

Existentialize Your Generics

Dimi Racordon

dimitri.racordon@epfl.ch
EPFL, LAMP
Lausanne, Switzerland

Matt Bovel

matthieu.bovel@epfl.ch
EPFL, LAMP
Lausanne, Switzerland

Hamza Remmal

hamza.remmal@epfl.ch
EPFL, LAMP
Lausanne, Switzerland

Abstract

The two main approaches to compile generic programs are dynamic dispatch and monomorphization. While the latter is typically preferred in the context of low latency applications, where the overhead of boxing may be prohibitive, it also comes at the cost of important limitations in terms of modularity, expressiveness, and code size.

The Swift programming language proposes an interesting third alternative that addresses these shortcomings, supporting dynamic dispatch without requiring boxing by factoring method tables out of object headers. This paper examines the merits of that strategy, which we call *existentialization*, across different programming languages. Our study shows that existentialization can produce code with competitive performance with respect to monomorphization.

CCS Concepts: • Software and its engineering → Object oriented languages; Polymorphism; Classes and objects; Source code generation.

Keywords: Existentialization, Generic Programming, Dynamic Dispatch, Type Classes, Separate Compilation

ACM Reference Format:

Dimi Racordon, Matt Bovel, and Hamza Remmal. 2025. Existentialize Your Generics. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '25)*, October 12–18, 2025, Singapore, Singapore. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3759426.3760975>

1 Introduction

Generic programming is the discipline of lifting general abstractions from concrete implementations [8]. Naturally, applying this discipline requires one mechanism to express these abstractions—e.g., parametric type polymorphism—and another to compile them into non-generic definitions,

usually implemented in one of two ways in mainstream programming languages. The first, *dynamic dispatch* [15], consists of deferring the selection of the function being applied to runtime. The second, *monomorphization* [7], consists of creating a specialized copy of each polymorphic function or data structure by substituting its generic parameters by concrete types.

Languages geared toward low latency applications typically opt for the latter. The appeal is that monomorphized code can be compiled as efficiently as if it had been written without generics. However, this technique also comes at the cost of three important limitations:

1. Firstly, monomorphization implies that the definition of a polymorphic function may potentially be recompiled at each of its call sites, which severely hinders separate compilation and works against binary compatibility [4].
2. Secondly, not all functions can be monomorphized. In particular, recursive calls nesting type arguments in structural types must be avoided, lest they would generate unbounded sets of monomorphized copies.
3. Lastly, monomorphization can lead to code blowup since the same definitions might be copied many times into compiled code.

Dynamic dispatch does not suffer from any of these limitations but brings another challenge to the table. If f is compiled only once, then how are its arguments supposed to be represented? The answer is simple in most object-oriented languages—e.g., Java—where all values but primitive ones are boxed along with an *object header* providing runtime information about the boxed instance and its operations. With this arrangement, arguments to generic functions can be passed as type-erased pointers to boxed values, relying on object headers to dispatch method calls. In systems programming, however, boxing is heavily discouraged as it may cause prohibitive runtime overhead, due to indirections. Instead, one should aim at representing data in contiguous memory regions to support cache-friendly memory-access patterns [6]. Further, when large collections of homogeneous data are considered, storing multiple references to the same header is space-inefficient and may also hinder optimizations.

One workaround is to box values “on demand” into existential containers [10]. In Rust, for example, one can use a so-called *dyn trait* to define a generic operation that does not require monomorphization. This trick is not without drawbacks, though. On the side of ergonomics, one issue

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MPLR '25, Singapore, Singapore

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-2149-6/25/10

<https://doi.org/10.1145/3759426.3760975>

is that existential containers weaken the guarantees that the type system can offer by erasing specificity, opening the door to well-known issues such as the binary method problem [1]. On the side of performance, the wrapping of a value into an existential container might require costly dynamic memory allocation. Even worse yet, the technique scales poorly in cases where parts of a generic data structure must be boxed rather than the whole. For instance, transforming a `Vec<Circle>` into a `Vec<Box<dyn Shape>>` in Rust may turn a constant time operation into a linear one.

The Swift programming language [9] proposes an interesting solution to alleviate these issues. When a polymorphic function cannot be monomorphized, either because of separate compilation requirements or because monomorphization would not terminate, it is compiled to a monomorphic version accepting its type parameters as additional term parameters. Intuitively, the idea is to factor out the information that would be otherwise stored in object headers, so that arguments to generic parameters can be passed as type-erased pointers to unboxed concrete values. This approach, which we dub *existentialization*, is an important part of Swift’s strategy to uphold resilient binary interfaces [14]. Nonetheless, the compiler systematically favors monomorphization if applicable, suggesting that existentialization should be used sparingly when performance is a concern [5].

This paper challenges that assumption. Specifically, it shows that existentialized code can run as efficiently as its monomorphized counterpart on state-of-the-art virtual machines and with competitive performance on bare metal. In fact, we observe that the largest contributor to the runtime overhead of existentialization is the cost of allocating objects whose sizes cannot be determined at compile-time, which can be optimized with careful use of the stack. In summary, we make the two following contributions:

- We give a language-agnostic description of existentialization, identify some key challenges, and comment on possible implementation strategies depending on the source language and the compilation target.
- We report performance measurements comparing existentialization to monomorphization using a series of micro benchmarks on homogeneous data structures featuring different levels of abstraction

2 Background

This section introduces some general background on object representations and binary compatibility.

