Compiling Structural Types on the JVM

A Comparison of Reflective and Generative Techniques from Scala's Perspective

Gilles Dubochet
gilles.dubochet@epfl.ch

Martin Odersky
martin.odersky@epfl.ch

École Polytechnique Fédérale de Lausanne

ABSTRACT

This article describes Scala's compilation technique of structural types for the JVM. The technique uses Java reflection and polymorphic inline caches. Performance measurements of this technique are presented and analysed. Further measurements compare Scala's reflective technique with the "generative" technique used by Whiteoak to compile structural types. The article ends with a comparison of reflective and generative techniques for compiling structural types. It concludes that generative techniques may, in specific cases, exhibit higher performances than reflective approaches, but that reflective techniques are easier to implement and have fewer restrictions.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.3.2 [**Programming Languages**]: Object-Oriented Languages; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Algorithms, Languages, Measurement, Performance

1. INTRODUCTION

νObj, which was presented in 2003, is a calculus for objects and classes with type members [5]. The Scala programming language [4, 6], first described in 2004, is Odersky's concretisation of these same ideas. For the sake of this article, we recall that *νObj* includes both nominal and structural subtyping in a unified way. Malayeri and Aldrich later proved that the unification of nominal and structural types is sound using their *Unity* calculus [3]. A type is a *nominal* subtype of another one if there exists, somewhere in the program, an explicit mention of this fact. In Java, such an explicit mention takes the form of an **`extends`** clause. *Structural* subtyping, also known as duck typing, declares a type to be subtype of another if their structures — their members — allow it. At the simplest, a type is allowed to be a subtype of another if it exposes at least the same members. This relation is available in Scala through structural types. They replace refinement types, a restricted form of structural types (see section 2) that are easier to compile. To illustrate Scala's structural types, let us consider an example.

Some objects have a `close` method that must be called after use. Without structural types, writing a generally applicable `autoclose` method would require all closeable objects to implement an interface containing the `close` method. Structural types solve the problem differently.

```
type Close = Any { def close: Unit }
def autoclose(t: Close)(run: Close => Unit): Unit = {
  try { run(t) }
  finally { t.close }
}
```

To be compatible with the "`Close`" structural type, an object must be compatible with `Any` — all objects in Scala are — and it must implement the `close` method. Such types are called "structural" because it is the structure of their members — name, type of the arguments and of the result — that define whether an object is of a given type. The `autoclose` method is implemented using the `Close` type. Its arguments are: first, the object to be automatically closed, such as a socket or file, and second, some code that uses the object and must be run before the object can be closed.

```
autoclose(new FileInputStream("test.txt"))( file =>
  var byte: Int = file.read
  while(byte > -1) {
    print(byte.toChar)
    byte = file.read
  }
)
```

The closeable object that is passed to `autoclose` is an instance of `FileInputStream`. It conforms to the structural type because it contains a `close` method. `FileInputStream`, which is part of Java's standard library, does not implement any particular interface related to type `Close`. The second parameter for `autoclose` is a block of code that will read the stream. After executing it, `autoclose` will `close` the file. The method call `t.close` must be executed from the body of `autoclose`. At that location of the program, the fact that `close` is defined in the object `t` is known because of the structural type. A Scala method call is normally directly compiled to an equivalent JVM method call instruction. This

ICOOOLPS '09 Genova, Italy

instruction requires that the JVM knows beforehand that the receiver of the call can respond to it — for reasons of performance. Because structural types are not part of the JVM, it does not know that `t` can respond to `close`. This shows that calls to methods which are statically defined as members of a structural type (structural method calls) cannot simply be compiled to a JVM method call. Instead, they require a different compilation technique that bypasses the constraints of the JVM. Two families of compilation techniques can be used to this end: generative techniques and reflective techniques.

Generative techniques create Java interfaces to stand in for structural types on the JVM. The complexity of such techniques lies in that all classes that are to be used as structural types anywhere in the program must implement the right interfaces. When this is done at compile time, it prevents separate compilation. When that is done at runtime, it requires to adapt dynamically objects to interfaces, which is expensive and complex.

Gil and Maman's Whiteoak [1] is an extension to Java with structural types that are compiled using a generative technique. Whiteoak does not modify classes to implement interfaces at compile time; instead, it uses ASM, a bytecode generation framework, to create wrapper classes at runtime. These wrappers implement the interfaces corresponding to structural types and forward all method calls. Whenever a structurally defined method is to be called on an object, a wrapper for the structural type is generated and the method is called on the wrapper instead of the object. The wrapper then delegates the call to the original object. A different wrapper class is needed for every structural type — because its interface changes — and for every type of delegation object — because the code of the forwarder methods is tailored for one specific delegation. Whiteoak uses a global cache strategy to reduce the number of wrappers that must be generated.

