Extending Scala with Records

Design, Implementation, and Evaluation

Olof Karlsson
A3J Consulting AB
Stockholm, Sweden
olof.karlsson@a3j.se

Philipp Haller
KTH Royal Institute of Technology
Stockholm, Sweden
phaller@kth.se

Abstract

This paper presents a design for extensible records in Scala satisfying design goals such as structural subtyping, typesafe polymorphic operations, and separate compilation without runtime bytecode generation. Using new features of Scala 3, the design requires only minimal, local changes to the Scala 3 reference compiler Dotty as well as a small library component. Runtime performance is evaluated experimentally using a novel benchmarking suite generator, showing that the design is competitive with Scala 2's cached reflection for structural field access, and excels at immutable extension and update operations.

CCS Concepts • **Software and its engineering → Language features**; *Compilers*; Software performance;

Keywords Scala, records, structural typing

ACM Reference Format:
Olof Karlsson and Philipp Haller. 2018. Extending Scala with Records: Design, Implementation, and Evaluation. In *Proceedings of the 9th ACM SIGPLAN International Scala Symposium (Scala '18), September 28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3241653.3241661

1 Introduction

A record is generally understood to mean a labeled product datatype, consisting of a set of label-value pairs called fields. Under this broad definition almost every practical programming language provides a similar construct, whether it is called a record (Haskell, F#, SML), a struct (the C-language family), or a class (most OOP languages). In this paper, however, we focus exclusively on records with *structural typing*.

While structural typing is common in the academic literature, the majority of the type systems of mainstream programming languages are nominal. This might gradually be changing, however, as structurally typed languages such as TypeScript and Go are gaining in popularity.[1] Structural typing can provide more flexibility than nominal typing and is especially well suited for handling semi-structured data such as JSON and XML. This makes structurally typed, efficient records an attractive choice for a wide range of applications, from rapid web-development to large-scale data analytics.

Scala supports both nominal typing for object-oriented constructs such as classes, traits, and singleton objects, as well as structural typing in the form of refinement types. The structural typing is reserved for instances of already-existing classes, however (breaking separate compilation), and there exists no language-provided constructor for record literals, nor support for typesafe polymorphic operations such as record extension. Furthermore, the runtime performance of structural member access is debated, as it utilizes reflection on the JVM. Efforts have been made to improve the situation, but so far library implementations of records in Scala 2 have been unsatisfactory, relying on the experimental macro system. These macros are now deprecated and are no longer supported in Scala 3.[2] Even if the same style of macros were to be supported, macro-based solutions often turn out to be hard to debug and have poor support for static code-analysis tools such as IDEs.[3] In addition, existing libraries have approach-specific limitations, such as no support for typesafe record extension in the case of scala-records [Jovanovic et al. 2018] and Compossible [Vogt 2015], or an enforced ordering of record fields in the case of Shapeless [Sabin et al. 2018].

In this paper we present a new, native approach to records in Scala that for the first time simultaneously satisfies the following design goals:

1. structural subtyping with unordered fields;
2. typesafe extensibility, also in the setting of parametric polymorphism;
3. separate compilation support;
4. no bytecode generation at runtime;
5. only small, local changes to Scala's type system; and
6. no dependency on the deprecated macro system.

Scala '18, September 28, 2018, St. Louis, MO, USA

[1] For example, see http://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/
[2] See https://www.scala-lang.org/blog/2018/04/30/in-a-nutshell.html
[3] For better macro support, the IntelliJ IDE requires creating special IDE plugins. See https://blog.jetbrains.com/scala/2015/10/14/intellij-api-to-build-scala-macros-support/

The approach builds on new features of Scala 3, the next major version of Scala, as implemented in the reference compiler Dotty. As a result, the compiler extensions required by our design are rather small and limited. Essentially, we propose two new compiler-synthesized type classes, coupled with a small library component. In summary, the contributions are as follows:

Contributions

1. A new design of records for Scala which overcomes the limitations of previous approaches. To the best of our knowledge it is the only approach satisfying all of the above design goals.
2. A complete implementation[4] of our approach as an extension of the Scala 3 reference compiler, integrated with the full Scala 3 language.
3. A novel micro benchmark generator, called Wreckage[5], with support for multiple mutually-incompatible JVM-based compilers, including Scala 2, Scala 3, and Whiteoak [Gil and Maman 2008]. It enables finegrained performance measurements on the industry-standard HotSpot JVM implementation, through integration with the JMH benchmarking harness.
4. An experimental evaluation of the runtime properties of various implementation strategies on the JVM, as well as a case study processing a real-world JSON dataset. Our results reveal, for the first time, how different implementation schemes for structural typing compare to related language constructs in Scala, such as nominally typed case classes.

The rest of the paper is organized as follows. In Section 2 we briefly introduce new features of Scala 3 used by our approach. Section 3 gives an overview of the record extension from the user's perspective. In Section 4 we describe the rationale and the design of the suggested approach. Section 5 presents benchmark results, comparing our approach to alternative record implementations. In Section 6 we discuss related work, and provide conclusions in Section 7.