2.1 Boxed vs Inline Representations

In managed languages, an object is typically represented as heap-allocated structures that contain the fields of the object along with a header containing runtime information about the object. The contents and layout of a header vary from one system to another but in general it comprises the type of

```

1 class Vector2(x: Float, y: Float)
2 class Circle(origin: Vector2, radius: Float):
3   def area() = radius * radius * 3.14f

```

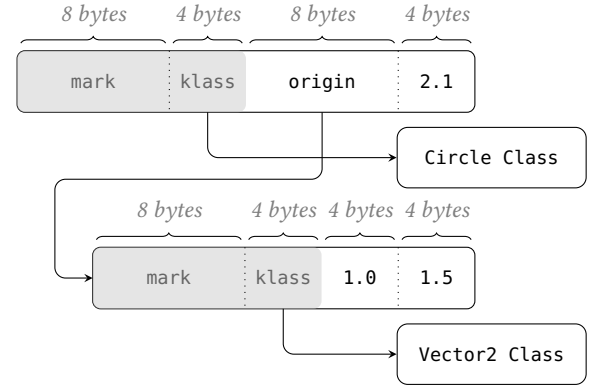


Figure 1. Representation of a Scala object in memory

```

1 struct Vector2 {
2   var x, y: Float
3 }
4 struct Circle {
5   var origin: Vector2
6   var radius: Float
7   func area() -> Float {
8     radius * radius * 3.14;
9   }
10 }

```

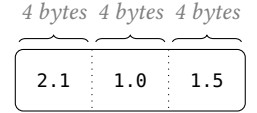


Figure 2. Representation of a Swift struct in memory

the object, the size of its payload, and a table mapping virtual methods to their implementation, aka a *vtable*. In Scala, for example, these elements are stored in the class metadata referred to by the header. Figure 1 illustrates, showing the in-memory representation of a simple object graph.

The box at the top depicts an instance of `Circle`. The first two segments of the box, shown in gray, form an object header. The `mark` word contains various information about the instance, notably an identity hash and metadata for garbage collection. The `klass` pointer refers to another object describing the class of the instance which, among other things, documents the offsets and alignments of the class’ fields and contains the class’ methods. The next segment of the box at the top contains a reference to the object encoding the origin of the circle,¹ which is represented similarly. Notice that the radius of the circle as well as the two coordinates of its origin are stored “inline”, as they are primitive values.

¹We assume a 64-bit machine without compressed pointers.

One issue with this traditional representation is that reading the origin of a circle requires one pointer indirection. In low-latency applications, the accumulated cost of such indirections can result in prohibitive runtime overheads. Further, the headers associated with each instance may also amount to excessive space overheads. In Figure 1, for instance, object headers account for half of the total memory. Consequently, to avoid these costs, systems programmers prefer laying out all data in contiguous regions and dispense with headers. In Swift for example, an instance of `Circle` would be represented as three contiguous floats only, as shown in Figure 2.

In the context of generic programming, this more efficient memory layout pairs well with monomorphization since the compiler has full knowledge of the information related to a circle and its parts during code generation. When doing separate compilation, however, the compiler lacks a closed-world view, so it must carry enough type information across module boundaries to perform dynamic dispatch at runtime, hence justifying a boxed approach.

2.2 Resilience Boundaries

Modern applications often involve a large amount of software designed as modular, reusable libraries. To speed up compilation times and, perhaps even more importantly, to support individual component updates, it is desirable for libraries to maintain application binary interfaces (ABIs) compatible with their previous versions.

The most obvious requirement is that it should be possible to modify the body of a function—e.g., to fix a bug—without requiring changes in clients’ code. A more complicated requirement is that it should be possible to modify the implementation of a data structure under the same constraints. To illustrate, imagine that we decide to publish the Scala code shown in Figure 1 as a library. Making the change below to the class `Circle` would not be a binary breaking change, meaning that any software compiled with the original code could be linked against this new version.

```
1 class Circle(origin: Vector2, radius: Float):
2   var scale: Double = 0.0
```

Doing a similar addition would be a breaking change in Swift unless the original code was compiled with support for library evolution enabled [12] as it would modify the size of the struct. Similarly, modifying the order in which the properties a `Circle` are declared could also constitute a breaking change. This problem does not occur in Scala because the contents of the class is boxed and therefore its layout is not part of the ABI.

To support changes to the contents of a data structure provided by a library in a language using inline representations, one must ensure that clients of that library do not emit code that depends on the specifics of a particular memory layout. Instead, clients should obtain information about sizes,

alignments, and field offsets at runtime and use pointers to exchange data across libraries.

Remark that there does not exist a universal definition of what precisely constitutes a software library across different programming languages or even software projects. Hence, in the context of Swift, ABI compatibility is instead described relative to *resilience boundaries*, which are defined explicitly in package manifests.

3 Existentialization

This section introduces existentialization by means of a running example. We start with general principles and close with a few observations on additional features that can be built on top of it.

3.1 Encoding Type Metadata

The central idea of existentialization is to provide the information necessary for allocating generic data structures and calling their methods without access to their implementations ahead of time and without altering the representation of concrete instances. The first part of this problem is orthogonal to parametric polymorphism and thus we can set type parameters aside for the moment.

One way to conceptually understand the task is to start from a classical boxed representation and factor relevant information out of object headers. Looking at Figure 1, we observe that the headers contain two kind of data. The first, stored in the `mark` word, relates to the associated instance specifically whereas the second, referred to by the `klass` pointer, relates to all instances of the type. Given that we are concerned about abstracting over types, we should focus our attention on the latter.

