The Val Object Model

Document #:	P2676R0
Date:	2022-10-13
Project:	Programming Language C++
Audience:	Evolution

Reply-to:	Dave Abrahams <dabrahams@adobe.com> Sean Parent <sparent@adobe.com> Dimitri Racordon <dimitri.racordon@gmail.com> David Sankel <dsankel@adobe.com>

1 Introduction

This paper presents a low-level programming model that is simple, powerful, efficient, and safe. We believe it could form the basis of a future safe dialect of C++.

2 Motivation and Scope

Software safety is a growing and well-justified concern across government and industry, with a recent Linux foundation plan specifically calling for "moving software away from C and C++ to safer languages." Until now, achieving memory safety in a language like C++ has meant the use of impractical whole-program analysis or the addition of hard-to-satisfy lifetime annotation requirements that significantly increase API complexity. As an alternative, we present an object model—implemented in the Val research language—based on value semantics; an idea already deeply ingrained in C++. It turns out that by "going all in" on value semantics, we gain three things at once:

— Memory safety by construction
— Thread safety by construction (Rust's "fearless concurrency")
— A simple and powerful programming model, with helpful diagnostics

As the C++ committee considers its response to the safety crisis, understanding the Val model could allow us to arrive at a safer C++ that is also simpler.

3 The Val Object Model

3.1 Independence

The Val model starts by identifying **independence**—the idea that a mutation to one variable cannot affect the value of another—as a key property of types with value semantics. Independence is the true source of the benefits functional programmers attribute to strict immutability, and of Rust's "fearless concurrency."

C++ supports independence in three ways:

— Pass-by-value gives the callee an independent value.
— A returned value is independent in the caller (every rvalue is independent).
— Operations such as copying and assignment can be written to avoid sharing mutable state.

Support for independence of mutable user-defined types can be traced back to Ada and Pascal, but for 30 years during the OO revolution, new reference-based language designs flooded the scene leaving C++ in a small club (along with Swift and Rust) of popular languages with first-class value semantics.

Unfortunately, though, C++ also *undermines* independence:

— Mutation occurs through a `this` pointer that can alias other reachable pointers and references.
— Pass-by-value eagerly copies parameters, so programmers use references as a substitute.

The key idea behind the Val model is to fully uphold independence while eliminating disincentivizing copies.

3.2 The Law of Exclusivity for C++ References

The use of pass-by-`const&` to mean "pass-by-value; just do it efficiently" is so ingrained in C++ practice that we do it automatically, even though it affects semantics:

Inefficient

```
// Offsets x by 2*delta.
void offset2(BigNum& x, BigNum delta) {
  x += delta
  x += delta
}

void main() {
  BigNum x = 3;
  offset2(x, x);
  std::cout << x << std::endl; // Prints 9
}
```

Efficiently Incorrect

```
// Offsets x by 2*delta.
void offset2(BigNum& x, BigNum const& delta) {
  x += delta
  x += delta
}

void main() {
  BigNum x = 3;
  offset2(x, x);
  std::cout << x << std::endl; // Prints 12
}
```

The only reasonable way to make the efficient code correct is to add an additional independence requirement, and hope that users uphold it.

```
// Offsets x by 2*delta. Requires: x and delta are distinct objects.
void offset2(BigNum& x, BigNum const& delta) {
  x += delta
  x += delta
}

void main() {
  BigNum x = 3;
  offset2(x, BigNum(x));  // Note explicit copy
  std::cout << x << std::endl; // Prints 9
}
```

Of course, independence requirements are almost never stated explicitly: in practice, there is an unstated **Law of Exclusivity** (LoE) [SE-0176], which requires the value of any object denoted by a mutable reference to be independent of the values of all other variables. We claim this law is built into every C++ programmer's mental model, because there is no other rational way to deal with mutation.