Reflective techniques replace JVM method call instructions with Java reflective calls. Reflective calls do not require a priori knowledge of the type of the receiver: they can be used to bypass the restrictions of JVM method calls. The complexity of such techniques lies in that reflective calls are much slower than regular interface calls. Optimisations are required for this technique to be competitive.

The compilation technique of structural types in Scala that is discussed in this article is based on a reflective technique. Its implementation is presented in section 2, which describes caching strategies used to optimise performance of reflective calls and discusses its limitations. Section 3 first analyses the performance of Scala's compilation technique of structural types. A second part of this section compares the performances of Scala's reflective compilation technique to that of Whiteoak's generative technique. Section 4 summarises the relative merits of both compilation techniques, from the point of view of their performance, and from that of their implementation.

2. IMPLEMENTATION IN SCALA

The process of compiling structural types is threefold:

1. The type system checks that structural types' declarations are valid, and that their uses conform to the type discipline of the program.
2. When Scala types are "erased" to JVM types, structural types disappear. All structural method calls are no longer valid in the erased type discipline. They are replaced by a special "apply dynamic" method call.
3. Every "apply dynamic" call is compiled to Java reflection so as to behave as a proper call on a structural value. The necessary caching infrastructure is added.

Type Checking.

In Scala, it is straightforward to support type checking of structural types: structural types are a generalisation of refinement types, which have been part of the language from the start. Like structural types, refinement types allow to add constraints on the members of an existing type. However, they require the constraints to be on the type of a member that is inherited: no new members may be defined structurally. To support structural types, the Scala type checker merely had to see a single test removed.

Erasure.

In order to understand why structural types have to be "erased", and why some method calls become invalid afterwards, it is necessary to discuss the role of erasure in the Scala compiler. JVM bytecode is annotated with Java types: the JVM will check these types and will reject the program if they are not valid. A Scala program compiled to JVM bytecode must be annotated with Java types that are valid and will not be rejected by the JVM. Erased types are Java types that define a *type discipline* on the program that is *equivalent* to, although less precise than that of the program typed with Scala types. If the process of erasure maintains this equivalence, JVM instructions — most importantly method calls — can be directly used to compile Scala programs: a method call which is valid in Scala is also valid in JVM bytecode. Erasure is described in detail in section 3.6 of [4].

Before structural types were introduced, all of Scala's type constructs could be erased in some way. For example, a mixin type such as "`A with B`" is erased to `A` — all objects of this type must then inherit `A` and implement `B` so that they can be cast to `B` when necessary. However, there is no Java type to which a structural types can be erased. The simple structural type "`Object { def f: Int }`", if erased to `Object`, will see the JVM reject a call to `f` because `Object` has no such method. A solution is to generate for every structural type in the program an interface that represents it — like "`interface FStruct { int f(); }`" — and have all of the objects of this structural type implement the interface. That is, in a simplified form, the "generative" technique used by Whiteoak. Scala's "reflective" technique does not require an erasure model that maintains the type equivalence property. Instead, the compiler will have to generate instructions that allow calling methods on objects having an erased type that does not contain the definition of the method.

"Apply dynamic" calls.

Method calls are represented in Scala's abstract syntax tree as `Apply` nodes. Normally, these nodes are compiled to JVM method call instructions; that is possible because of erasure, as we explained in the previous paragraph. During erasure, when the compiler encounters in the abstract syntax tree an `Apply` node that cannot be compiled to a JVM method call because the receiver has a structural type, it will replace the `Apply` with an `ApplyDynamic`. The observed semantics of `ApplyDynamic` are that of `Apply`, but it is com-

piled so that the receiver of the call can have an erased type that does not define the method, as will be explained below.

As a reminder, the dispatching semantics of method calls in Scala, like in most popular object-oriented languages, is dynamic on the receiver's type but static on the parameters' types. Consider for example the method call "`a.f{x}`". The actual instances used at run time for `a` and `x` may have types that are compatible with, but different from the types statically assigned to `a` and `x`. The implementation of `f` that should be used is that defined in the class corresponding to `a`'s dynamic type. If, for a given call site, the type of the receiver instance changes, the implementation of the method must be modified. On the other hand, if `f` is overloaded, the alternative that should be used is statically defined, based on the static types of `f`'s parameters.

It may be noted that we present `ApplyDynamic` as a means to compile structural types. However, we believe that it is a more general solution that can be used for other types that bypass the JVM's type system.

