# Native Implementation of Mutable Value Semantics


Dimitri Racordon[1], Denys Shabalin[2], Dan Zheng[2], Dave Abrahams[3], Brennan Saeta[2]

[1]University of Geneva, Switzerland

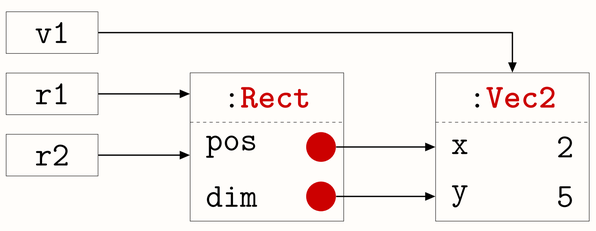
[2]Google, United States [3]Adobe, United States


## "Spooky action at a distance"

Popular programming languages have converged on a common mutation model, where types behave either as **primitive**/**built-in** types or as **aggregate** types.

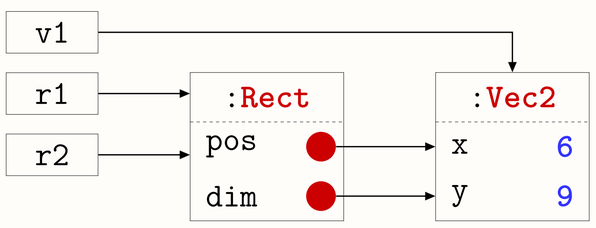
```
class Rect { Vec2 pos, dim; }
class Vec2 { Int x, y; }

var v1 = Vec2(2, 5);
var r1 = Rect(v1, v1);
var r2 = r1;
```




A variable of primitive type always has an **independent** value, but a variable of aggregate type has a value that is **shared** when assigned to another variable.

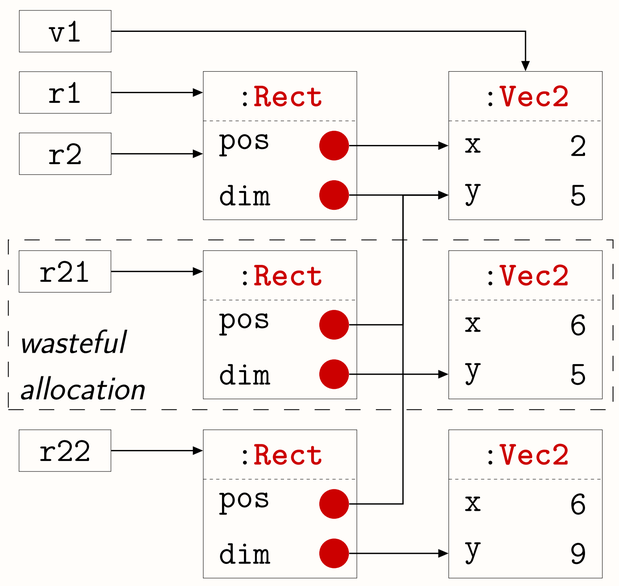
```
r2.dim.x += 4
r2.dim.y += 4
print(r1.pos.x) // Prints 6
```




### *Pure functional programming*

Immutability may fail to capture the programmer's mental model, or prove ill-suited to express and optimize some algorithms. By design, functional updates do not support in-place mutation, demanding optimizer heroics to recover efficiency.

```
var r21 = up_dim(r2,
  up_x(r2.dim, r2.dim.x + 4))
var r22 = up_dim(r21,
  up_y(r21.dim, r21.dim.y + 4))
```

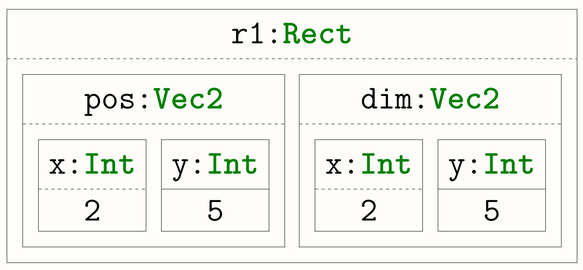



### *Advanced type systems*

Newer programming languages (e.g., Rust) have blended different ideas aimed at taming aliasing into flow-sensitive type systems for more expressiveness. Unfortunately, these approaches have complexity costs that significantly raise the entry barrier for inexperienced developers [3].