In fact, **the semantics of a mutating function is nearly impossible to describe unless the Law is upheld**. As a result of issues discovered over the course of two decades, there are numerous examples of standard language designed to prevent specific LoE violations (e.g. 27.2 [algorithms.requirements]/7, 24.2.4 [sequence.reqmts]/37), but looking through the standard with the Law in mind, it is still easy to find cases like these:

Legal; undefined behavior in practice

```cpp
#include <algorithm>
#include <vector>

int main() {
  std::vector v = { 0, 1 };
  std::ranges::sort(
    v, [&v](int x, int y) {
         v.push_back(2); return y < x;
       });
}
```

Behavior unspecified

```cpp
#include <algorithm>
#include <vector>
#include <iostream>

int main() {
  std::vector v = { 0, 1, 2, 1, 2 };
  std::ranges::replace(v, v[1], v[2]);
  for (auto x: v) { std::cout << x; }
}
```

The standard says the first one (godbolt) is legal, but it causes undefined behavior in practice. A careful reading shows that the behavior of the second (godbolt) is not specified at all. That's not because the standard library specification is careless, but because the general Law was never recognized.

3.3 Parameter Passing Conventions

Val supports four parameter-passing conventions that underlie its object model. Each can be described either by a simple high-level semantics in terms of owned values, or mechanically in terms of references, but with additional independence guarantees not provided by C++.

Declaration syntax	Semantics	C++ mechanics	Static guarantee/Note
`x: let T` *or* `x: T`	pass-by-`const`-value	`T const&`	Referent is truly immutable
`x: inout T`	borrow for mutation	`T&`	Access to `x`'s value is only via `x`
`x: sink T`	ownership transfer	`T&&` (non-universal)	Callee is responsible for destruction
`x: set T`	initialization	placement `new`	Post: `x` is initialized

The default passing convention is called `let`. It provides the same semantics as C++ pass-by-`const`-value, but without the disincentivizing copy. Inside the callee, `x` is an independent value: it cannot mutate, either directly through `x` or by any other means (such as a hidden mutable pointer or reference).

Mutation in Val is always via `inout`. An `inout` parameter is an independent value wholly owned by the callee and only accessible via the parameter itself. This guarantee is enforced at the call site by disallowing (potentially) overlapping accesses, thus upholding the Law of Exclusivity. *Note that mutations in Val are marked with* `&` at the call site:

```
swap(&x.a, &x.b)      // OK if a and b are guaranteed distinct.
&v.remove(v[2])       // ERROR: v, being mutated, overlaps v[2]. Pass v[2].copy() instead.
swap(&x[i], &x[j])    // ERROR: x[i] and x[j] may overlap; use index_swap to ensure `i != j`.
&x.index_swap(i, j)   // OK; i == j can be checked for dynamically.
```

The `sink` convention denotes consumption of the argument; the difference from C++ pass-by-rvalue-reference is that an lvalue argument becomes inaccessible in the caller, rather than leaving behind a meaningless shell, and responsibility for destruction passes to the callee along with ownership of the value.

```
fun consume(_ x: sink Int) {}
var a = 1, b = 1
consume(a)    // OK, last use of a
consume(b)    // ERROR: b is still in use. Pass b.copy() here instead.
print(b)      // last use of b
```

In Val, the lifetime of a binding extends until its last use, so an lvalue can be sunk without any explicit marking (such as `std::move`). A sunk argument is not actually moved in memory until and unless it escapes the callee (i.e. is stored or returned).

We can ignore the `set` convention for the purposes of this paper. It is seldom used but completes the language calculus in a way we consider valuable. Further details at https://val-lang.dev.

3.4 Explicit copies

You may have noticed diagnostics in the previous section that prompt the user to call `.copy()` explicitly. Because copies can have signficant cost, Val never copies implicitly (unless the user explicitly opts-in with `@implicitcopy`). Because pass-by-value semantics does not imply a copy, though, the number of explicit copies required in Val code is small. Finally, in case the idea of explicitly copying an integer is anathema to the reader, it should be noted that explicit copies are not fundamental to the model.