2 Background

This section introduces implicits as well as Scala 3's new implementation of structural types. Other features of Scala 3 used in our design are introduced on the fly in later sections.

Implicits. With *implicits* Scala supports extensible, generic programming which enables encoding type classes as a pattern [Oliveira et al. 2010], among others. Methods may have multiple parameter lists, one of which may be marked as `implicit`:

```
def print[T](it: T)(implicit s: Show[T]): Unit =
  println("printing: " + s.show(it))
```

The type `Show[T]` of the implicit parameter `s` has a single method `show` which converts a value of type `T` to a string:

```
trait Show[T] { def show(t: T): String }
```

Invoking `print[S]` requires an *implicit value* of type `Show[S]` to be in scope, for example:

```
implicit val showCar: Show[Car] = new Show[Car] {
  def show(c: Car) = "Car of model " + c.model
}
print(new Car("Space Cruiser"))
```

The concrete implicit value is inferred by the type checker; ambiguity leads to a static type error. Using a *context bound* we can simplify the method signature of `print`:

```
def print[T : Show](it: T): Unit =
  println("printing: " + implicitly[Show[T]].show(it))
```

The context bound `: Show` expands to the implicit parameter list as shown above, but with a synthetic parameter name; `implicitly[S]` returns the unique implicit value of type `S` that is in scope at the invocation site.

Structural types. Scala 3 introduces an approach to structural types that enables programmable member access. Suppose `v` is a value of type `C { Decls }` where `C` is a class and `Decls` are refinement declarations. Then, if `C` implements a special trait `Selectable`, a selection `v.fld` is expanded to

```
(v: Selectable).selectDynamic("fld").asInstanceOf[T]
```

provided that `fld` is declared with type `T` in `Decls`.

3 Overview

This section provides a short demonstration of the capabilities of the introduced records, as seen from the end user's perspective in a tutorial-style REPL session.[6]

Record capabilities are imported from a new `records` module in the `dotty` package of the core library; a record with two fields `name` and `age` is constructed as follows:

```
scala> import dotty.records._
scala> val r = Record("name", "Rick") + ("age", 70)
val r: Record{name: String; age: Int}
  = Record(name=Rick, age=70)
```

The record type is represented as a structural refinement of the `Record` class, providing typesafe field access:

```
scala> r.name
val res0: String = "Rick"
scala> r.foo
1 | `foo` is not a member of Record{name: String; age: Int}
```

as well as structural *width and depth subtyping*:

```
scala> val s: Record{val name: Any} = r
val s: Record{name: Any} = Record(name=Rick, age=70)
```

Furthermore, existing records can be extended with additional fields:

[4] Available here: https://github.com/obkson/dotty/tree/dotty-records-final
[5] Available here: https://github.com/obkson/wreckage
[6] Due to space considerations some REPL output has been truncated.

```
scala> val u = r + ("grandson", "Morty")
val u: Record{name: String; age: Int; grandson: String}
  = Record(name=Rick, age=70, grandson=Morty)
```

and the same syntax allows (immutable) updates of existing fields:

```
scala> val older = r + ("age", r.age + 1)
val older: Record{name: String; age: Int}
  = Record(name=Rick, age=71)
```

Record extension is only allowed if the added field is guaranteed to not already exist on the updated record type, or if the added value has a subtype of the existing field's type. Otherwise it is a compile-time error:

```
scala> val e = r + ("age", "old")
1 |Cannot prove that Record{name: String; age: Int} is
  |extensible with String("age") ->> String.
```

Record extensibility is not limited to concrete record types but can be supported in polymorphic contexts as well by adding a type variable with one or more context bounds:

```
def center[R <: Record : Ext["x",Int] : Ext["y",Int]](r: R)
  = r + ("x", 0) + ("y", 0)
scala> val p = center(r)
val p: Record{name: String; age: Int; x: Int; y: Int}
  = Record(name=Rick, age=70, x=0, y=0)
```

Here, each context bound `Ext[L, V]` ensures that the record type `R` can be safely extended with a field with label `L` and type `V`.

4 Design and Approach

One of the main goals of the implementation is to introduce as few new concepts to the language as possible, and in particular not to introduce any new syntax. To that end, Scala 3's new `Selectable` trait provides an excellent starting point. Using a `Map` data structure to hold the record fields internally, a simple record implementation can be written in Scala 3 as follows:

```
case class Record(m: Map[String, Any]) extends Selectable {
  def selectDynamic(name: String) = m(name)
}
```

Such a record can then be instantiated by supplying the field map and giving the result a structural refinement type explicitly:

```
val r = Record(Map("name"->"Morty", "age"->14)).
  asInstanceOf[Record{val name: String; val age: Int}]
```

Since the `Record` class extends the `Selectable` trait the compiler automatically translates each subsequent field access on the structural type into a call to the `selectDynamic` method, casting the returned value to the type of the selected field (see Section 2). This simple implementation using nothing but native Scala 3 constructs already provides much of the functionality expected from records, such as structural typing with width, depth, and permutation subtyping and typesafe field access. However, the implementation does not provide *typesafe record creation* nor *extensibility*. Type safety is only guaranteed as long as the user assigns the correct type in the initial explicit cast, and there are no means to add more fields later and have those fields included in the result type. This paper addresses precisely these issues by adding compiler support for typesafe record extension. As shown in Section 3, typed record creation then comes for free as it can be expressed as a series of extensions of an empty record.