## Mutable value semantics

Mutable value semantics bans sharing instead of mutation to support in-place mutation and local reasoning. Conceptually, values are nested boxes, rather than graphs, which are copied when assigned or passed as parameters.




By design, all values are independent: mutating the part of a value has no impact on the value of other variables. Immutability and copy apply transitively, obviating the "shallow" vs "deep" distinction.

### *Mutation across function boundaries*

Values of parameters whose type is annotated by `inout` are written back to their original location at the end of a function call.

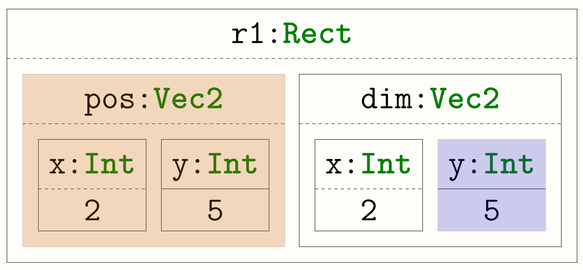
```
void scale_up(inout Vec2 v, Int f) {
  v.x *= f
  v.y *= f
}

scale_up(&r1.dim, 8)
print(r1.dim.x) // Prints 16
```

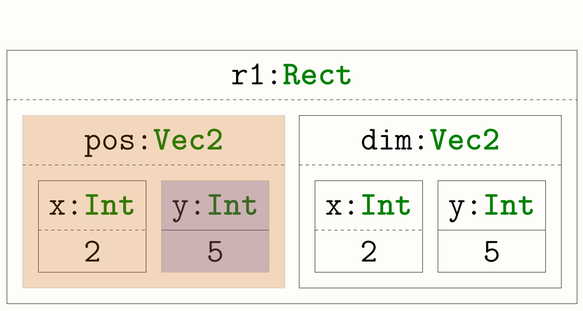
Overlapping mutations are prohibited to prevent any write-back from being discarded.

```
void swap_gt(inout Vec2 v,
             inout Int x) {...}

// Well-typed situation.
swap_gt(&r1.pos, &r1.dim.y)
```




```
// Ill-typed situation.
swap_gt(&r1.pos, &r1.pos.y)
```




### *References*


[1] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams and Brennan Saeta. 2021. Native Implementation of Mutable Value Semantics. *ICOOOLPS@ECOOP*

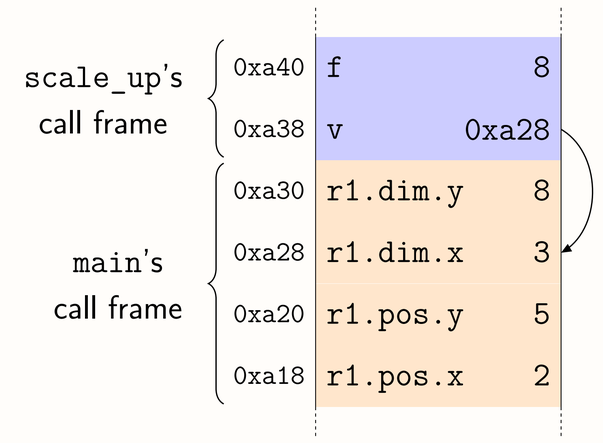
[2] `https://github.com/kyouko-taiga/mvs-calculus`.

[3] `https://blog.rust-lang.org/2017/09/05/Rust-2017-Survey-Results.html`

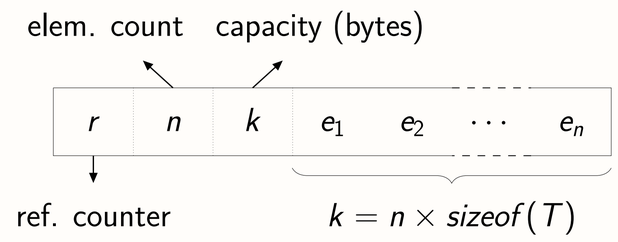

## Implementation

Our implementation relies heavily on stack allocation. Built-in types typically have a 1-to-1 correspondence with machine types. Structures (a.k.a. records) are laid out sequentially.

```
var r1 = Rect(Vec2(2,5), Vec2(3,8))
scale_up(&r1.dim, 8)
```




Structures are exploded into scalar arguments and passed directly through registers, `inout` arguments are passed as (possibly interior) pointers.




An array is a pointer to a block of heap-allocated memory that is reference-counted.

