CS349 Essay: Oxynium

Generics, Immutability, and the False Dichotomy Between Abstraction and Performance

u5502784

December 14, 2025

Abstract

This report explores the trade-off between high-level abstraction and runtime performance in programming language design, using Oxynium as a case study - a language created by the author. Two features of Oxynium are examined: generics and immutability. Firstly, Oxynium's type erasure model of generics is compared to the reification strategies proposed by Kennedy and Syme. Secondly, Oxynium's strict, transitive immutability is assessed against the immutability classification framework proposed by Coblenz et al..

While Oxynium succeeds in providing a high-level syntax, it lacks the zero-cost abstractions found in mature languages like Rust and C#, which improve runtime performance. To conclude, the false dichotomy between abstraction and runtime performance can only be resolved by increasing compiler complexity and reducing compiletime performance. This trend can be seen in the current PL landscape.

Introduction

Traditionally, programming languages are defined by their placement along a single dimensional scale from low-level (machine code) to high-level (natural language). High-level languages such as Python, JavaScript and Ruby are slower, interpreted, and have a higher level of abstraction. Low-level languages such as Assembly, C, Zig and C++ are faster [6], compiled, and have a lower level of abstraction. Further, lower level languages tend to have more complex syntax [10], relying on language features such as the '&' and '*' operators to help with memory management, operations managed by the compiler for higher level languages. Oxynium is a case study in the challenges of creating high-level language which also has excellent runtime performance, by sacrificing compile time performance.

This report critically analyses the design decisions of two key features of Oxynium: generics and immutability. Firstly, Oxynium's implementation of generics is contrasted to that proposed by Kennedy and Syme for the .NET Common Language Runtime [4]. Secondly, the strict immutability of Oxynium is examined against the findings of Coblenz et al. [2]. This report explores the crucial trichotomy of compiler implementation effort, runtime performance, and developer expressiveness when using the language. It additionally argues that modern compilers trend towards increasing all three: more complex compilers [1] and longer compilation times, in exchange for increased 'zero-cost' abstraction and better runtime performance.

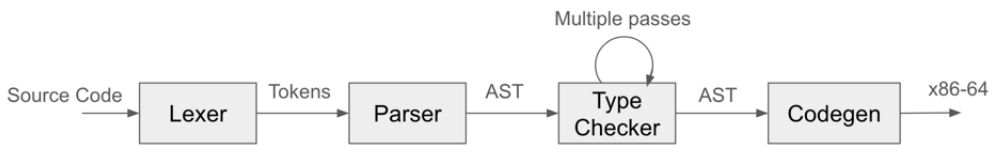
Language Overview

I started the Oxynium Project in 2022, and have continued development since. Oxynium's compiler is implemented in Rust and is open source: `https://github.com/revers3ntropy/oxynium`. The compiler is still under development, but you can find out more at `https:`

//oxynium.org. Oxynium's features include C++ style classes, explicit Zig-like error handling, Pythonic operator overloading, first-class functions, and generics similar to TypeScript.

```
// C-style macro to import another file.
#include "another_source_file.oxy"
class Something {
    my_attribute: Int,
    // A static method does not take ‘self’ as the first parameter.
    // Methods implicitly return using ‘->’ operator.
    // Oxynium does not have constructors (similarly to Go)
    // so static ‘make’ functions are the equivalent by convention.
    def make(a: Int) ->
        new Something { my_attribute: a },
    // Override the ‘+’ operator when used on an instance of this class.
    def + (self, b: Something) Something {
        return Something.make(self.my_attribute + b.my_attribute);
    }
    // Instance methods defined with ‘self’ as the first parameter.
    def my_method(self) {
        // Use ’.Type()’ to cast to that type.
        // All casts are explicit.
        print(self.my_attribute.Str());
    }
    def Str(self) ->
        "Something(" + self.my_attribute.Str() + ")"
}
def main () {
    let s = Something.make(9);
    // prints ‘Something(9)’
    print(s.Str());
    // prints ‘12’
    (s + Something.make(3)).my_method();
}
```

The Oxynium compiler follows a typical compiler pipeline, generating x86-64 Assembly as the intermediate representation:



A native executable can be generated using the Oxynium toolchain, which uses GCC[9] as the backend. Optimisation passes are only done over this generated assembly: the compiler has no AST-level optimisation passes. Macro expansion occurs between parsing and type checking, and modifies the AST directly. Oxynium has no C-style object-like macros, only function-like [3].