Extensibility. Scala 3's new intersection types can *almost* be used to express record concatenation at the type level. Type intersection is defined such that the intersection of refinement types `S` and `T` contains the union of the fields from both `S` and `T`. The result of concatenating a record type `Record{name: String}` with `Record{age: Int}` can therefore be typed as the intersection `Record{name: String} & Record{age: Int}` which is equivalent to `Record{name: String; age: Int}`. A naïve version of record concatenation can therefore be implemented as follows:

```
def concat[S <: Record, T <: Record](s: S, t: T)
  = new Record(s.m ++ t.m).asInstanceOf[S & T]
```

However, this definition is not sound. If records `s` and `t` both contain a field `f` with type `A` in `S` and with type `B` in `T`, type intersection is by definition applied recursively so that the resulting type `S & T` contains a field `f: A & B`. At the same time, the map concatenation operator `++` overwrites the value of `f` in `s` with the value of `f` in `t`. The result is a record with a field `f` of static type `A & B` but dynamic type `B` (or some subtype) and a runtime exception follows:

```
// val s: Record{f: Int} = Record(Map("f" -> 42))
// val t: Record{f: String} = Record(Map("f" -> v))
scala> val e = concat(s, t)
val e: Record{f: Int & String} = Record(Map(f -> v))
scala> e.f // Exception: String cannot be cast to Integer
```

In a setting without subtyping, this problem can be addressed by statically checking that the fields of `S` and `T` are disjoint. As noted by Cardelli and Mitchell [1991] however, subtyping introduces further difficulties. Even if the types `S` and `T` are statically known to be disjoint, the dynamic record values `s` and `t` might not be. In the above example, the record `t` has an equally valid typing `Record{}` which effectively hides the dynamic field `f` from the static type `T`. With this typing the record types `S` and `T` are disjoint and the result type is `Record{val f: Int}`. The value of `e.f` is still `"v"`, however, and a runtime exception follows.

Our proposed solution avoids these problems by a) only allowing a record to be extended by a single, statically known field at a time (allowing record *extension* rather than record concatenation), and b) only allowing a record to be extended if the resulting type can be expressed as an intersection type. The general rule here is that a record type `R` can only be extended with a field `f: B` if `f` is missing in `R`, or if `f: A`

exists in R it holds that B <: A so that B <: A & B and the updated record's new value can safely be typed as A & B.

This is accomplished by introducing two new compiler-synthesized type classes defined as follows:

```
trait Extensible[R <: Record, L <: String, V]
```

and

```
trait FieldTyper[L <: String, V] { type Out <: Record }
```

The `Extensible[R, L, V]` type class carries proof that the record type R can safely be extended with a field with label L and type V, and the `FieldTyper` type class makes it possible to still express the result of this extension as an intersection type by mapping the field to its corresponding record type `Out`. Note that the label is lifted to the type level using a type variable L which is expected to be a literal singleton type.

Using these two type classes, a polymorphic function `extend` can be implemented that only extends a record `r` with label `l` and value `v` if it is safe to do so:

```
def extend[R <: Record, V](r: R, l: String, v: V)(implicit
  ev: Extensible[R, l.type, V],
  ft: FieldTyper[l.type, V])
  = Record(r.m.updated(l, v)).asInstanceOf[R & ft.Out]
```

Both the `Extensible` and `FieldTyper` type classes require special compiler support to be automatically derived and instantiated when required. This is achieved by augmenting the implicit resolution process of the Scala 3 compiler, so that whenever an instance of `Extensible` or `FieldTyper` is required as an implicit argument, but cannot be found in the current scope, an algorithm inspects the type of the required instance and, if possible, instantiates it. Following certain derivation rules, the algorithm may also reject instantiating the type class, resulting in a compilation error. Scala already has similar language-provided type classes that are automatically instantiated when required as implicit parameters, such as the `ClassTag`, `TypeTag` and `Eq` type classes, and so we argue that this is a natural extension to the overall design of the compiler.

Again, consider the example of extending a record `r` of type `Record{name: String}` with an age field:

```
val e = extend(r, "age", 123)
```

Since the type `Record{name: String}` lacks a field with label age, it is safe to extend using type intersection and an instance of `Extensible[R, "age", Int]` is synthesized by the compiler. Furthermore, the compiler synthesizes an instance `ft` of type `FieldTyper["age", 123]` and calculates the path-dependent type `ft.Out` to equal the corresponding record type `Record{age: Int}`. The resulting type of extension is `Record{name: String} & Record{age: Int}` as desired.