Memory Layout. In order to determine how to allocate memory for a data structure, we first need to determine its size and alignment. In our running example, this information is simply contained in the class metadata.² If the source programming supports direct fields access, then type metadata should also describe the memory layout of each stored member, which can be done by mirroring the structure of class’ contents. Naturally, other sorts of types (e.g., sum types, arrays, etc.) may require other kinds of layout descriptions. An alternative approach to support field access is to generate getters and setters, theoretically trading speed for memory.

Runtime-Sized Types. Some programming languages support types whose sizes is only determined at runtime, such as variable-length arrays in C. Perhaps surprisingly, supporting them does not require any particular consideration during existentialization. In fact, allocating new memory for *any* generic type requires runtime-sized allocation in the

²All mark words are aligned at 8 and padding is inserted in the payload if the first field has a larger alignment requirement.

compilation target. Stack-allocation remains possible on systems supporting low-level instructions for manipulating the stack pointer. On other systems, a compromising strategy is to use fixed size buffers large enough to fit most values and default to heap allocation otherwise.

Primitive Operations. We should finally encode primitive operations that are universally supported for all data types, such as assignment, copy, and deinitialization, and have compiler-generated definitions handling internal implementation details, such as pointer tagging or small object optimization [3]. In monomorphic contexts, the compiler can simply generate the appropriate code depending on concrete type information. In generic contexts, however, these primitive operations should be dispatched dynamically.

Example. The following snippet shows a possible implementation of the type metadata for a vector type similar to that of Figure 2, using C++ as a compilation target.

```
1  template<size_t n>
2  void trivialcopy(void* d, void* s) {
3      memcpy(d, s, n);
4  }
5  void trivialdrop(void*) {};
6
7  struct Type {
8      size_t size;
9      vector<Type const*> members;
10     void(*copy)(void*, void*);
11     void(*drop)(void*);
12 };
13
14 const Type Float{
15     4, {},
16     &trivialcopy<4>, &trivialdrop
17 };
18 const Type Vector2{
19     8, {&Float, &Float},
20     &trivialcopy<8>, &trivialdrop
21 };
```

Type represents the metadata of a type. It describes the size of an instance, its members, and two operations, one for copying instances and another for deinitializing them. Abstracting over these operations will let us implement more sophisticated behavior, as we will discuss in Section 3.5. Line 14 shows the construction of the metadata for Swift’s Float. Likewise, line 18 shows the metadata of Vector2. In both cases, copying is implemented as a simple call to memcpy and deinitialization is a no-op.

3.2 Using Type Metadata

Existentializing a generic function consists of replacing type parameters by term parameters accepting type metadata and replacing operations that depend on a generic type parameters by operations that depend on these additional term

parameters. To illustrate, let us start with the following definition of the polymorphic identity in Swift:

```
1 func id<T>(_ x: T) -> T { x }
```

The first step in existentializing this definition is to replace the generic type parameter T by a corresponding term parameter. Then, the only generic operation to transform is the copy of x that is made to form the return value. Copying is a non-customizable primitive operation in Swift and therefore its implementation for T can be accessed through the type metadata passed as argument. More concretely, one possible way to existentialize id is to implement it as follows:

```
1 void id(void* r, void* x, Type const* T) {
2     T->copy(r, x);
3 }
```

Notice that the function’s result is written to a pointer taken as parameter rather than returned like in the original definition. The reason is that the size of the type T is unknown at compile-time and thus there is no way for the compiler to determine how much memory id should allocate. Instead, this responsibility is transferred to the caller which has either access to type metadata that can be passed along, or knowledge of the concrete type for which id is instantiated and can thus provide the appropriate type metadata. For example, an expression of the form id(4.2) could be desugared as follows:

```
1 float x = 4.2;
2 float y;
3 id(&y, &x, &Float);
```

Remark that this encoding satisfies the requirements of a resilient ABI, mentioned in Section 2.2. Since id is not monomorphized, its body can be modified without requiring any change at call sites. Meanwhile, it makes no assumption about the memory layout of its argument or the way instances should be copied. Furthermore, the existentialization of id can be used with any arbitrary type without involving any code duplication.

3.3 Generating Type Metadata

The previous examples have used type metadata constructed statically, as global variables. It may however be necessary to generate new instances at runtime, since the set of all types possibly instantiated by a program cannot be enumerated ahead of time in general. For example, consider a simple function swapping the elements of a pair:

```
1 template<typename T, typename U>
2 pair<U, T> swap(pair<T, U> const& p) {
3     return make_pair(p.second, p.first);
4 }
```

The existentialized form of swap requires the metadata of the type pair<U, T> to handle the result of make_pair, which must be constructed from the metadata of T and U, received as parameters. There is no way for the compiler

to determine all possible values for these parameters ahead of time and thus the metadata of `pair<U, T>` must be determined at runtime.

It may be desirable to ensure a one-to-one mapping between a type and the instance representing its metadata to amortize the cost of creating new instances, curb memory overhead, and support fast type equality. All of these goals can be achieved by *interning* the metadata by means of a hash table maintained by the runtime system.

Incidentally, this approach also circumvents a crippling issue of monomorphization, namely the inability to process recursive functions nesting type arguments in structural types. For example, the program below cannot be compiled in C++ although its equivalent in Java would have no issue. The call in `main` triggers the monomorphization of `f`, demanding a copy substituting `T` for `int`. Then, the recursive call in line 5 demands another copy substituting `T` for `pair<int, int>`, which eventually demands another copy substituting `T` for `pair<pair<int, int>, int>`, and so on. Thus, monomorphization never terminates even though the call `f(1, 0)` would not diverge at runtime.

```
1 template<typename T>
2 int f(T const& a, int n) {
3     return (n > 3)
4         ? 0
5         : f(make_pair(a, 1)), n + 1);
6 }
7 int main() { return f(1, 0); }
```

Existentialization sidesteps this issue completely because only one version of the function will be compiled while the type metadata passed down to each recursive call to `f` will be computed lazily, at runtime.