Compilation of "Apply Dynamic" Calls

The technique that we present to compile `ApplyDynamic` nodes to JVM bytecode is based on Java reflection. At first sight, this transformation is trivial. Since reflective method lookup and application are purely dynamic, the JVM's type system won't be in the way. A method call of the form "`a.f(b, c)`" — where `a` is of a structural type, while `b` and `c` are of type `B` and `C` respectively — can be replaced by the compiler with the following expression.

```
a.getClass
  .getMethod("f", Array(classOf[B], classOf[C]))
  .invoke(a, Array(b, c)
```

In this implementation, the semantics of dispatching is preserved, mostly thanks to the fact that Java reflection supports it directly. The `getMethod` call is sent to `a`'s dynamic class, routing the lookup to the right place. The static types of the parameters, obtained with `classOf` operators, are sent to `getMethod` in order to select from overloaded alternatives.

Caches.

This naive compilation technique cannot be considered because its performance is not acceptable in practice; a purely reflective method call is about 7 times slower than a regular call (on JVM 1.6). However, about 4/5 of the compiled expression's complexity lies in the sole `getMethod` operation. If the method is somehow already available and only `invoke` and `getClass` are used, the reflective implementation is "optimal" and a method call may be only two times slower than a regular call.

Caching brings the performance of structural method calls closer to the optimal performance than the naive implementation. Of course, it never allows for an optimal performance since there will always be cache misses — at least once for the first call. We will discuss compilation techniques using two different caching strategies, which approach the optimal situation quite closely in realistic cases. In this discussion, and in section 3 where the performance of caching is assessed, we will use the following caching techniques.

0c No caching. This is a slight variant of the naive, purely reflective, compilation techniques above, with the array of static parameter types precalculated.

1c Monomorphic inline caching is a technique where only a single method is cached for every call site.

Nc Polymorphic inline caching is a technique that caches a method for every receiver type at the call site.

Monomorphic caching.

A call site is monomorphic if the type of the method call's receiver stays the same throughout the execution of the program. Many call sites are monomorphic in practice. Monomorphic inline caching takes advantage of this property and caches one method implementation per call site. If the call site is in fact monomorphic, the same implementation can be reused every time at practically no cost. If the site is not monomorphic, the cached implementation will be discarded every time the receiver type changes, and a new implementation will be obtained using `getMethod`.

For every dynamic call site of a class, the 1c implementation adds a static method called `dynMethod` — plus an identifier to make that name unique. `dynMethod` takes the class of the receiver and returns a method implementation that corresponds to the call site and to the receiver. A call site of the form "`a.f(b, c)`" is replaced by the compiler with the following expression.

```
dynMethod(a.getClass).invoke(a, Array(b, c))
```

We wish to generate a `dynMethod` that can, for monomorphic call sites, return immediately. Here is how `dynMethod` is implemented for the above call site.

```
[static]
def dynMethod(forReceiver: JClass[_]): JMethod = {
  if (dynMethod$Class != forReceiver) {
    dynMethod$Method =
      forReceiver.getMethod("f",
        Array(classOf[B], classOf[C])
      )
    dynMethod$Class = forReceiver
  }
  dynMethod$Method
}
```

The static variables `dynMethod$Class` and `dynMethod$Method` contain, respectively, the class of the previous receiver and the looked-up method for that class. Whenever the previous receiver class differs from the current one, `dynMethod$Class` and `dynMethod$Method` are recalculated, at considerable expense of time. Otherwise, `dynMethod$Method` is returned, at the expense of a test and a variable dereference. The latter case is almost equivalent to immediately returning, and gives to the call site a performance close to the optimal.

Polymorphic caching.

A call site is polymorphic if the method call's receiver type changes over the lifetime of the program. Polymorphism has many dimensions: its degree — bimorphic, trimorphic, etc. — describes how many different receiver types can be observed at that call site throughout the program's execution; its intensity describes how frequently the call's receiver type changes. A call site may have a high degree of polymorphism but may remain quasi-monomorphic for most of the program's execution, for example if polymorphism at that point is linked to initialisation. It is, for obvious reasons, on highly intensive polymorphic call sites that it is the hardest

to obtain a performance close to that of the optimal compilation. In what follows, and in section 3, we discuss worst case polymorphic call sites; reality will usually lie somewhere between the worst case and a monomorphic site.

The technique that we use for polymorphic inline caching is based on that proposed in [2], using JVM objects instead of low-level memory blocks.

The technique generates a `dynMethod` method for every call site, and the call site itself refers to that method. Instead of caching a single pair made of the receiver's type and the method's implementation, `dynMethod` caches a list thereof. When looking-up the implementation of the method at a call site, and if the receiver type was never encountered before, `dynMethod` will use `getMethod`, as in the naive implementation. The receiver class and method implementation pair will be appended to the front of the list. A method's implementation already in the list will be reverted from the list.