Furthermore, extending a record with a field that already exists with an incompatible type is rejected:

```
scala> val e = extend(r, "name", 123)
1 |Cannot prove that Record{name: String} is extensible...
```

This restriction only applies to the static type of the extended record however, since it is safe to overwrite the value of a field hidden by widening:

```
scala> val e = extend(r: Record{}, "name", 123)
val e: Record{name: Int} = Record(Map(name -> 123))
```

Derivation Rules. An instance of `Extensible[R, "l", V]` is automatically synthesized by the compiler if R can be proven to be extensible with field `(l: V)` by a recursive case analysis on the type tree of R. For the complete set of rules, the reader is referred to the documentation;[7] here, we only cover the most interesting cases. Extending a record type R with field `(l: V)` is allowed if either

1. an instance of `Extensible[R,"l",V]` exists in scope;
2. R is an intersection type `A & B`, and extension is allowed for both A and B;
3. R is a type variable, and extension is allowed for its *lower* type bound;
4. R is a concrete refinement type `S{Fs}` where S is a subtype of `Record` and `Fs` a set of refinement declarations, for which it holds that
   a. `S{Fs}` has no member with name `l`, or
   b. the refinement declarations `Fs` contain a member with name `l` and type U, where V is a subtype of U.

As an example, consider the following function:

```
0| def f[R >: Record{val name: String} <: Record](r: R)
     (implicit ev: Extensible[R, "age", Int]) = {
1|   val s = extend(r, "ssn", "n/a")
2|   val t = extend(s, "age", 99) }
```

In the first extension on line 1 the type of `r` is the abstract type variable R, and there exists no evidence in scope that this type is extensible with the field `(ssn: String)`. However, the type variable R is lower bounded by the concrete refinement type `Record{val name: String}` for which rule 4a applies. The intuition here is that R is guaranteed to be a super type of this bound, and so it cannot contain any more fields, and in particular not a field with label `ssn` and a conflicting type. Thus, R is extensible and an instance of `Extensible[R, "ssn", String]` can safely be synthesized and passed to the `extend` function.

In the second extension on line 2 the type of `s` is the more complicated intersection type `R & Record{ssn: String}`. Following rule 2, extensibility must then be proven for both operands. In the case of R the implicit instance `ev` of type `Extensible[R, "age", Int]` is already in scope, and so rule 1 applies. In the case of `Record{ssn: String}` we again have a concrete refinement type for which rule 4a applies. Having proven that extension will interfere with no fields in R and no fields in `Record{ssn: String}`, the whole intersection is safe to extend and the `Extensible` is synthesized.

[7] Available here: https://github.com/obkson/dotty/blob/dotty-records-final/Records.md

Note that it is only the `Extensible` type class that has to be passed around to carry proofs of extensibility into polymorphic contexts. In contrast, instances of the `FieldTyper` type class *must always* be generated at the invocation site of the above `extend` method, since the type member `Out` of an already-existing instance would always be equal to its less informative upper bound `Record`.

Library Component Implementation. To allow the `Extensible` and `FieldTyper` type classes to refer unambiguously to `Record` types without restricting their usability to a single record implementation, the `Record` type is defined as a trait extending `Selectable`.

```
trait Record extends Selectable {
  def updated(name: String, value: Any): Record
}
```

The default concrete implementation of the `Record` trait uses the immutable hash map from Scala's collections library as its underlying data structure. Field selection through `selectDynamic` is implemented as a regular key-lookup, and the `updated` method implements immutable update by creating a new record with an updated internal map:

```
class MapRec(val _data: Map[String, Any]) extends Record {
  def selectDynamic(name: String) = _data(name)
  def updated(name: String, value: Any): Record =
    new MapRec(_data.updated(name, value))
}
```

To make the syntax for extending records more convenient, the `extends` method in earlier examples is provided as an extension method + that delegates the actual updating operation to the extended records implementation of `updated`, and assigns the correct type to the result:

```
implicit class RecOps[R <: Record](r: R) extends AnyVal {
  def +[V, S <: Record](l: String, v: V)(implicit erased
    ev: Extensible[R, l.type, V]
    ft: FieldTyper.Aux[l.type, V, S])
    = r.updated(l, v).asInstanceOf[R & S]
}
```

As an optimization, the implicit `Extensible` and `FieldTyper` parameters are marked as `erased`. This is a new feature of Scala 3 that indicates that the parameters are only needed to carry type information at compile time, but can safely be removed after type-checking to get rid of the overhead of passing them around at runtime.

Finally, a shorthand for the `Extensible` type class is provided in the form of the `Ext` context bound. This is implemented as a type alias that partially applies the `Extensible` type and returns a *type lambda* that takes a record type as its single argument:

```
type Ext[L <: String, V] =
  [R <: Record] => Extensible[R, L, V]
```

Since the returned type is parameterized by a single type argument it can then be used as a context bound using regular Scala syntax.