Personal Experience

I attempt to follow the practice of ’dogfooding’ while developing Oxynium, which means continually using my own product for testing, ideally very similarly to a typical developer writing in Oxynium. However, as the designer of Oxynium, my experience will always differ from a typical user. The vast majority of Oxynium code I have written has been for testing (over 1000 individual tests), and the Oxynium standard library (STD). In both cases, the software tends to be short and independent, without interaction between components. Although it has given me insight into the usability of the syntax and basic language usage, its practicality for large software projects is untested. Creating the Oxynium 2 self-hosted compiler will be a key experiment in the usability of the language, however, this is still in the very early stages of development. The immature STD limits the usability of language - for example, without a battle-hardened web server library, I would not feel comfortable exposing a server written in Oxynium to the internet as it would likely be insecure [12].

Writing tests generally consists of producing minimal examples to isolate a particular feature. On the other hand, the STD aims to be as abstract as possible, covering as many use cases as possible. An example of keeping the STD general is the 'Result' class, explored later, which is the foundation for error handling in Oxynium. This class stores either an 'expected' value, or an 'error'. In other words, it is the sum of two types. In Oxynium, this can be implemented using a generic class: similar to C++ templates, 'Result' is parametrically polymorphic over its type parameters, declared with the familiar angle-brace syntax. This 'Result' class is based on the implementation of the same class in Rust, which also uses generics.

Oxynium Type System: Generics

The Oxynium type system is both strong and static. This means that objects declared as one type must be explicitly cast into other types, and will always stay as that type. There are two escape hatches from this strong type system, which are used sparingly throughout the STD. The first is the assembly injection macro `#asm` which is generic, taking a type parameter as the first argument. The second is the macro `#unchecked_cast`, which similarly takes a target type, and bypasses the type-checker without modifying the underlying data.

Type checking is done with multiple passes, progressively resolving unknown types into concrete types as more types are 'discovered', allowing arbitrary declaration order without forward declaration. This is inefficient and could just use two passes rather than an unknown number, but allows for generics to be implemented more easily. An example of a generic class with a generic method, in Oxynium:

```
class MyGenericClass<T> {
    // Is able to reference the classes generic parameter and its own.
    def generic_method<A, B, C>(self, t: T, a: A, b: B) C {
        ...
    }
}
// The !<> operator specifies generic parameters,
// which is similar to the Rust Turbofish operator ::<>.
// Usage of MyGenericClass:
(new MyGenericClass<Int>).generic_method!<Str, Bool, Void>(1, "hi", true)
```

The design goal of generics in Oxynium was to feel close to TypeScript generics.

In TypeScript, generics are simply erased from the code during compilation and no additional code is generated at runtime for generic functions and classes, as all polymorphic runtime behaviour is handled by the JavaScript engine. Most compiled languages use either this 'type erasure' approach, or the alternative approach of 'reification', normally implemented as 'monomorphisation' (defined below). A third approach 'code-sharing' is a type of reification - whereas monomorphisation is 'static', code-sharing is dynamic. Kennedy and Syme[4] discuss a hybrid approach, which selects which strategy to use at runtime based on the generic types provided. I find this paper very trustworthy, as this was the approach chosen to be implemented in the public release version of C# .NET, one of the most popular programming languages ever created. This strategy works very well in C# as the required type information for dynamic reification is already provided by the runtime, where most compiled languages would face a runtime penalty by implementing this approach.

In static reification / monomorphisation (the approach most compiled languages such as D, C++, and Rust use) generic classes and functions are 'templates' for concrete classes. The compiler finds all usages of the generic class and duplicates the class, replacing the generic types with the arguments provided in the type. This leads to code bloat, because the definition for classes may be duplicated many times for a large codebase, but allows type information to be preserved at runtime as code may be different for each concrete implementation of the generic interface.