3.4 Dynamic Dispatch

So far we have avoided the problem of dispatching arbitrary user-defined methods at runtime. Unlike the class metadata of Java, the type metadata that we have presented do not contain a method table and thus additional information is required to support dynamic dispatch.

Two approaches can be taken, depending on the source language. If the methods of a type form a closed set that can be enumerated by looking at the type's definition, as in Java, then one can simply store virtual method tables in types metadata, provided that each entry is either monomorphic or existentialized. Otherwise, one can rely on dictionary-passing, taking inspiration from systems supporting type classes [13]. We illustrate the latter technique below with the following Swift program:

```
1 protocol Shape {
2     func area() -> Float
3 }
4 func smallest<T: Shape>(x: T, y: T) -> T {
5     x.area() > y.area() ? y : x
6 }
```

Line 1 introduces a type class—a *trait* in Rust—describing an interface for values representing shapes. It is used by the generic function in line 4 to constrain its generic parameter, thus ensuring that calling `x.area` is well-typed in the body. At call site, the compiler must provide an instance of this class for the type with which the function is instantiated. This instance will then be used at runtime to dispatch calls to the implementation of the `area` method. Interestingly, this strategy is in fact dynamic dispatch, only through type class instances rather than method tables. Static dispatch can only occur if the implementation of a type class' interface can be resolved at compile time, after type substitution. Hence, existentialization can just re-use this machinery.

A type class is usually compiled to a record whose members are implementations of the functions defined by the class. For example, the `Shape` class introduced above can be compiled as follows using C++:

```
1 struct Shape {
2     float(*area)(void*);
3 };
```

Because the generic type parameter of `smallest` is declared conforming to `Shape`, the existentialized form of the function has to take an instance of the above struct as an additional parameter, which in turn will be used to dispatch calls to `area` dynamically. Putting everything together, we obtain the following definition:

```
1 void smallest(
2     void* r, void* x, void* y,
3     Type const* T, Shape const* shape
4 ) {
5     if (shape->area(x) > shape->area(y)) {
6         T->copy(r, y);
7     } else {
8         T->copy(r, x);
9     }
10 }
```

As we have discussed, functions like `id` or `smallest` write their result to storage expected to have been allocated earlier in the call hierarchy. If such an allocation is for a concrete type, then we can allocate a fixed amount of bytes on the stack or the heap before passing a pointer to an existentialized function, as in the example from Section 3.2. However, if the allocation is for a generic type, then its size can only be determined at runtime, using some type metadata. For example, consider the following Swift function:

```
1 func totalArea<T: Shape>(xs: [T]) -> Float {
2     var s: Float = 0.0
3     for i in 0 ..< xs.count {
4         s += xs[i].area()
5     }
6     return s
7 }
```

The expression `xs[i]` in line 4 is applying the subscript accessor of Swift’s Array type. The return value of this operation is an instance of `T`, which is generic in this context. Hence, we cannot determine statically how much memory the existentialized form of `totalArea` must allocate to store the result of `xs[i]`. Instead, we must allocate the size described by the type metadata corresponding to `T`.

Another interesting aspect of this function is that the iteration over the indices of `xs` also involves a generic operation, namely the call to `count`, since the type of `xs` is itself generic—`[T]` is sugar for `Array<T>` in Swift. This method `count` is defined by the `Collection` protocol, to which `Array` conforms, and thus we must obtain an instance of this type class, as shown below: Since Swift adopts global uniqueness of type class instances [11], we can construct one directly in the existentialized form of `totalArea`, without adding new parameters, as shown below:

```

1 float totalArea(
2     void* xs, Type const* T, Shape const* shape
3 ) {
4     auto* c = array_is_collection(T);
5     auto* e = alloca(T->size);3
6     float s = 0.0f;
7     for (size_t i = 0; i < c->count(xs); ++i) {
8         c->at(e, xs, i);
9         s += shape->area(e);
10        shape->drop(e);
11    }
12    return s;
13 }
```

Line 4 retrieves the witness of `Array<T>`’s conformance to `Collection`, assuming the existence of global a function implementing the construction of such type class instance. This assumption is justified by the fact that Swift enforces global uniqueness of type class instances [11], meaning that types can conform to a protocol in only one way, globally. In a language supporting local type class instances—e.g., Scala—this witness should be taken as an additional parameter instead. Line 5 allocates enough storage to store an instance of `T` on the stack. Then, the loop in line 7 uses the collection witness to iterate over the indices of the array and access its contents, which are copied one element at a time into the storage allocated dynamically. Finally line 10 cleans up each copied instance. Note that the stack allocation is not freed until the function returns, though, so the storage can be reused by the next iteration.

3.5 Sharing

While our examples have focused on supporting independent value representations, existentialization is not incompatible with sharing and managed pointers. To illustrate, let us consider the problem of representing a managed buffer

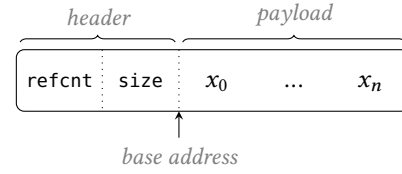


Figure 3. Representation of shared managed buffer

of arbitrary length. One way to implement such a data structure is to tail-allocate the buffer behind a header consisting of a reference counter and a size, as shown in Figure 3. As the most common operation one performs on a buffer is to access its elements, it is convenient to keep the address pointing at the start of the tail-allocated contents rather than the start of the header. The latter can be computed with a simple subtraction.