Intersection Type Merging. Although the intersection of two refinement types such as `Record{a: A} & Record{b: B}` by definition is equivalent to `Record{a: A; b: B}`, the current Scala 3 compiler only performs the actual merging under a limited set of circumstances. To keep record types smaller, reducing compile time and making them easier to read, our implementation makes one final modification of the type checker so that intersections of refinement types are always merged if possible.

Limitations. Under some circumstances Scala 3's type inference algorithm puts unnecessarily strict bounds on the operands of intersection types. In our proposed solution, the resulting type of extending `Record{a: A}` with a field `(b: B)` is the intersection type `Record{a: A} & Record{b: B}` which is then merged to `Record{a: A; b: B}`. If the expected return type is inferred or explicitly given as a type ascription such as `Record{a: A; b: B}`, however, the current Scala 3 compiler not only applies this bound to the intersection type as a whole, but also to each of the operands. A type error then results from the fact that `Record{a: A}` is not a subtype of `Record{a: A; b: B}`. This problem is believed to be solvable in future work by loosening the type inference rules for intersection types. In the meantime, it can be circumvented by either avoiding explicit type ascriptions or by wrapping the created record in an identity function `def id[T](x: T) = x` that guards the intersection type from type inference.

Our approach also inherits some limitations from Scala's current support for structural typing. For example, neither Scala 2 nor 3 allows recursive type aliases. This means that it is not possible to define recursive record types, and a type declaration such as

```
type Tree = Record{val c: List[Tree]}
```

results in an "illegal cyclic reference" error. This is a fundamental limitation of Scala's current type system, though, and beyond the scope of this paper to address.

5 Experimental Results

The runtime performance for various approaches to records was evaluated using Wreckage,[8] a novel benchmarking source-code generator built on top of the JMH microbenchmarking harness.[9] Given an abstract syntax description for a record implementation, Wreckage generates source code for JMH benchmarks of various common record operations such as creation, update, and field access. This significantly simplifies the otherwise tedious and error-prone process of manually writing benchmarks for records with a large number of

[8] Available here: https://github.com/obkson/wreckage
[9] See http://openjdk.java.net/projects/code-tools/jmh/

fields, and repeating the process for each approach-specific syntax. Furthermore, by generating source code rather than using, for example, macros, the executed code can easily be inspected and verified.[10] The supported languages are Java, Scala, and Whiteoak, and the generated code is compiled and built into a standalone executable jar using Maven.

JMH is the de-facto standard tool for micro-benchmarking on the JVM [Stefan et al. 2017], and helps with protecting against JIT compilation optimisations that would render the results invalid, such as constant folding and dead code elimination, as well as making sure the measurements are not affected by the system clock latency and granularity.

All benchmarks were executed on a 4-core 3.1 GHz Intel Core i7 CPU with 16 GB DDR3 RAM, running the 64-bit server HotSpot JVM version 8 (build 25.112-b16, mixed mode) on MacOS 10.13.

The benchmarks include the following approaches:

- Scala 3 Records: The suggested approach in this paper.
- Scala 3 Case class: For reference, using nominal typing.
- Scala 3 Trait fields: A class implementing one trait per field, achieving field access through interface calls on the JVM. See Section 6, "Interface Fields".
- Scala 2.12 Compossible 0.2: A macro-based library supporting unsafe extension using the same hash map as Scala 3 Records.
- Scala 2.12 Shapeless 2.3.2: Ordered records using heterogeneous lists (HLists).
- Scala 2.12 Structural: Using structural member access on an underlying class instance. See Section 6, "Reflection with Inline Caching".

5.1 Micro Benchmarks

The micro benchmarks measure mean steady-state execution time with 95 % confidence intervals using the statistically rigorous approach suggested by Georges et al. [2007]. Each micro benchmark is run until a window of 10 consecutive measurements shows a coefficient of variation (CV) below 0.02. Then steady state is assumed, and the window mean execution time is calculated. This process is repeated $n = 10$ times in independent JVM forks, and the mean steady-state execution time is the mean over the results from each fork. The confidence interval is calculated using the students t-distribution with $n - 1$ degrees of freedom.

Creation Time against Record Size. The time it takes to create a record was measured as a function of the size of the created record, where the size denotes the number of integer fields. The results are presented in Figure 1, and reveal significant differences in performance between the benchmarked approaches.

Except for the case class baseline, Shapeless requires the least creation time. This is to be expected as the linked HList is constructed by simply prepending each element to the previous list in constant time. Both Compossible and our approach use a hash map as underlying data structure and implement record creation as a series of extensions. The difference in performance can be explained by the fact that Compossible's extension is implemented as record concatenation. This means that every added field must first be converted to a complete record instance that is then merged with the extended record. Our approach on the other hand adds the fields directly and only has to instantiate records to hold the result of each extension.