The list used to store method implementations is a simple linked list. Searching it has a complexity of $O(n)$ (n being the length of the list), adding to it has a complexity of $O(1)$. The latter task is a lot less common than the former; using a linked list may not be optimal when compared with a binary search tree, for example. We did not attempt to evaluate the performance of a cache backed by a binary tree. It must be noted that, contrary to most faster data structures, some implementations of linked lists can be used as caches in a multithreaded environment without synchronisation. In a parallel environment, a conflict leads at worst to one implementation obtained by `getMethod` being lost; it will have to be recalculated the next time it is used.

Exception handling.

When an exception is thrown in a method that is called reflectively, `invoke` wraps it in an `InvocationTargetException`. That changes the semantics of method calls. The problem is trivially circumvented by catching the reflective exception from the `ApplyDynamic` call site, and then rethrowing the original exception, which can be obtained from the `InvocationTargetException`.

Boxing of native values.

Java reflection only works with objects, not native values like `int`, `float`, or `boolean`. If a method requires a argument of type `int`, for example, reflectively calling `invoke` on it will require a boxed integer (instance of `java.lang.Integer`). Similarly, if a method returns a `boolean`, for example, the result of the reflective call will be a boxed boolean (instance of `java.lang.Boolean`). The compilation of `ApplyDynamic` must take that into account.

The Scala language has a purely object-oriented unified type system with no distinction between native values and objects. `Any` is a type that is compatible with both native values and objects. A type variable (with an implicit `Any` upper bound) can be instantiated to either an object type or a native value type. Therefore, both `Any` and type variables must be erased to a type that the JVM recognises as compatible with all instances. `Object` is the closest approximation of such a type. However, using `Object` requires all native values to be boxed when they are referred to with a type that exceeds Java's `Object` in generality. Early versions of Scala used a custom boxing scheme that was incompatible with Java reflection. The compilation of `ApplyDynamic` required constant unboxing of Scala boxed values and reboxing to Java ones, and vice-versa. The performance lost because of that was very noticeable. Changing Scala to use the same boxing technique as Java solved that problem.

Native bytecode operations.

Another difficulty caused by Scala's unification of types is that operations like integer addition (`+`) on native values, for which the JVM uses bytecode instructions, are represented in Scala as methods. The normal compilation process has the bytecode generator recognise these "fake" method calls, and rewrite them to the corresponding instructions. Let us consider a program where a native value is referred to as a structural type, as in the example below.

```
def g(x: Any{ def + (i: Int): Int }) = x + 2
```

From Scala's perspective, "`x + 2`" is considered as "`x.+(2)`". Since `+` is a structurally defined method, it must be compiled as an `ApplyDynamic`. The `getMethod` call will lookup a `+` method on `x`'s class and will fail — `x` is boxed to a `java.lang.Integer`, which does not have a `+` method.

Instead, the `ApplyDynamic` transformation detects any fake method — using a list of such methods — and generates a call to a static utility method that implements the fake method's behaviour. Of course, a fake method like `+` may actually be dispatched to an instance that does, in fact, implement it. Therefore, the call to the utility method for the addition must be guarded by a dynamic type check: if the call's receiver is of type `java.lang.Integer`, the utility method should be used; if not, the call must be dispatched like a normal `ApplyDynamic`.

Type parameters.

To call a method, `invoke` must know the static types of the method's parameters, which are passed as an array of classes. These types are used to maintain correct dispatch semantics and to select the right method if its name is overloaded. When `invoke` is used to implement an `ApplyDynamic`, the compiler must know the signature of the structurally defined method that is being called. In general that is easy, as in the example below.

```
def g(x: { def f(a: Int, b: List[Int]): Int }) =
  x.f(4, List(1,2,3))
```

The declaration of the structural type of `x` contains the static types of both parameters of `f`. These types can be used to generated the reflective call to `invoke`.

There is no problem either if the parameter types of the structurally defined method are type variables, as long as the variables are declared as part of the method definition.

```
def g(x: { def f[T](a: T): Int }) =
  x.f[Int](4)
```

When `g` is called, its argument `p` will be an object that contains a method with the signature "`x[T](t: T)`". Erasure changes type variables to their erased upper bound, which is `Object` in `T`'s case. Erasure will make sure that `f`'s parameter is boxed when it is a native value. This is the expected behaviour in such a situation: `invoke` is called with `Object` as the static type for the `a` parameter.