The following definitions describe how to construct the type metadata of the data structure. Since an instance can be represented by storing the base address of the contents, the size of the data type is defined as the size of a single pointer, as shown in line 15. Rather than copying the bytes of an instance, the copy operation simply increases the reference counter in line 5 and copies the box pointer. Notice that `h`, which denote the address of the reference counter, is computed by offsetting the address of the buffer’s payload. Finally, the drop operation decreases the reference counter in line 11 and deallocates the whole buffer when it is no longer referenced.

```

1 struct ManagedBuffer {
2     static void copy(void* d, void* s) {
3         auto* b = static_cast<size_t**>(s);
4         auto* h = *b - 2;
5         *h += 1;
6         memcpy(d, s, sizeof(void*));
7     }
8     static void drop(void* s) {
9         auto* b = static_cast<size_t**>(s);
10        auto* h = *b - 2;
11        *h -= 1;
12        if (*h == 0) { free(h); }
13        *b = nullptr;
14    }
15    static constexpr Type metadata{
16        sizeof(void*), {}, &copy, &drop
17    };
18 };
```

More efficient garbage collection mechanisms are obviously possible. The take away is that the ability to customize primitive operations can be leveraged to support compiler-defined behaviors in generic contexts without requiring a uniform representation of all data types.

³We omit the handling of alignment requirements for brevity.

3.6 Type Reflection for Free

As we have briefly mentioned in Section 3.1, type metadata can be leveraged to support reflection by allowing the user to consult their contents. This observation is actually not at all surprising considering the fact that, looking at existentialization through the lens of type classes, type metadata can be understood as witnesses that types are types.

One of the most common type reflection features is the *type test*, i.e., the ability to inspect the runtime type of a value at runtime. Type tests are often used in conjunction with pattern matching, as in the Scala below. The function `isTwo` inspects the type of its argument to evaluate different expressions, depending on that type.

```
1 def isTwo[T](v: T) = v match
2   case i: Int => i == 2
3   case s: String => s == "two"
4   // case o: Option[Int] => o == Some(2)
5   case _ => false
```

Perhaps surprisingly, uncommenting line 4 will result in a warning complaining that type arguments cannot be determined from the signature of `isTwo`. This limitation is consequence of type erasure on the JVM.

Although Scala supports parametric polymorphism, type parameters are ignored on the JVM which relies only on subtyping and dynamic dispatch. This strategy simplifies the bytecode but also comes at the cost of information loss at runtime: it is not possible to recover the arguments of a type application in general. Consequently, one cannot indeed perform a type test against `Option[Int]` as the test would succeed not only for actual instances of `Option[Int]`, but also for instances of `Option[T]` for any other `T`.

This issue could be solved by existentializing `isTwo`, as that would translate its generic type parameter into a term parameter rather than erase it. Since this parameter is meant to identify the type with which `isTwo` has been instantiated, we could then simply check whether its value is equal to the metadata of another concrete type to perform a type test, as shown below. The call to metadata returns the runtime description of `Option[Int]`, which must be distinct from that of another application of `Option`.

```
1 bool is_two(void* v, Type const* T) {
2   if (T == metadata(&Option, &Int)) { ... }
3   ...
4 }
```

4 Evaluation

This section reports how existentialization fares against monomorphization by comparing the runtime performance of a synthetic microbenchmark compiled with either techniques in five different programming languages, namely Swift, C++, Rust, Scala, and Javascript. The purpose of our

benchmarks is to isolate and measure the overhead on method-call dispatch introduced by the different compilation strategies. We chose Swift because it implements existentialization natively, and C++ and Rust because both provide low-level primitives that allow us to emulate existentialization manually. To broaden the comparison, we also included two JIT-compiled languages (Scala and JavaScript) to assess whether JIT optimizations can mitigate the indirection cost of existentialization. We start with a description of our benchmarks, discuss how we implemented them in each language, and finally present our results.

4.1 Benchmarks

Our benchmarks consist of summing the areas of 10 million shapes stored in one collection. They involve two generic parameters: one abstracts over the type of the shapes, the other abstracts over the type of the collection. We reuse some of the types and type classes discussed throughout in Section 3, whose definitions are repeated below:⁴

```
1 protocol Shape {
2   func area() -> Double
3 }
4
5 struct Vector2 {
6   var x, y: Double
7 }
8
9 struct Circle: Shape {
10  let center: Vector2
11  let radius: Double
12  func area() -> Double {
13    radius * radius * 3.14
14  }
15 }
```

Each individual benchmark is a different variant of the same generic summing algorithm, testing increasing levels of existentialization:

Monomorphized The element type (`Circle`) and the collection type (a dynamically-sized array) are known at compile-time and the function is fully specialized.

Partially existentialized The element type is abstract and but the collection type is still known to be a dynamically-sized array at compile time.

Fully existentialized The element type and the collection type are abstract.

We use Swift’s built-in support for existentialization and emulate it in other languages by applying the transformation manually. We come to this process in Section 4.3.

⁴We use `Double` instead of `Float` for simplicity, as it is the default floating-point type in most of the languages we examine.