Oxynium uses an alternative - type erasure - prioritising bundle size over expressiveness. Type erasure, unlike reification, strips all generic type information during compilation, so generic functions and classes aren't 'aware' at runtime that they were generic at compiletime. I chose this approach - as the developer of Oxynium - due to its similarity with TypeScript and ease of implementation. Java, TypeScript and Python all use type erasure for their generics; notice that these are all dynamically typed, with type information preserved at runtime by default. In type erasure, generic classes (and functions) are not duplicated, but expected to handle every type. Therefore, when a variable of a generic type is used, runtime type information must be queried to determine how to use that variable, similarly to the Rust 'dyn' keyword. Additional type information increases runtime memory usage and querying this type information decreases runtime execution speed of generic methods - however, the generated code is smaller.

The third strategy - code-sharing - is similar to type erasure in that definitions are not duplicated. It typically involved passing a hidden 'type information' parameter to all generic functions, so functions have information about their generic parameters at runtime, allowing increased expressiveness. For example, a function under generic parameter `T` could only allocate a new object of type T '`new T`' in code-sharing languages, and not in languages such as Oxynium, Rust, TypeScript and C++.

When Microsoft was implementing generics for the C# .NET runtime, they realised none of these methods were suitable. Monomorphisation is incompatible with code generated from previous versions of the compiler (a key requirement) but also due to the dynamic linking performed by the .NET runtime. Code-sharing requires expensive 'boxing' and 'unboxing' which requires heap allocations - this is very slow. Oxynium's approach of type erasure was deemed to limit code expressiveness too much.

So Kennedy and Syme [4] proposed a new idea: use both code-sharing and monomorphisation, and decide the strategy at runtime. While using code-sharing allows for both performance and expressiveness, attempting to use it for types with differing runtime shapes requires 'boxing' all types to the same shape, adding runtime overhead. Instead, code-sharing can be chosen for all pointer types which must have the same shape, and for primitive types such as 'int', 'float', 'bool', use monomorphisation. This limits the size of the ever-growing executable file size when using only monomorphisation, while keeping the expressiveness of reification. As C# already provides runtime type information for every object, the increased memory overhead is minimal.

Oxynium uses pure type erasure, so generic types are treated as black boxes - no type information is provided at runtime. Although avoiding the executable file size growth issue, this massively limits expressiveness, as generic types can only be interacted with in limited ways. For example, objects of a generic type cannot be instantiated using type erasure: `new T` can only work if the type information is somehow preserved at runtime through reification, either by adding type metadata to every function call (code-sharing) or duplicating the code for each type (monomorphisation).

Another example of a runtime feature that reification allows is selection based on the generic type. Through monomorphisation in Rust, this looks like generic constraining: `impl A for B<C>`. Through code-sharing in C#, this would be `if (a instanceof T)`. This form of polymorphism may sometimes be better implemented as a method on the generic type if it is constrained to a single interface (a common pattern that can be refactored by calling a method that is implemented differently for classes which inherit from the same interface), but in cases where this is not possible (for example a generic class over built-in types) then this feature can allow succinct and well-optimised code.

Modern languages tend towards a hybrid approach similar to that proposed by Kennedy and Syme. For example, Go uses GCShape Stenciling [7] to use the same monomorphisation of a generic class for types with the same data layout. While type erasure and monomorphisation maximise runtime performance and code-sharing maximises expressiveness (and so abstraction), the hybrid approach increases the complexity of the compiler in order to increase runtime speed and abstraction, a clear example of the movement towards 'zero-cost abstractions' (explored further later).