A problem arises if the parameter types of the structurally defined method are type variables that are declared outside of the scope of the structural type.

```
def g[T](x: { def f(a: T): Boolean }, t: T) =
  x.f(x.t)
g[Int](new { def f(a: Int) = true }, 4)
g[Any](new { def f(a: Any) = true }, 4)
```

The type variable `T` is instantiated to a concrete type every time `g` is called. The static type of `f`'s parameters therefore changes for every call to `g`. On the other hand, the transformation of `ApplyDynamic` for `x.f` is done only once, in the body of `g`, no matter what type `T` will eventually be assigned to. The value of type variables are not available at runtime so that `ApplyDynamic` cannot be compiled in a way that reconstructs the static types of the method's parameters at runtime.

A similar issue arises if the parameter types of the structurally defined method are type variables that are declared as type members of the structural type.

```
def g(x: { type T ; def t: T ; def f(a: T): Boolean }) =
  x.f(x.t)
g(new { type T = Int; def t = 4; def f(a:T) = true })
g(new { type T = Any; def t = 4; def f(a:T) = true })
```

Like before, the `ApplyDynamic` for `x.f` cannot be compiled because it depends on the static type of `T`. In this situation, `T` changes every time the parameter `x` of `g` implements the structural type using another value for type `T`.

If Scala were to support runtime types of some form, these problems could be overcome. For the time being, however, Scala rejects problematic type variables for structurally defined method's parameters.

3. PERFORMANCE

All the tests of this section have been run using a Java HotSpot 1.6.0_07 64-bit Server VM on an iMac computer (2.33 GHz Core 2 Duo with 4MB L2, 2GB Memory) running Mac OS X 10.5.6. All times are averages over multiple executions, and their variances have been checked to be insignificant — typically in the order of 1/100 of the average. For tests involving pseudo-random values, values change between executions.

In Scala

Only a fraction of method calls will be structural in real-life scenarios. The first performance benchmark that we discuss is a functional implementation of merge sort, which uses structural types. Such code is representative of the performance of code that heavily depends on structural types.

Functional merge sort.

To implement a functional merge sort algorithm for a list, a comparison operator between elements of the list is needed. On solution is for all objects that are to be merge sorted to implement the `Comparable` trait in Scala's library. But if merge sorting is to happen on objects that are not ready for it — that is, have not implemented `Comparable` — structural types can be used instead. This is the signature of the `mergeSort` method when using an interface:

```
type ComparableList =
  List[Comparable[Any]]
def mergeSort(elems: ComparableList): ComparableList
```

When using structural types, the signature of the method remains identical; only the type is declared differently:

```
type ComparableList =
  List[{ def compareTo(that: Any): Int }]
```

The implementations of both methods are identical. It requires $O(n \log n)$ merges on average. Each merge is composed of three calls for dereferencing list heads or tails, one comparison (a call on a structurally defined method), and one concatenation (which instantiates an object).

Figure 1 charts the time required to order 2000 times a list of 1000 pseudo-random elements. For *monomorphic* measurements, the list contains only instances of a single class. That means that the structurally-defined comparison method can always be called on the same implementation. For *bimorphic* measurements, the list contains instance of two classes at even, respectively odd positions. That forces the implementation of the method to be changed whenever the receiver object of the comparison method changes. For both monomorphic and bimorphic lists, the performance of different caching techniques is measured. S is the reference situation where a `Comparable` interface is used instead of structural types, and where regular interface calls are used instead of `ApplyDynamic` calls.

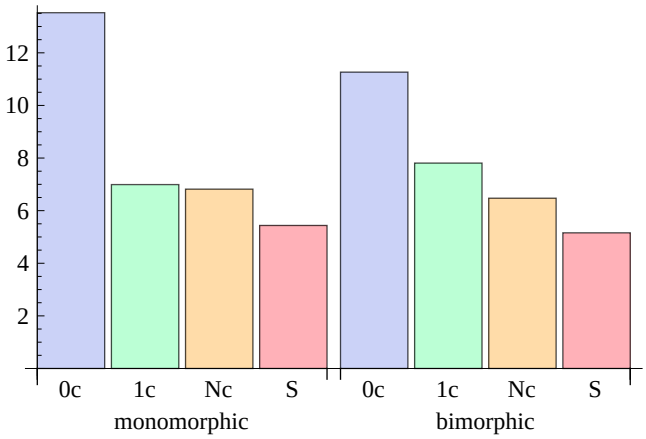

Figure 1: Execution time, in seconds, of the merge sort algorithm.

As expected, there is a performance penalty for using structural types when compared to using a `Comparable` interface; in this case the time is around 25% more for Nc caching. If we assume that un-cached structural method calls are about 10 times slower than regular interface calls, our data indicates that calling the comparison method represents about 1/8 of the test's execution time in the reference case.