[10] Samples of the generated benchmarking code can be found at https://github.com/obkson/wreckage/blob/master/README.md

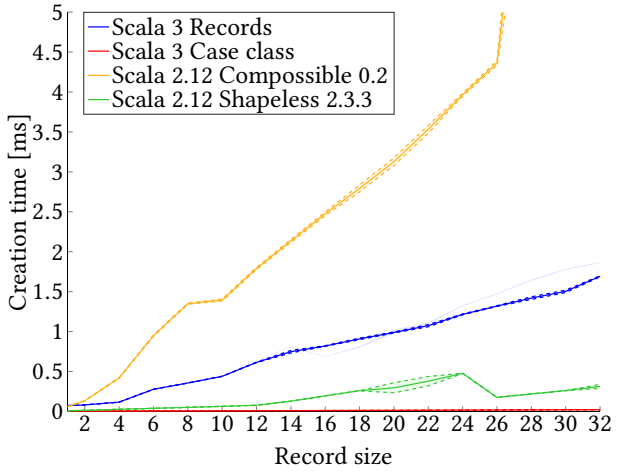

Figure 1. Record creation time against record size in number of integer fields.

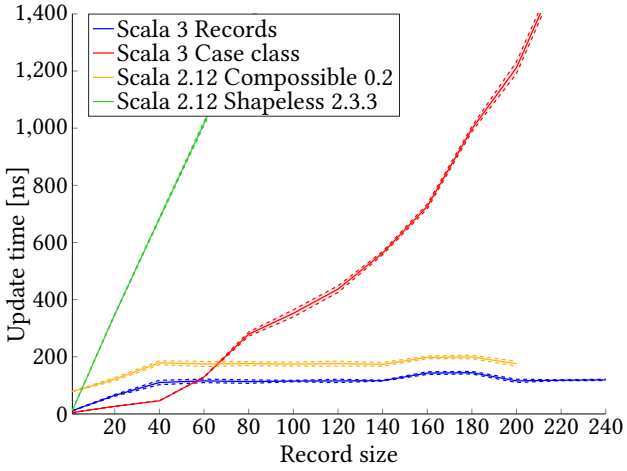

Figure 2. Record update time against record size in number of integer fields. After 200 fields Compossible exceeded the JVM code size limit.

Update Time against Record Size. A record of increasing size is created before the measurement and then the field with highest index is updated by increasing its value by one

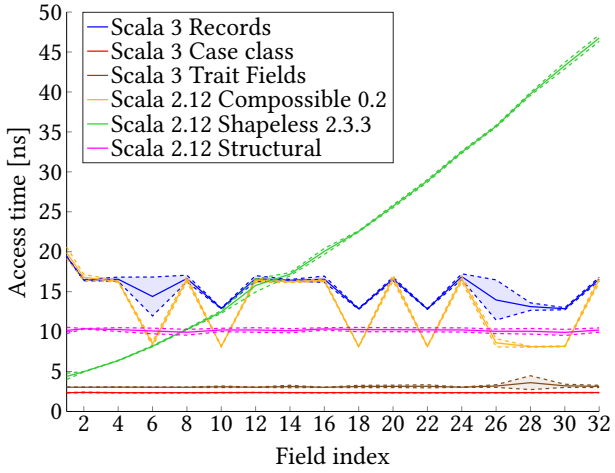

Figure 3. Record access time against field index on a record with 32 integer fields.

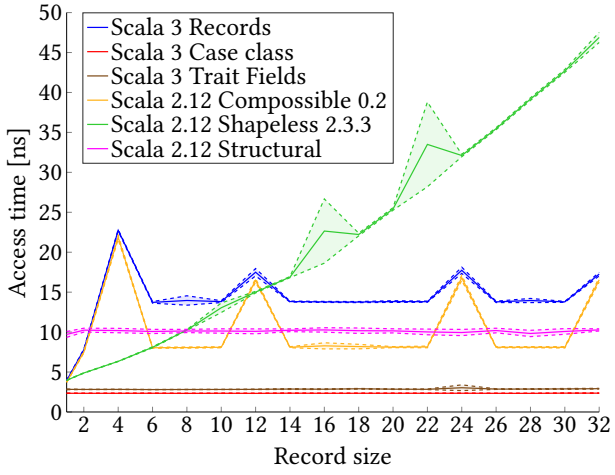

Figure 4. Record access time against record size in number of integer fields. For each record size, the field with highest index was accessed.

in an immutable update operation. For ordered records (only Shapeless) the last index corresponds to the last element in the linked list, whereas for the unordered approaches the index merely identifies the field name. The results are presented in Figure 2.

Shapeless shows a linear curve as the last index is the worst case for each record size, and the complete HList has to be copied in every update. This is also the case for the case classes, which has to be copied regardless of updated field. Our approach and Compossible on the other hand show rather stable performance across record size, as Scala's immutable hash maps have effectively constant time complexity for adding key values [Odersky and Miller 2017]. For big records with more than 60 fields this characteristic turns out to be a huge benefit.