I find using generics in Oxynium intuitive, but unexpressive. In particular, when writing the STD, the type system escape hatches were required for many generic classes. They fulfil the design goal of imitating TypeScript, however due to the lack of dynamic typing and runtime type information in Oxynium, they are often clunky to use. See the implementation of the 'Result' class mentioned previously:

```
class Result <T, E> {
    value: Int,
    // is 'value' a T or an E?
    ok: Bool,
    // construct a Result to hold an E
    def err <Val, Err> (err: Err) -> new Result<Val, Err> {
        // as every type in Oxynium is the same size,
        // cast the value into an Int in order to store an E in the same
        // memory location that a T could be stored
        value: #unchecked_cast(Int, err),
        ok: false
    },
    // construct a Result to hold a T
    def ok <Val, Err> (val: Val) -> new Result<Val, Err> {
        value: #unchecked_cast(Int, val),
        ok: true
    },
    // convert expected error E to unexpected error using panic
    // or take the T
    def unwrap (self) T {
        if self.ok ->
            return #unchecked_cast(T, self.value)
        panic("tried to unwrap a Result that was an error")
    }
}
```

In order to simulate a union type, an additional field is required on every instance to determine which type the object stores, effectively adding runtime type information. The equivilent class is implemented in Rust with the following:

```
enum Result<T, E> {
   Ok(T),
   Err(E),
}
```

More expressiveness is required for generics to feel natural to use: generic parameter constraints, native support for enums and sum types, and interfaces.

In languages where type information is already present on objects at runtime, type erasure adds no overhead while maintaining expressiveness in using variables of a generic type. However, in Oxynium the type information is not present; to preserve efficiency in both time and space, monomorphisation may be a better approach for Oxynium to allow more expressiveness.

Oxynium as a Functional Language: First-Class Functions, Immutability

Oxynium, similarly to Rust, makes local bindings immutable by default. Where Oxynium differs from Rust, TypeScript and other multi-paradigm languages is the immutability of class properties.

As discussed by Coblenz et al. [2] this deep, 'transitive' immutability has many theorised advantages such as bug prevention and thread safety. Coblenz et al. argues the status quo terminology used to describe immutability is imprecise and leads to confusion. For example the 'const' keyword in C++ does not guarantee an object or its children are immutable, and is unable to guarantee thread safety. They propose a new classification system for language-level restrictions on state modification, with 8 features:

1. **Type**: Objects in Oxynium are immutable, the strongest Type of restriction a system can have. However as local variables may be declared as mutable which allows them to be reassigned, this feature has 'assignability'. Other options are read-only restriction, and ownership.

2. **Scope**: Oxynium is class-based, as all instances of every class are immutable. Instance-based languages only allow specific instances of a class to be immutable.

3. **Transitivity**: Oxynium has transitive immutability, as objects at every depth of an object graph are immutable. JavaScript's `Object.freeze` is an example of non-transitive immutability, as it does not prevent modification to the object's children.

4. **Initialization**: Oxynium enforces immutability even during initialisation, as every field must be set exactly once, and null defaults are not permitted. Some languages relax mutation restrictions while an object is being constructed, allowing cyclical data structures to be built.

5. **Abstraction**: Oxynium enforces 'bitwise' or concrete state immutability, preventing even internal state like caches or splay trees from being modified.

6. **Backwards Compatibility**: Oxynium has open-world immutability, assuming that external code such as `extern` functions or `#asm` calls may not enforce the same immutability restrictions.

7. **Enforcement**: Oxynium checks immutability violations at compile time, so has static enforcement.

8. **Polymorphism**: Oxynium does not have mutable state, so this field does not apply. If a language did not allow mutable values to be assigned to immutable variables then it would be non-polymorphic. Non-polymorphic languages treat mutable and non-mutable instances of the same class as different types.

The paper then goes on to interview eight expert software engineers in order to understand state management issues in the real world. These interviews involved little quantative data to back up the claims made by the experts, rendering their validity questionable. On top of this, the sample size used (8) is small, and likely suffers from confirmation bias as the authors are more likely to pick subjects with similar experiences to themselves, with little effort made to increase representativeness across all SWEs. Particularly, the confusion felt by novice programmers was completely omitted from the study. However, the interviews were detailed, and grounded in examples, and did not focus on the individual's feelings.