In the monomorphic case, Nc and 1c exhibit similar performances, while 0c's performance is, predictably, more than twice as slow as regular interface calls. In the bimorphic case, the advantage of Nc becomes evident. That is not surprising as 1c must, in average, revert to a full method lookup equivalent to 0c, for half of the calls. The difference in performance between the mono– and polymorphic cases for 0c may be attributed to side effects of just-in-time compilation.

Single method call.

The test repeatedly calls a structurally defined method in a tight loop. The degree of polymorphism at the call site varies from one to four — from mono (morphic) to

quadri (morphic). Figure 2 charts the time required to call a method 10 million times. Bi- and quadrimorphic cases are worst-cases, where the class implemented by the receiver changes for every method call. To obtain polymorphism at the call site, the call's receiver is looked-up in an array, changing the index into the array, modulo the degree of polymorphism, for every iteration of the loop. Profiling this operation did not reveal the relative cost of this operation.

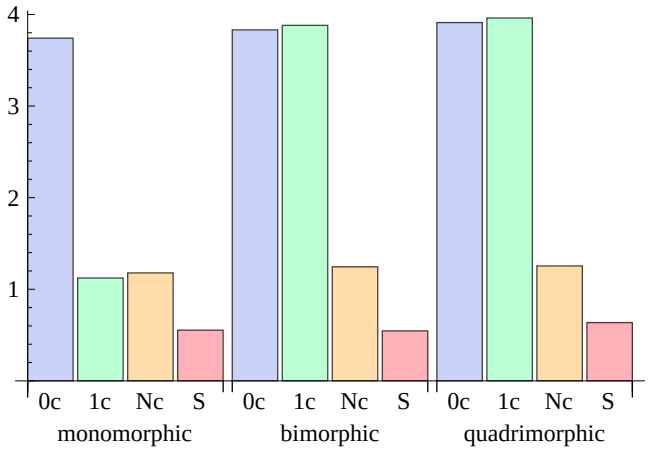

Figure 2: Execution time, in seconds, in relation to polymorphism degree of call site.

As was previously mentioned, the uncached implementation (0c) of `ApplyDynamic`, when compared with a regular interface call (S), is 7 times slower. When polymorphic inline caching is used (Nc), the slowdown is reduced to between 2.0 and 2.3 times, depending on the degree of polymorphism. The situation for monomorphic inline caching (1c) is equally favourable for a monomorphic call site, but the performance is as bad as 0c for polymorphic sites. The performances of the three caching strategies for different degrees of polymorphism are perfectly coherent with their implementations. If we assume that the time of a method call in Nc is entirely composed of the execution of the `invoke` method, comparing Nc and 0c shows that the time of a full reflective method call is distributed between 30% in the `invoke` call and 70% in the `getMethod` call.

Figure 3 represents the same data as figure 2, but the JVM is run in such a way that it does not use just-in-time compilation, only interpreting the bytecode. Whilst this data is not representative of real performance, it may be considered as an upper bound for the slowdown caused by structural types, and shows how effective JVM 1.6's just-in-time compilation is at optimising them. In the interpreted case, a regular method call is 7.6 times faster than for the Nc implementation, whilst when just-in-time compilation is used, the difference is of only 2.1 times.

Megamorphic call sites.

Some call sites have a degree of polymorphism much higher than that used in the previous test. Such sites are rare — Hölzle et al. [2] report that no call site in their real-life benchmarks have a degree of polymorphism of more than 10 — but they can seriously harm the performance of polymorphic inline caching. Figure 4 graphs the time (y-axis) required to call a method 10 million times with respect to the degree of

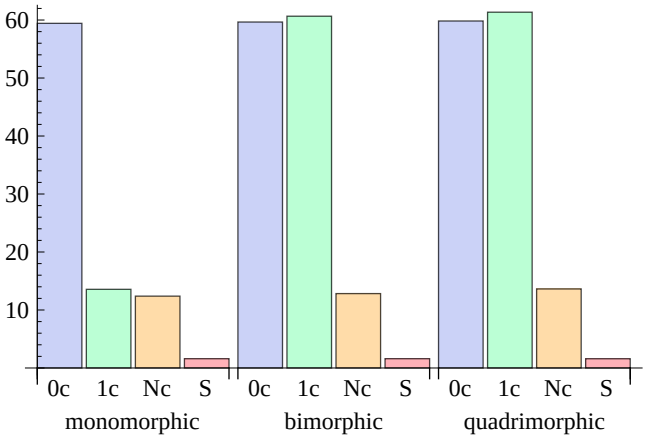

Figure 3: Execution time — with JVM just-in-time compilation disabled (interpreted) — in relation to the polymorphism degree of call site.

polymorphism of the call site (x-axis).