3.5 Bindings

Bindings, which play the same role as variables in C++, have three forms:

Declaration	Convention	Static guarantee/Note
`var x =` *expr*	`sink`	`x` is a variable. Any lvalue *expr* becomes inaccessible forever.
`let x =` *expr*	`let`	`x` (and any lvalue *expr*) are truly immutable during `x`'s lifetime. No copy implied.
`inout x =` *expr*	`inout`	*expr* (and anything of which it is a part) is inaccessible during `x`'s lifetime.

Each form can be understood as though the binding were a parameter to a continuation formed by the code through the binding's last use, with a corresponding parameter passing convention. During the lifetime of a `let` or `inout` binding to (part of) an lvalue, use of that lvalue is restricted as necessary to preserve independence:

```
fun use<T>(_ x: T) {}

fun test(_ inout x: (first: Int, second: Int)) { // x is a tuple with two named fields.
  let y = x.first
  print(x)          // OK
  x.first += 1      // ERROR: `x.first` is `let`-bound to `y`.
  use(y)

  inout z = x.first
  z += 1            // OK
  print(x)          // ERROR: `x` is `inout`-bound to `z`.
  use(z)

  print(x)          // OK; `x` no longer `inout`-bound.
}
```

Because the semantics of these bindings are based on those of our parameter passing conventions, they have the semantics of independent values, just like parameters.

3.6 Projections: Abstracting Partwise Access

A **projection** is an abstraction for access to a part of an object. For example, you might want to provide read/write access to the top element of a stack. To support that sort of access without exposing references, Val provides projections in two forms: subscripts and computed properties (which can be thought of as subscripts without parameters).

```
type Stack<T> {
  var storage: Array<T> = []
  public fun push(_ x: T) inout { &storage.append(x) }
  public fun pop() -> T inout { return &storage.removeLast() }

  public property top: T {
    inout { yield &storage[storage.count() - 1] }
  }
}

fun test() {
  var x = Stack<Int>()
  &x.push(3)
  &x.push(4)
  &x.top += 1  // <====
  print(x.top) // 5
}
```

The `top` property works without exposing references by using *inversion of control*: where the implementation `yield`s, the last element of storage is passed for mutation to a closure (lambda) representing the mutation, which is synthesized to encode the mutation at the marked line:

```
  // rewritten accessor
  public fun top:inout(_ mutate: (inout T)->Void) {
    mutate(&storage[storage.count() - 1]) // Explicit inversion of control
  }
  ...
  // rewritten call site
  x.top:inout((y){ y += 1 })
```

The idea of combining inversion of control with this syntactic transformation is due to John McCall of the Swift project. Although it (still!) isn't part of the official Swift language, the provisional unofficial support for it is widely used for reasons that should become clear.

3.6.1 Projection Accessors

Aside from an `inout` accessor, there are three other projection accessor types:

Accessor	Meaning	Result lifetime
`inout`	yield a part of `self` for mutation	bounded by lifetime of `self`
`let`	yield a part of `self` for reading	bounded by lifetime of `self`
`set`	write to `self` without exposing the previous value	n/a
`sink`	consume `self`, exposing a part	independent

In the case of `Stack`'s `top` property, these accessors are all synthesized from the `inout` accessor, but in general each one can be written separately, when there is a performance advantage. If `inout` and `set` are omitted but `let` is provided, the projection is immutable regardless of the mutability of `self`.

3.6.2 Projecting Ephemeral Notional Parts

The inversion of control provided by `inout` and `let` accessors allows us to project *notional* parts that aren't actually stored anywhere in memory. For example, this `Angle` type stores its value in radians, but also exposes a value in degrees that can be `let` bound or passed `inout` like any other mutable property:

```
type Angle {
  public var radians: Double
  public property degrees: Double {
    inout {
       var d = radians * 180.0 / Double.pi
       yield &d
       radians = d * Double.pi / 180.0
    }
    set(newValue) {
      radians = newValue * Double.pi / 180.0
    }
  }
}

fun test() {
  var x: Angle(radians: Double.pi / 2)
  print(x.degrees)      // 90
  var y: Double = 180
  swap(&y, &x.degrees)
  print(x.radians)      // 3.14159265...
  x.degrees = 0         // uses the `set` accessor
  print(x.radians)      // 0
}
```

This is an example where an explicit accessor can beat a synthesized one. If `degrees` did not have an explicit `set` accessor, it would be synthesized from the `inout` accessor, and the line `x.degrees = 0` would compute the existing value in degrees, only to immediately assign over it.