4.2 Setup in Swift

The monomorphized variant is implemented as a generic function accepting a type parameter `T` constrained to `Shape`, and a term parameter of type `ContiguousArray<T>`.⁵

```
1 public func area<T: Shape>(  
2     _ shapes: ContiguousArray<T>  
3 ) -> Double {  
4     var totalArea: Double = 0.0  
5     var it = shapes.makeIterator()  
6     while let e = it.next() {  
7         totalArea += e.area()  
8     }  
9     return totalArea  
10 }
```

As expected, this version gets fully monomorphized by the compiler. As the following LLVM IR excerpt confirms, the optimizer is capable of simply inlining the call to `area`, resulting in straightforward code.

```
1 define double @areaCircle6(ptr %0) {  
2     ...  
3     %11 = fmul double %9, 3.140000e+00  
4     %12 = fmul double %9, %11  
5     %13 = fadd double %7, %12  
6     ...  
7 }
```

We can wrap calls to a generic function in a forwarder annotated with `@_optimize(none)` to force existentialization. The annotation prevents any optimization on the body of the forwarder, including monomorphization, without affecting the wrapped function.

```
1 @_optimize(none)  
2 public func areaPartialExist<T: Shape>(  
3     _ shapes: ContiguousArray<T>  
4 ) -> Double { area(shapes) }
```

This trick essentially emulates a resilience boundary in a source single file. We can confirm that `area` gets existentialized in the way we have discussed in Section 3 by inspecting LLVM IR and observe that the compiled function accepts three witness parameters. Further, and as expected, the call to `Shape`'s `area` is dispatched through a pointer in the witness that `T` conforms to `Shape`.

```
1 define double @areaShape(  
2     ptr %0, ptr %T, ptr %T.Shape  
3 ) {  
4     ...  
5     %10 = getelementptr inbounds i8,  
6         ptr %T.Shape, i64 8  
7     ...  
8     %17 = load ptr, ptr %10, align 8
```

⁵We use `ContiguousArray` to avoid the runtime overhead incurred by the handling of the different possible internal representations of `Array` on platforms supporting interoperability with Objective-C.

⁶The name of compiled function is actually mangled in LLVM IR. We have renamed it for clarity.

```
9     %18 = call swiftcc double %17(  
10         ptr noalias nonnull swiftself %4,  
11         ptr %T, ptr %T.Shape)  
12     ...  
13 }
```

The same forwarding trick can be used to implement the fully existential variant, adding one type parameter to the wrapped function.

```
1 func areaColl<T: Shape, C: Collection<T>>(  
2     _ shapes: C  
3 ) -> Double {  
4     var totalArea: Double = 0.0  
5     var it = shapes.makeIterator()  
6     while let e = it.next() {  
7         totalArea += e.area()  
8     }  
9     return totalArea  
10 }  
11  
12 @_optimize(none)  
13 func areaCollEx<T: Shape, C: Collection<T>>(  
14     _ shapes: C  
15 ) -> Double { areaColl(shapes) }
```

This version is compiled down to LLVM IR similar to the partially existentialized case, only with two additional parameters denoting the type metadata of `C` and a witness that `C` conforms to `Collection`.

```
1 define double @"areaShape"(  
2     ptr noalias %0, ptr %T, ptr %C,  
3     ptr %T.Shape, ptr %C.Collection  
4 ) { ... }
```

4.3 Setup in C++ and other languages

As other languages do not support existentialization natively, we emulate it by passing type metadata and type class instances explicitly. We present our C++ implementation thereafter, which is similar to the code generated by the Swift compiler and satisfies the same requirements with respect to resilience. Specifically, existentialized code makes no assumption about the layout or the operations of generic parameters. Our implementations in Rust, Scala, and Javascript are built similarly and satisfy the same properties.

We start with the data structures that are generated automatically by Swift's compiler. The first represent type metadata and was already discussed in Section 3.1. We ignore the representation of fields for simplicity, as this information is irrelevant to our benchmark: in the benchmark we never access fields of generic types directly in a generic context; fields are only accessed indirectly, through the `area` method.

```
1 struct Type {  
2     std::size_t size, alignment;  
3     void(*copy)(void*, void*);  
4     void(*drop)(void*);  
5 };
```

Next, we define three data structures denoting type class instances. Associated types [2] are represented as type metadata whereas methods are represented as function pointers. For example, consider the following protocol in Swift:

```
1 protocol Iterator {
2     associatedtype Element
3     func hasNext() -> Bool
4     mutating func next() -> Element
5 }
```

Its translation in C++ is a record containing a pointer to type metadata for encoding the associated type and two function pointers for the methods `hasNext` and `next`.

```
1 struct IteratorWitness {
2     Type const* element_type;
3     bool(*has_next)(void*);
4     void(*next)(void*);
5 };
```

We use `std::vector` to replace Swift’s `ContiguousArray` and define the following helper to act as an iterator:

```
1 template<typename E>
2 struct VectorIterator {
3     std::vector<E> const* base;
4     std::size_t i;
5     inline bool has_next() const {
6         return i < (*base)->size();
7     }
8     inline E const* next() {
9         return (*base)->at(i++);
10    }
11 };
```

This implementation deviates a little from Swift’s as it produces pointers to the contents of the vector rather than copying its elements. These semantics yield better performance in existentialized contexts as we will discuss later. Nonetheless, a copying version can be defined in a similar fashion and is included in our artifact.⁷