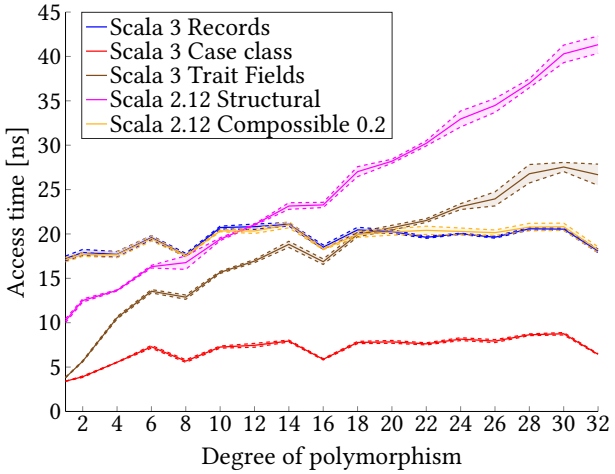

Figure 5. Record access time against degree of polymorphism on an array of different records with 32 integer fields.

Access Time against Field Index and Record Size. In Figure 3 a record with 32 integer fields `f1`, `f2`, ..., `f32` is created and used during all measurements, and then the execution time is measured for accessing field `f1` up to `f32`. In Figure 4 a record of increasing size is created before measurement and then the access time is measured for the field with the highest index. Again, the index corresponds to the field's position in the record type for ordered records whereas the index merely identifies the field's name for unordered ones.

It is somewhat surprising that the cached reflection of Scala 2's structural typing in many cases turns out to be the fastest of the structurally-typed approaches. On the other hand, in this benchmark the call site is monomorphic and so reflection is only carried out once per JVM fork and then the cached method handle gives an immediate match for every subsequent call.

In contrast to all the other approaches, Shapeless shows a clear linear access time in field index as the whole list has to be traversed to find the accessed field. This trend is also clear when measuring access time against record size, where the last index is accessed for each size. In practice though, this approach is actually the fastest for the first 6 fields (except for the case class baseline) and on par with the hash maps for the first 12 fields. The access time for our approach and Compossible fluctuates depending on the accessed field but with no noticeable increasing trend with record size, supporting the claim of effectively constant time complexity for hash look-up [Odersky and Miller 2017].

Access Time against Degree of Polymorphism. The degree of polymorphism at a call site is defined as the total number of different runtime record types that are represented among the receivers of the call. The general benchmarking technique is described by Dubochet and Odersky [2009], and is here implemented as follows: an array of 32

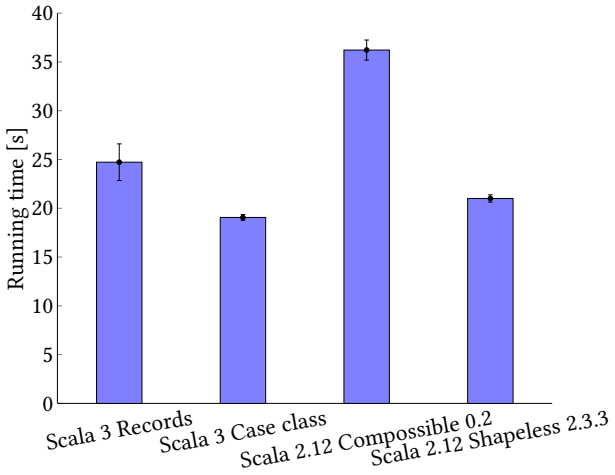

Figure 6. Mean running time for 10 executions of complete case study: parsing, reading and updating records. Plotted with 95 % confidence intervals, assuming normally distributed measurements.

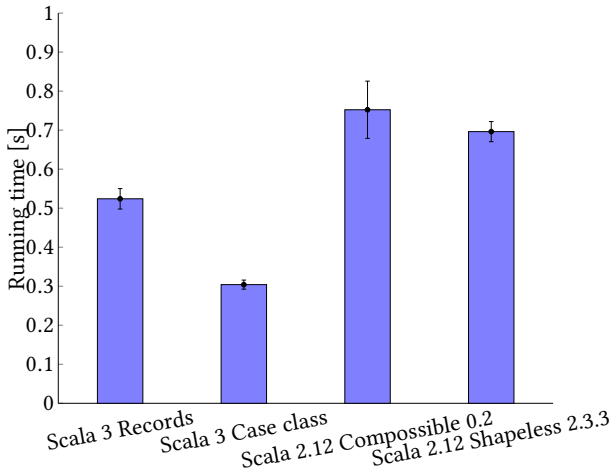

Figure 7. Running time for 10 executions of part of the case study: only reading and updating records. Plotted with 95 % confidence intervals, assuming normally distributed measurements.

records with different record types is created, but where all records have size 32 and a single field (`g: Int`) in common. To make a measurement at a call site with polymorphism degree d, the benchmark cycles over the first d records in this array and in each iteration the field g is accessed on a record with a different type from the preceding iteration. It should be noted that due to this cycling, each measurement also includes a small overhead of incrementing the index modulo d and making an array access operation. The results are presented in Figure 5.