Because `inout` and `let` projections have partwise access semantics but don't require pre-existing storage, they perfectly match any use case where resources of an object need to be temporarily and safely used in a different context. This covers many cases that in Rust would require difficult-to-use named lifetime annotations. A less general but perhaps more recognizable set of use cases correspond to places we'd be tempted to use proxy references in C++, if only they didn't cause so many problems:

- A `ZipCollection` containing tuples of projected elements can be projected from a tuple of underlying collections.
- A matrix in row-major storage can project its columns as a property.
- A collection type `C` can have a slicing subscript that safely projects a region as an ephemeral `Slice<C>`. The slice contains the region's bounds and a projection of the collection itself.
- A collection can be projected into a pair of independent `Slice`s that divide it at some position and can be mutated in distinct threads.
- A `Dictionary<Key, Value>` where keys, values, and liveness bits are stored in contiguous memory regions can be subscripted with a `Key`, projecting a writable ephemeral `Optional<Value>` (writing `nil` deletes the key). Due to inversion of control, in-place mutation of a subscripted dictionary uses only one hash lookup where a getter/setter formulation would require two.

3.7 When and How to "Go Unsafe"

So far we've been looking only at Val's safe subset, which can't be used to efficiently express every correct program. Val also contains unsafe constructs accessible only by tagging the enclosing expression with the `unsafe` keyword. That makes `Val` a safe-by-default language in the same category as Rust. [1] The usual way to use unsafe code in such languages is to encapsulate it in a carefully-vetted component that can be declared safe. For example, a doubly-linked list can be implemented in Rust using only safe constructs, but if the list code is correct you will pay for needless dynamic checks that never fail. Instead, you can reimplement the list with carefully-used unsafe

[1] Swift is safe-by-default within a thread, but uses a higher-level actor model for inter-thread data isolation.

operations, and after extensive validation, declare it safe. The new list is no less safe than any safe primitive, which has likewise been extensively validated and declared safe.

Depending on the details of a programming language design, different programs may be forced to use unsafe constructs or accept dynamic safety checks. We could make it possible to express more things safely in Val were we to add Rust-style named lifetime annotations. These annotations, however, add cognitive overhead and are so notoriously difficult to use that "fighting the borrow checker" has become a meme among Rust programmers. At the same time, because Val exploits the semantics of whole/part relationships, we believe it can safely express far more than lifetime-annotation-free Rust can.

4 Comparison with the C++ Core Guidelines

The work most closely related to Val's object model in the landscape of C++ safety is Stroustrup and Sutter's C++ Core Guidelines [CoreGuidelines]. This project captures common wisdom on ways to "use modern C++ effectively" and safely in the form of rules enforceable through static analysis and dynamic checks.

Some things are simple: avoid unsafe casts and type punning for type safety, and use cheeap dynamic checks to ensure accesses are in-bounds. Lifetime safety, however, requires a more sophisticated approach.

The C++ Core Guidelines rely on an ownership discipline for reasoning about the lifetime of an object. An *owner* is an object that owns another object (e.g. an instance of `std::vector` owns the elements that it contains), and the owner is solely responsible for the destruction of its owned object(s). Preventing code from destroying things it doesn't own eliminates a large class of errors.

To keep the annotation burden at a minimum, rules have carefully designed defaults that correspond to most valid use of modern C++. It is worth noting that many of these defaults seem to express the same ideas expressed by Val's part projection semantics. The patterns that do not fit these defaults can be expressed by user annotations. We refer the interested reader to [P1179R1] for the complete specification.