The monomorphized variant of our benchmark operates directly on an instance of `std::vector`. Rather than using a range-based loop, however, we perform iteration using the iterator presented earlier. This choice is intended to make the comparison with Swift and other C++ variants fairer.

```
1 template<typename T>
2 double area(Vector<T> const& shapes) {
3     double total_area = 0.0;
4     auto i = VectorIterator<T>(&shapes, 0);
5     while (i.has_next()) {
6         auto const* s = i.next();
7         total_area += (*s)->area();
8     }
9     return total_area;
10 }
```

⁷<https://github.com/mbovel/generics-benchmarks>

The existentialized version accepts the type metadata of `T` and a witness that it conforms to `Shape`, matching the signature of its compiled counterpart in Swift exactly.

```
1 template<typename T>
2 double area_partially_existentialized(
3     Vector<T> const& shapes,
4     Type const* t_type,
5     ShapeWitness const* t_shape
6 ) {
7     double total_area = 0.0;
8     auto* w = VectorIteratorWitness(t_type);
9     auto* i = alloca(w->element_type);
10    w->make_iterator(i, &shapes);
11    while (w->has_next(i)) {
12        auto* s = w->next(i);
13        total_area += t_shape->area(s);
14    }
15    return total_area;
16 }
```

Finally, the fully existentialized version further abstract over the type of the collection, thus accepting additional witnesses. A small difference compared to the Swift version is that the type metadata for the collection and its elements are not passed as individual parameters. Instead, they are obtained through the type class instances of `Iterable` taken as parameters.

```
1 template<typename T>
2 double area_fully_existentialized(
3     void* shapes,
4     IterableWitness const* t_iterable,
5     ShapeWitness const* t_element_shape
6 ) {
7     double total_area = 0.0;
8     auto* w = t_iterable->iterator_iterator;
9     auto* i = alloca(
10        t_iterable->iterator_type->size();
11    t_iterable->make_iterator(i, shapes);
12    while (w->has_next(i)) {
13        auto* s = w->next(i);
14        total_area += t_element_shape->area(s);
15    }
16    t_iterable->iterator_type->drop(i);
17    return total_area;
18 }
```

4.4 Performance Measurements

Figure 4 presents the performance results for the three benchmark variants across five programming languages—Swift, C++, Rust, Scala, and JavaScript—and two hardware architectures (x86 and ARM).

As expected, monomorphized versions are faster than their existentialized counterparts in most cases. In both C++ and Rust, the overhead introduced by dynamic dispatch is around 2–2.5× using monomorphized execution as a baseline.

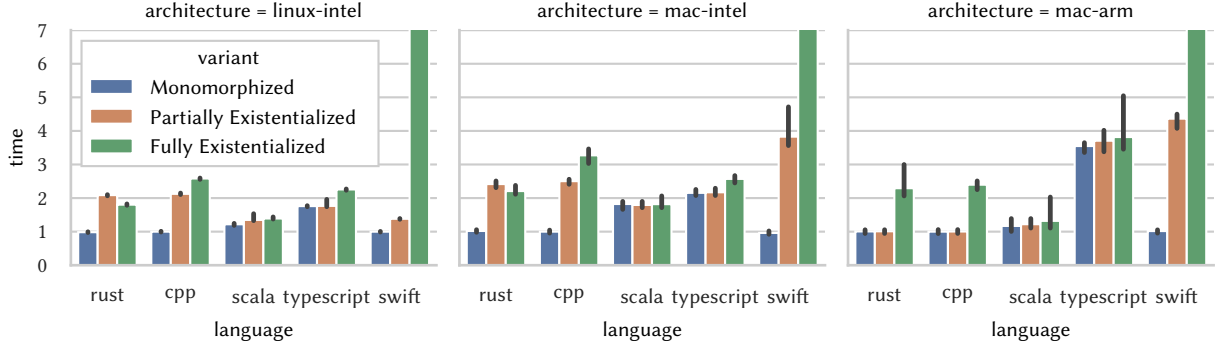


Figure 4. Run time of computing the sum of areas of 10 million circles in a contiguous array. Lower is better. Colors correspond to the three benchmark variants (see Section 4.1). The x-axis shows the programming language; the y-axis shows runtime normalized relative to the C++ monomorphized version. The y-axis is truncated at $\times 7$ —the fully existentialized version in Swift exceeds $\times 30$ for all architectures. The benchmark was executed with 50 warm-up iterations followed by 50 measured iterations. Error bars indicate the minimum and maximum values, and the top of each bar represents the mean. Linux Intel results obtained on a machine with an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz processor, Mac Intel on a MacBook Pro with a 2.6 GHz 6-core Intel Core i7 and Mac ARM results on a MacBook Pro with an M1 Pro. The compiler version/runtime versions used are rustc 1.89.0-nightly (45f256d9d 2025-05-27), clang 20.1.2, TypeScript using ts-node 10.9.2 on Node.js 23.6.0, Scala 3.7.0 on openjdk 21.0 and Swift 6.1.2

Swift performs significantly worse than the other languages for the fully existentialized version. Part of the reason is that Swift’s iterator returns owned copies of the iterated sequence rather than references, which costs two additional dynamic calls, as explained in Section 3.4. However, profiling with Xcode shows that the main contributor of this overhead is the heap-allocation used to store the iterator and its results, causing an order of magnitude slowdown (over 650ms): more than 50% of the time is spent in `nanov2_malloc_type` and `nanov2_free`. To confirm this observation, we have reimplemented the fully existentialized implement in Swift benchmarks by applying existentialization manually and using a custom iterator producing references rather than copies. This variant achieved much better performance, with a runtime of ~ 100 ms. We believe that these results highlight a performance pitfall in Swift’s design which could be addressed with by bringing support for borrowed references—a proposal going in this direction is currently underway—or improve the compiler’s ability to optimize common iteration patterns in existentialized code. Note also Swift also performs poorly for the partially existentialized variant on macOS platforms. The overhead is due to a compiler-inserted runtime check protecting against stack overflow.