The consensus among the participants was that immutability would be useful as a language feature due to the high number of bugs caused by incorrect state changes, and that immutability was generally not well supported in common programming languages. In order to address this in Java, the authors proposed a library IGJ-T which enforced transitive immutability, and found that it prevented bugs compared to the non-transitive immutability in IGJ, in a pilot study. Oxynium's strict, transitive immutability therefore aligns with the findings of this paper on improving developer experience.

As well as immutability, Oxynium also has other features promoting functional code. Not only are functions first-class citizens in Oxynium, but a concise syntax for lambda functions promotes their use.

```
// mappers must have the signature 'fn (num: T, index: Int) T'
def double_mapper(num: Int, _index: Int) Int {
    return num * 2;
}
def main() {
    let my_list = range(5).List();
    let doubled = my_list.map!<Int>(double_mapper);
    let _same_as_doubled = my_list.map!<Int>(
        // lambda function shorthand for 'double_mapper'
        fn (a: Int, _: Int) -> a * 2
    );
    // 'for' loop over an iterator
    for i in doubled {
        print(i.Str()); // 0, 2, 4, 6, 8
    }
}
```

In order for this highly abstract code to be compiled into performant machine code, zero-cost abstractions are required. This is the cornerstone of the abstraction-runtime-compiletime trade-off: ensuring highly abstract code is transformed into the most efficient equivalent machine code, re-writing it completely where necessary. As machine code is inherently imperative, declarative code is often highly inefficient.

Sophisticated compilers similarly transform complex immutable code into imperative code which utilises the mutability of memory and registers to be more efficient at runtime. By increasing the complexity of the compiler and so the compilation time, high level code can be written which is as efficient as hand-crafted assembly.

Oxynium unfortunately has few of these optimisations, meaning highly abstract code does not perform as well as lower level imperative code. By enforcing restricting such as transitive immutability, these optimisations are easier to implement, but this is still the key area I will need to focus on in order to achieve the goal of a high level performant language, and requires many complex optimisation steps to achieve well.

To demonstrate this, I ran my own benchmarks with the following Oxynium and equivalent Rust code, then took an average of 5 runs:

```
// Rust Benchmark Source Code
fn multiply_array_functional(n: &Vec<i64>) -> Vec<i64> {
    n.iter().map(|a| a * 10).collect()
}
fn multiply_array_imperative(n: &Vec<i64>) -> Vec<i64> {
    let mut new_list = Vec::with_capacity(n.len());
    for i in n {
        new_list.push(i * 10)
    }
    new_list
}
fn main() {
    let n = (0..1000).collect();
    // time each function on the same input 1000 times and log the total duration
    let time_start = std::time::Instant::now();
    for _ in 0..1000 {
        multiply_array_imperative(&n);
    }
    println!("{:?}", std::time::Instant::now().duration_since(time_start));
    let time_start_2 = std::time::Instant::now();
    for _ in 0..1000 {
        multiply_array_functional(&n);
    }
    println!("{:?}", std::time::Instant::now().duration_since(time_start_2));
}
```

```
// Oxynium Benchmark Source Code
def multiply_array_functional(n: List<Int>) List<Int> {
    return n.map!<Int>(fn (a: Int, _: Int) -> a * 10)
}
def multiply_array_imperative(n: List<Int>) List<Int> {
    // very similar to implementation for Array.map
    let len = n.len()
    let new_list = List.with_capacity!<Int>(len * 8)
    for idx in range(0, len) {
        new_list.push(n.at_raw(idx) * 10)
    }
    return new_list
}
def main() {
    let n = range(1000).List()
    // time each function on the same input 1000 times and log the total duration
    let time_start = Time.now()
    for _ in range(1000) {
        multiply_array_imperative(n);
    }
    print((Time.now() - time_start).Str());
    let time_start_2 = Time.now()
    for _2 in range(1000) {
        multiply_array_functional(n);
    }
    print((Time.now() - time_start_2).Str());
}
```

The Rust code example does not generate any lambda functions in the output, generating very similar machine code for both functions, meaning the Rust compiler was 'smart' enough to realise the intent of the program and generate highly optimised machine code that does not correlate strongly with the input source code.