In Nc, the length of the cache n is eventually equal to the degree of polymorphism of the call site. Searching the cache has a complexity of $O(n)$ so that the worst-case performance of Nc will decrease linearly with the degree of polymorphism. On the other hand, the performance of 0c is expected not to change with the degree of polymorphism. The results do show a slowdown for the latter case, possibly because of variations in the effectiveness of just-in-time compilation. However, this slowdown remains lower to that of Nc so that the performance of the two implementations become equal when the call sites have a degree of polymorphism equal to about 180. At that point, the performance of 0c and Nc are equivalent because Scala's implementation of Nc automatically reverts to 0c when reaching that threshold.

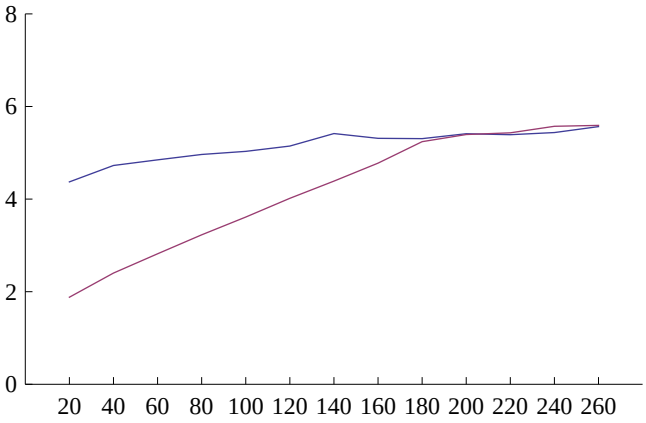

Figure 4: Execution time, in seconds, in relation to the degree of polymorphism of the call site. Upper line is for 0c, lower line is for Nc.

Interpretation.

The results reported above show that the Nc implementation, using polymorphic inline caching, is either significantly superior or roughly equivalent to the other implementations in all situations. Even for monomorphic call sites, the 1c

implementation that is designed with such call sites in mind is not significantly better than Nc.

Scala and Whiteoak

Gil and Maman have compared [1] the performance of structural method calls in Whiteoak with the performance of regular interface calls. Their results show that in the best case — when the fast "primary cache" can be used — their implementation allows for constant-time structural method calls that are about as fast as interface calls. However, a start-up cost of about 200μs is incurred before the first structural call is executed. They also report that the worst case scenario is about 7 times slower than the best case. Sufficiently small performance-critical areas are covered by the primary cache and will be close to the best-case performance. For most real-life applications, however, that would rarely be the case.

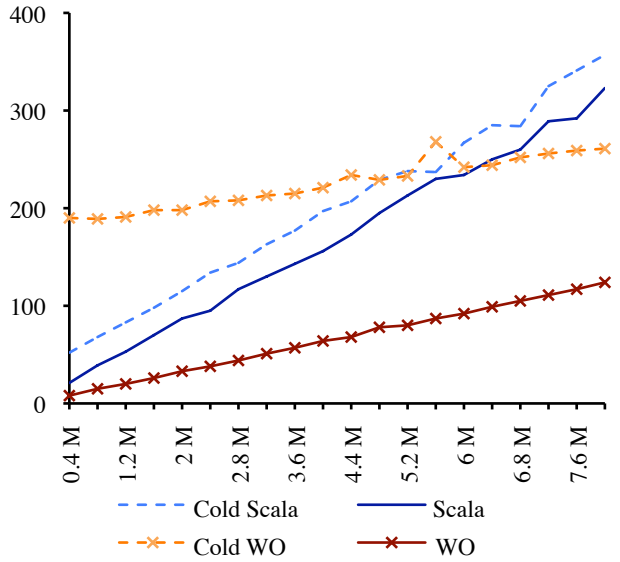

Figure 5: Execution time, in microseconds, in relation to the number of calls.

Figure 5 shows the results obtained in our own tests, comparing the performance of Whiteoak's best case with that of Scala's Nc implementation for a monomorphic call site. Results for polymorphic call sites do not differ significantly. The observed times are linear in the number of calls: they confirm the constant call times reported by Gil and Maman. To understand better the initial start-up cost of Whiteoak, the tests are run twice, without restarting the JVM, for different call sites. The reported "cold" times are for the first structural call site, and include the infrastructure's initial start up overhead. The "hot" times are for the second call site, and only include the overhead incurred at a given call site, if any. A best-fitting linear regression ($y = ax + b$) is calculated for every data set. The value at which the regression line intercepts the y-axis (b) is an approximation of the overhead; the slope of the line (a) is an approximation of the time for a single structural call.