We note that the ownership discipline proposed by the C++ Core Guidelines upholds the Law of Exclusivity (although it goes overboard, forbidding even immutable references from aliasing one another). This rule effectively eliminates reference semantics.

In Val, ownership is conveyed through whole/part relationships. For example, a vector (i.e., a dynamic array in Val parlance parlance) is naturally a composition of its parts: the elements. A whole is responsible for the lifetime of its parts, just like an owner is responsible for that of its owned objects. Further, mutating a whole requires its independence, just like passing an owner as an argument prevents its contents from being simultaneously passed by reference.

To illustrate, consider the type declaration below, which follows the C++ Core Guidelines. The default rules for identifying owners do not apply in this example, requiring the a user-defined annotation to declare `Matrix3` as an owner type of `double`.

```
class [[gsl::Owner<double>]] Matrix3 {
  double components[9];

public:
  double& operator()(size_t row, size_t column) {
    if (row > 8 || column > 8) {
      throw out_of_range("component position out of bounds");
    }
    return components[row * 3 + column];
  }
};
```

*Note that the operator declaration is implicitly annotated by `[[gsl::post(lifetime(ret,{this}))]]` to denote the fact that the lifetime of the returned reference is bound by that of `*this`.

In Val, annotations are unnecessary because the whole/part relationship between a matrix and its components can be derived from the type's declaration.

```
type Matrix3 {
  var components: Double[9]

  public subscript(_ row: Int, _ column: Int): Double {
    inout {
      precondition((0 ..< 9).contains(row) && (0 ..< 9).contains(column))
      yield &components[row * 3 + column]
    }
  }
}
```

Without first-class pointers and references in Val's safe subset, reasoning about whole/part relationships is sufficient to achieve similar expressiveness as ownership-based approaches, without an ad-hoc ownership model. The definition of a whole is unambiguous in a world governed by mutable value semantics. In contrast, the definition of an owner in a world of references is orthogonal and it compels the user to teach the compiler or static analyzer about the distinctions between owners and non-owners. While the category to which a type belongs might be sometimes inferred, other cases must resort to annotations.

Although the C++ Core Guidelines do not claim complete memory safety and do not address thread safety yet, they take the same basic approach as Rust, which already offers statically guaranteed memory and thread safety by construction. Since they already have a Law of Exclusivity, which is a sufficient basis for both kinds of safety, we are confident they can reach these goals through iterative refinement.

Val's safe subset is nearly as expressive as Rust's; the main difference being that in Val, one cannot "reseat" a `let` or `inout` binding (i.e., a reference in Rust) to make it refer to a different object. We beleive this limitation has very little impact in practice, as reseatable bindings can expressed with key paths [SE-0161]. Hence, using Rust as a proxy for a future more complete version of the C++ Core Guidlines, we can predict that Val's expressiveness is on par with the latter.

5 Conclusion

We believe the Val object model is in a "sweet spot" of simplicity and expressive power due to the combination of a few powerful ideas:

- The Law of Exclusivity ensures independence, the foundation of value semantics.
- Projections allow access to parts, including ephemeral ones, without introducing references.
- It's not worth trying to capture every possible thing in the type system. Accepting localized, reviewed use of unsafe constructs, or dynamic safety checking, may be better than adding language features to statically prove every program safe.

Adopting this model may not be the right thing for C++, but we hope the insights we've shared can inspire some fresh thinking about safety.

6 References

[CoreGuidelines] Bjarne Stroustrup and Herb Sutter. 2022. C++ Core Guidelines.
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[P1179R1] Herb Sutter. 2018. Lifetime safety: Preventing common dangling.
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1179r1.pdf

[SE-0161] David Smith, Micheal LeHew, and Joe Groff. 2017. Key Paths.
[SE-0176] John McCall. 2017. Enforce Exclusive Access to Memory.