The Rust code ran in a mean of 8.38ms for the functional version, and 11.1ms for the imperative version - a 24% slow-down, likely due to the additional information provided to the compiler by using the iterator interface. On the other hand, the Oxynium functional version ran in a mean of 32.3ms and 28.6ms for the imperative version - 11% faster when not using high level abstractions.

This perfectly demonstrates how mature compilers are able to transform highly abstract code into the most efficient machine code given the provided information, while Oxynium's younger compiler is unable to perform these optimisations, causing runtime penalties for using abstractions.

Reflections on Language Design

Oxynium's attempt to bridge the gap between high-level abstractions and low-level performance fall short due to the immaturity of the compiler and the complexity of the domain. Significant resources are required to build compilers, like Rust, capable of taking high-level or declarative code and producing performant binaries. As noted in the analysis of Coblenz et al., transitive immutability provides benefits for software engineering at a large scale. This is rendered pointless without being able to transform this idealistic software, written for teams of humans, into a language our computers are most comfortable with, as immutable code is generally less performant without these optimisations. Similarly, reflecting on Kennedy and Syme, Oxynium's choice of type erasure for generics may be most performant and simplest to implement, but moves the burden to the other two components of the trichotomy: the user and the runtime.

Perhaps, in order to create a programing language for humans, capable of translating well into machine code, a paradigm shift will be needed. Optimisations of the future will need to work at a higher level, producing performant code on increasingly abstract source code. Increasing the abstraction of the intermediate representation, either a linear IR or the AST itself, may be the way forward. Alternatively, Machine Learning may offer non-deterministic code translations to fill the need.

Oxynium 2

Oxynium 2 will be self-hosted, written in Oxynium 2, using Oxynium 1 as a bootstrap compiler [11]. Of the many improvements planned, perhaps the single most important will be moving away from x86-64 Assembly to a Control Flow Graph (CFG) based Intermediate Representation (IR) in Static Single Assignment (SSA) form, i.e. LLVM IR with the LLVM toolchain [5]. This will enable many zero-cost abstractions for 'free', and massively improve compatibility for target environments.

Other missing features include 'interfaces' for polymorphic sub-types using dynamic dispatch, modules, custom macros, local variable capture in lambda functions, and a compile-time memory management model similar to Rust. The developer tooling will also be a priority: linting, a Language Server Protocol implementation, static analysis code checker, package manager, plugins for IDEs, testing framework and online documentation.

Conclusion

This report has critically analyzed the design and implementation of Oxynium, specifically focusing on its approach to generics and strict immutability. By contrasting Oxynium's type erasure model with the hybrid reification strategy proposed by Kennedy and Syme, and evaluating its transitive immutability against the framework established by Coblenz et al., the trichotomy of compiler implementation effort, runtime performance, and developer expressiveness can be understood and used to guide the next generation of programming languages.

References

[1] Vishakha Agrawal. Challenges and complexities in enabling compilers to automatically optimize code. *International Journal of Innovative Research in Engineering & Multidisciplinary Physical Sciences (IJIRMPS)*, 8(1), 2020.

[2] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. Exploring language support for immutability. In *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 736–747. IEEE, 2016.

[3] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[4] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET common language runtime. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 1–12. ACM, 2001.

[5] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, 2004.

[6] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pages 256–267. ACM, 2017.

[7] Keith Randall and The Go Team. Generics implementation - gc shape stenciling. Design document, Google, 2021.

[8] Rust Embedded Working Group. Zero cost abstractions. `https://doc.rust-lang.org/beta/embedded-book/static-guarantees/zero-cost-abstractions.html`, 2023.

[9] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection (GCC)*. Free Software Foundation, 2024. Version 14.2.

[10] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4):19:1–19:40, 2013.

[11] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[12] J checking Zhu, K P N S K Amla, and Michalis Polychronakis. Security risks in asynchronous web servers: When performance optimizations amplify the impact of data-oriented attacks. In *Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15. IEEE, 2018.