldrich (CMU)
and Alcides Fonseca (ULisboa)

Aliasing

Variables (or expressions) x and y are **aliased** if they refer to the same value. For example:

```
Object x = new Object();
Object y = x;
```



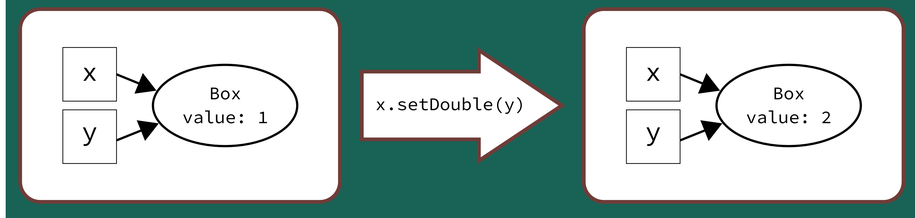
Motivation

Suppose we have the following Java class definition:

```
class Box {
  int value;
  void setDouble(Box other) {
    this.value = other.value * 2;
  }
}
```

Question
Suppose we have variables `Box x` and `Box y`. After executing
`x.setDouble(y)`
does
`x.value == y.value * 2`?

Answer
No! Consider the case when x and y are aliased:



In this case
`x.value == y.value`
therefore
`x.value != y.value * 2`.

Aliasing can cause surprising behavior, even in “simple” cases like this. In this example, aliasing breaks our desired post-condition.

Aliasing makes it difficult or impossible to verify program code, especially when using automatic verification tools like LiquidJava (Gamboa et al. 2023).

Solution

1. Add annotations
Using a special-purpose type system, Latte validates annotations added to field definitions and method parameters. The type system enforces an aliasing invariant, depending on the annotation used:

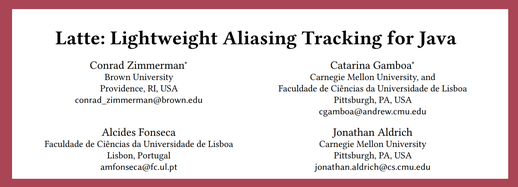
- **`@Unique`**: aliases not allowed
- **`@Shared`**: possibly aliased.
- **`@Owned`** (parameters): aliased with a unique value owned by a caller, but all aliases are inaccessible inside the method, thus it may be treated as a unique value.

2. Infer aliasing for local variables
Latte infers annotations for local variables, and allows aliasing among them when it can be statically determined.

3. Profit
Latte statically determines aliasing of non-shared values, thus tools such as automated verifiers can correctly reason about the effects of mutation.

Latte uses regular Java code, and programmers need to learn only three different annoations. Destructive reads, among other concepts used in similar aliasing systems, can be done using regular Java code.

We plan to add Latte to LiquidJava (Gamboa et al. 2023), which will allow it to correctly verify code involving mutation. Combining Latte with Liquid Types allows verification of Java programs with minimal developer effort.


Latte: Lightweight Aliasing Tracking for Java

Conrad Zimmerman*
Brown University
Providence, RI, USA
conrad_zimmerman@brown.edu

Catarina Gamboa*
Carnegie Mellon University, and
Faculdade de Ciências da Universidade de Lisboa
Pittsburgh, PA, USA
cgamboa@andrew.cmu.edu

Alcides Fonseca
Faculdade de Ciências da Universidade de Lisboa
Lisbon, Portugal
amfonseca@fc.ul.pt

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, PA, USA
jonathan.aldrich@cs.cmu.edu

Submitted to *International Workshop on Aliasing, Confinement, and Ownership* (IWACO) and *Human Aspects of Types and Reasoning Assistants* (HATRA).

Type Checking with Latte

A typing environment Δ stores the current annotation for every variable. Variables are annotated with `unique`, `shared`, owned, `alias(x)` (to track aliasing), or ⊥ (to track inaccessible variables). Formal rules included in paper.

```
class Node {
  @Unique Object value;
  @Unique Node next;
  ...
}

class Stack {
  @Unique Node root;

  void push(@Owned Stack this,
            @Unique Object value) {
    Node r; Node n;
    r = this.root;
    this.root = null;
    n = new Node(value, r);
    this.root = n;
  }
}
```

1. Begin with the method signature:
   `Δ = this: owned Stack, value: unique Object`
2. Declare (uninitialized) variables:
   `Δ = ..., r: ⊥ Node, n : ⊥ Node`
3. Alias `r` and `this.root`:
   `Δ = ..., r: alias(this.root) Node, ...`
4. Clear `this.root`, (this makes `r` unique):
   `Δ = ..., r: unique Node, ...`
5. Create a new Node (this renders `r` and `value` inaccessible since they were passed to `unique` parameters):
   `Δ = ..., value: ⊥ Object, r: ⊥ Node, n: unique Node`
6. Set `this.root`, which aliases `this.root` and `n`:
   `Δ = ..., n: alias(this.root) Node`