### *Optimizations*

Move semantics: Values are often assigned just after having been created (e.g., `var x = [1, 2]`). In this situation, one can *move* the temporary value into the variable and avoid unnecessary heap allocation.

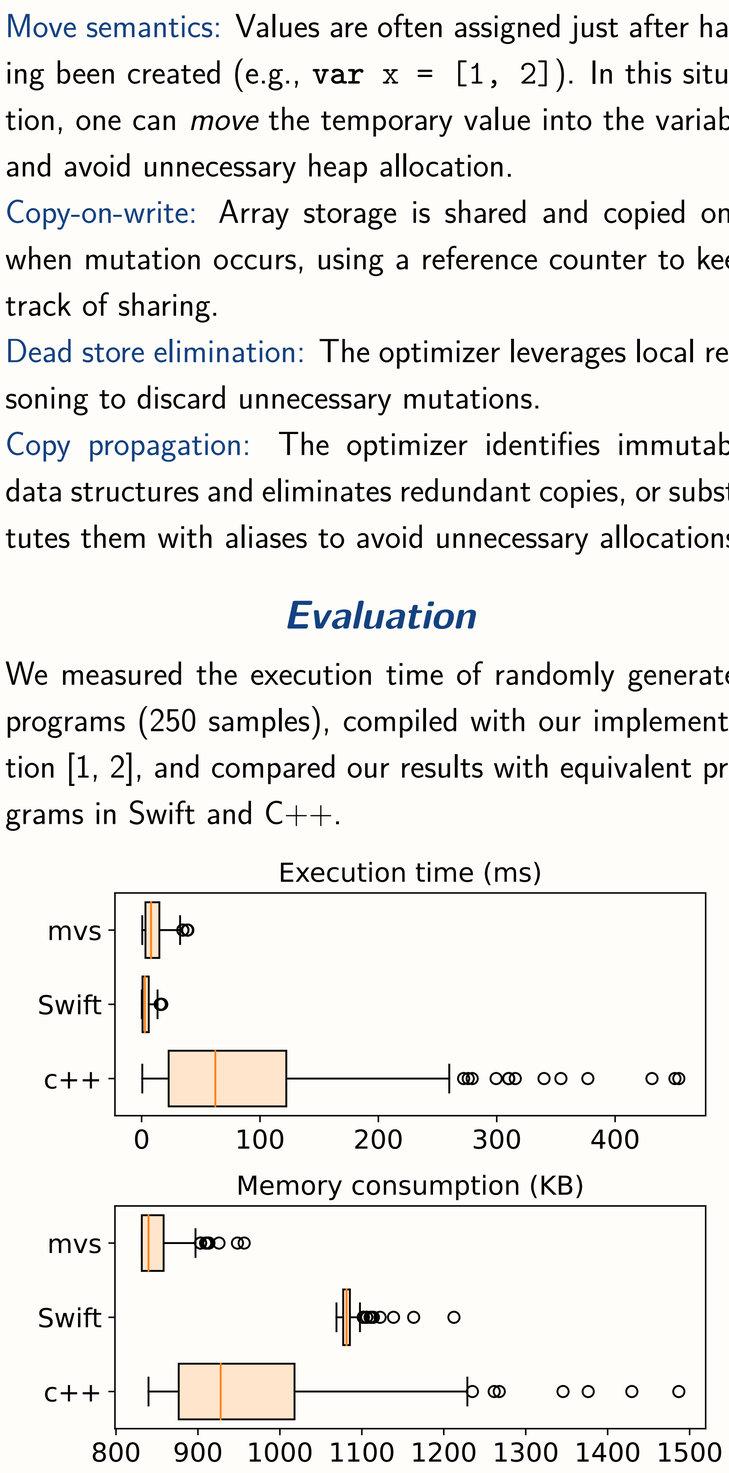
Copy-on-write: Array storage is shared and copied only when mutation occurs, using a reference counter to keep track of sharing.

Dead store elimination: The optimizer leverages local reasoning to discard unnecessary mutations.

Copy propagation: The optimizer identifies immutable data structures and eliminates redundant copies, or substitutes them with aliases to avoid unnecessary allocations.

### *Evaluation*

We measured the execution time of randomly generated programs (250 samples), compiled with our implementation [1, 2], and compared our results with equivalent programs in Swift and C++.





**ECOOP/ISSTA, 11 - 17 July 2021, Online**