	Overhead	Call time
Cold Scala	36μs	38ηs
Scala	3–6μs	38ηs
Cold Whiteoak	180μs	11ηs
Whiteoak	2μs	15ηs

A comparison of hot Scala with hot Whiteoak shows that both have relatively low overheads when calling a site for the first time — equal to about 80,000–160,000 method calls for Scala, 130,000 for Whiteoak. The call time for Scala is about 2.5 times slower than that of Whiteoak, which is in line with the result reported above. The initial start-up overhead of Scala's implementation is one order of magnitude greater than the overhead at a subsequent call site. This overhead's origin is likely to come from the infrastructure for polymorphic inline caching, as this is the only code that is shared amongst call sites. The initial start-up overhead for Whiteoak's implementation is 5 times greater than that of Scala, and can probably be explained in part by the time necessary to initialise ASM, the bytecode manipulation framework required by Whiteoak. Scala's call times are 3.4 times slower than cold Whiteoak's call times, which is a result at odds with those comparing Scala's times with regular interface calls. A possible explanation is that just-in-time compilation differs between this test and those obtained on Scala alone.

Interpretation.

The results that are reported above show Whiteoak's implementation of structural types to be superior to Scala's implementation. They indicate that Scala is in between 2.5 and 3.5 times slower than Whiteoak when considering the performance in tight loops, a situation where Whiteoak excels. Therefore, Whiteoak's implementation is preferable if structural method calls are to be used in a hot spot of a performance-critical algorithms. On the other hand, these results very much depend on Whiteoak using its small primary cache. Since Whiteoak's cache is shared amongst all call sites — contrary to Scala's caches which are specific to each call site — the performance of Whiteoak's implementation will diminish when a program's execution cost is not contained in a tight loop. Gil and Maman report the performance of Whiteoak in a situation that "mostly hits the secondary cache" to be 7 times smaller than the best-case performance. Because of that, in such a situation, Scala's technique using reflection and polymorphic inline caching will likely be faster.

4. SUMMARY

There is a performance penalty associated with the use of structural types on the JVM, but a reflective compilation technique that uses efficient caching brings this cost to acceptable levels for all but the most critical programs. Generative compilation techniques such as that of Whiteoak have, in principle, the potential for even higher performance. On tight loops, a situation that is important in the context of time-critical algorithms, generative techniques are clearly superior. On the other hand, reflective implementations may be equally good and possibly better when the cost of using structural types is distributed throughout the program. From a performance's point of view, choosing one technique or the other is a question of balance: generative compilation optimises tight hot-spots of structural method

calls whilst reflective compilation may lead to higher overall performance.

From the point of view of the implementation, reflective techniques are simpler. They only require changing the compilation scheme for structural method calls and a small runtime library that implements polymorphic inline caching. Generative techniques, on the other hand, require changing the compiler to generate interfaces for each structural type, changing the compilation scheme for structural method calls — to adapt the receiver to the structural type using a wrapper — and require an infrastructure to generate wrapper classes dynamically and to cache them. A dependency on a code generation library is added to all programs. Furthermore, loading generated classes at runtime requires access to the program's class loader, which may not be available when running on environments such as application servers.

There is only one language restriction that is related to reflective compilation: the limitations on type variables in Scala's structural types. Arguably, this restriction has more to do with erasure than with the implementation of structural types; if the effects of erasure were hidden by dynamic types, this restriction may be lifted. Gil and Maman's article does not detail the restrictions that the generative technique imposes upon structural types, although Whiteoak does not allow "generic structural types", a restriction that is similar to that imposed on Scala's type variables. We can assume that Whiteoak would have similar difficulties with type variables in structural types as Scala has. Other important features, such as structural arrays, are not available with Whiteoak's generative technique. Whether generative compilations techniques could be used to compile Scala's structural types, which are part of a much more complex type system and language than those of Java, remains an open question.

Reflective compilation techniques are probably not noticeably slower than generative ones in real applications. However, reflective techniques are simpler to implement, have fewer dependencies and have been shown not to restrict the language. In this context, it is worth noting that a recent overhaul of Whiteoak's source (version 2.1) did not retain the generative compilation infrastructure, leaving it to be implemented later, and uses a simple un-cached reflective implementation.

5. REFERENCES

[1] J. Gil and I. Maman. Whiteoak: introducing structural typing into java. In *Proceedings of the 23rd OOPSLA conference*, pages 73–90, 2008.

[2] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP '91*, number 512 in LNCS, pages 21–38, 1991.

[3] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *ECOOP 2008*, number 5142 in LNCS, pages 260–284, July 2008.

[4] M. Odersky. *The Scala Language Specification*, November 2007.

[5] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *ECOOP 2003*, number 2743 in LNCS, pages 201–224, 2003.

[6] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Press, 2008.