Focusing on ARM platforms, an interesting observation is that the partially existentialized version is as fast as the monomorphized one both for Rust and C++. We hypothesize that this result is due to the use of the BL (branch with link) instruction, which avoids saving the return address on the stack, thereby reducing memory reads.

On x86 platforms, note that Rust’s fully existentialized variant is slightly *faster* than its partially existentialized counterpart. We inspected both the LLVM IR and the final assembly after disabling link-time optimizations (LTO) to investigate but confirmed that the partially existentialized variant contains a single dynamic call in the hot loop while the fully existentialized variant contains several, as expected. Hence, the performance advantage of the fully existentialized variant remains unfortunately unexplained; we have reached out to Rust experts for possible answers.

5 Conclusion

We studied existentialization, a technique first implemented by the Swift programming language for the separate compilation of generic functions and data structures. Unlike traditional approaches to dynamic dispatch, existentialization does not rely on boxing, thus supporting efficient inline data representations. Instead, the information necessary for allocating generic data structures and calling their methods is passed down separately, in a style reminiscent of dictionary-passing on type-class-oriented systems.

We have explained existentialization independently from its original implementation in Swift and applied it manually in different programming languages to evaluate its performance. Although monomorphization remains the technique producing the fastest code on bare metal, our measurements reveal that existentialization yields competitive performance. Surprisingly, however, we measured significant slowdowns using Swift, mostly due to heap allocation and unnecessary copying. These results suggest possible improvements to

Swift’s compiler implementation, as our manual translations in Swift and other languages could avoid these heap allocations without compromising resilience boundaries. Finally, our measurements showed that existentialization benefits fully from the optimizing JIT compilers, making it a viable alternative to subtyping for compiling generics.

Acknowledgments

This project has been funded by the Swiss National Science Foundation project entitled “Capabilities for Typing Resources and Effects” (TMAG-2_209506/1).

References

- [1] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. 1995. On Binary Methods. *Theory Pract. Object Syst.* 1, 3 (1995), 221–242.
- [2] Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones, and Simon Marlow. 2005. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 1–13. doi:10.1145/1040305.1040306
- [3] Raymond Chen. 2024. Inside STL: The string. <https://devblogs.microsoft.com/oldnewthing/20230803-00/?p=108532> Accessed on June 22nd, 2025.
- [4] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. 1998. What is Java Binary Compatibility?. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1998, Vancouver, British Columbia, Canada, October 18-22, 1998*, Bjørn N. Freeman-Benson and Craig Chambers (Eds.). ACM, 341–361. doi:10.1145/286936.286974
- [5] Doug Gregor. 2024. Swift for C++ Practitioners, Part 4: Generics. <https://www.douggregor.net/posts/swift-for-cxx-practitioners-generics/> Accessed on June 22nd, 2025.
- [6] Rishabh R. Iyer, Katerina J. Argyraki, and George Candea. 2024. Automatically Reasoning About How Systems Code Uses the CPU Cache. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 581–598. <https://www.usenix.org/conference/osdi24/presentation/iyer>
- [7] Matthew Lütze, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2025. The Simple Essence of Monomorphization. *Proc. ACM Program. Lang.* 9, OOPSLA1 (2025), 1015–1041. doi:10.1145/3720472
- [8] David R. Musser and Alexander A. Stepanov. 1988. Generic Programming. In *Symbolic and Algebraic Computation, International Symposium (ISSAC) (Lecture Notes in Computer Science, Vol. 358)*, Patrizia M. Gianni (Ed.). Springer, 13–25. doi:10.1007/3-540-51084-2_2
- [9] Slava Pestov. 2024. Compiling Swift Generics: Implementation Reference Manual. (June 2024). Unpublished.
- [10] Dimi Racordon, Eugene Flesselle, and Matthieu Bovel. 2024. Existential Containers in Scala. In *ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR, M. Anton Ertl and Christoph M. Kirsch (Eds.)*. ACM, 55–64. doi:10.1145/3679007.3685056
- [11] Dimi Racordon, Eugene Flesselle, and Cao Nguyen Pham. 2025. On the State of Coherence in the Land of Type Classes. *Art Sci. Eng. Program.* 10, 1 (2025). doi:10.22152/PROGRAMMING-JOURNAL.ORG/2025/10/15
- [12] Jordan Rose and Ben Cohen. 2024. Library Evolution for Stable ABIs. <https://github.com/swiftlang/swift-evolution/blob/main/proposals/0260-library-evolution.md> Accessed on June 22nd, 2025.
- [13] Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 60–76. doi:10.1145/75277.75283
- [14] Andrew Wagner, Zachary Eisbach, and Amal Ahmed. 2024. Realistic Realizability: Specifying ABIs You Can Count On. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1249–1278. doi:10.1145/3689755
- [15] David Wonnacott. 2001. Using Accessory Functions to Generalize Dynamic Dispatch in Single-Dispatch Object-Oriented Languages. In *USENIX Conference on Object-Oriented Technologies and Systems*, Rajendra Raj and Yi-Min Wang (Eds.). USENIX, 93–102. <http://www.usenix.org/publications/library/proceedings/coots01/wonnacott.html>

Received 2025-06-24; accepted 2025-07-28