Scala 2's structural member access shows a clear linear trend in the degree of polymorphism, confirming the results of Dubochet and Odersky [2009]. It is also worth noting that the JVM struggles with making interface calls efficient as the degree of polymorphism increases; the Scala 3 Trait Fields approach is still faster than Scala 2 Structural, but shows the same linear dependency on degree of polymorphism. This should be contrasted with the hash-map based approaches which are stable across degrees of polymorphism. After type-checking, the hash-map-based field lookup is independent of the type of the accessed record, and so the internal record type at the Scala language level can be erased during compilation to a single `Record` base class. The resulting JVM bytecode in fact contains a *monomorphic* call to the hash lookup method and the performance is not affected by the polymorphism at the Scala language level. For polymorphism degrees higher than 12 the hash-map-based approaches are faster than Scala 2 Structural, and for polymorphism degrees higher than 20 they are also faster than Scala 3 Trait Fields. Shapeless was not included as the benchmark requires the records to support permutation and width subtyping.

5.2 Case Study

Although micro benchmarks are good for pinpointing differences in performance between the different implementation strategies, it is not obvious how these differences affect the performance of a real program performing more meaningful work. To shed some light on this we also provide benchmarks of a more realistic case study. The benchmarked program parses 392440 JSON-encoded commit events from the GitHub API[11] into typed records. This list is then aggregated into a collection of user statistics keeping a count of the number of commits by each user, grouped by weekday. Thus, the program consists of a good mix of all the previously benchmarked operations: creating a nested structure of records during parsing, reading each commit event to find out its author and timestamp, and updating a user stats record for each commit event.

Each parsed commit event is stored into the following nested record structure, corresponding to the structure of the source JSON data:

```
{ ...7 fields, commit: { ... 6 fields, author: {
  name: String, email: String, date: java.time.Instant }}}
```

For each commit event, the fields `commit.author.email` and `commit.author.date` are read to get the commit author and date. These fields are last in the record structure which results in a worst-case scenario for Shapeless. For each user, the aggregated statistics are stored in a record of the following structure

```
Record{email: String, mon: Int, tue: Int, ... , sun: Int}
```

and placed in a hash map using the email as key.

[11] See https://developer.github.com/v3/repos/commits/

Create, Read, and Update. In a first run of the case study, the total execution time from parsing the JSON data to collecting the complete user stats was measured. Before the benchmark begins the complete JSON file has been read into memory in the form of a list of strings to prevent disk I/O from affecting the measured running time. The results are shown in Figure 6.

Shapeless is the fastest of the record libraries and performs almost as well as nominally typed classes. Our approach has a significantly lower running time than Compossible, although they both use a hash map as underlying data structure. Overall, the results agree well with the micro benchmark for record creation (Fig. 1) which suggests that creating records from the parsed JSON data is the bottleneck of the application. Judging by the confidence intervals, our approach is between 18 % and 42 % slower than nominally-typed classes.

Read and Update. To see how much of the execution time is spent on JSON parsing and record creation compared to reading and updating, the case study was run again, but with all parsing already done before the measurement begins. Thus, the benchmark starts with a complete list of parsed commit event records in memory and only the part of the case study that reads this list and updates the user stats records is taken into account. The result is shown in Figure 7.

Comparing the running time of this part of the case study with the complete case study in Figure 6, it is evident that more than 95 % of the execution time is spent on parsing and creating records. With most of the operations consisting of reading and updating records, a new trend emerges; Case classes still have the best overall performance, but now our approach is the fastest of the record libraries. The read fields `commit`, `commit.user`, `commit.user.email` and `commit.user.data` have index 8, 7, 2 and 3 respectively, which suggests that Shapeless should be faster than our approach and Compossible (Figure 3). On the other hand, the updated fields on the user statistics record have indices 2 to 8, which is within the interval where Shapeless is expected to perform worse than our approach but better than Compossible (Figure 2).

6 Discussion and Related Work

In this section we discuss related work and how alternative approaches to records may provide more efficient structural field access, but at the expense of violating some of our design goals.

For nominally typed records (F#, Haskell) or structurally typed monomorphic records (Standard ML, OCaml objects) it is trivial to compile field selection to a constant-time operation by storing the record values in an array and mapping each label to a statically-known offset. Record polymorphism, however, typically introduces abstract record types for which only a subset of the fields a record value might contain are present in the type. Consequently, the index of each field in the underlying value's data structure is not trivially known at compile time.

Implicit Index Abstractions. Ohori [1995] suggests a compilation scheme that resolves this difficulty in his record calculus with kinded quantification, and Gaster and Jones [1996] independently suggested essentially the same compilation scheme in their record calculus based on qualified types. In this scheme, the record values are stored in an array and sorted according to a global ordering on the set of labels, and field access on concrete record types is translated into direct array indexing. To allow field access on abstract record types, every polymorphic function is augmented with extra index parameters during compilation that provides the correct runtime indices for the known fields. At each call-site, the record type parameter is then either instantiated to a concrete record type for which the appropriate index argument values are statically known, or instantiated to another abstract record type for which similar index values already exists in scope. Gaster and Jones [1996] showed that the scheme can be augmented to also support extensible records by inserting conditional index fix-ups after each record extension operation. It should be noted however, that none of the systems of Ohori [1995] or Gaster and Jones [1996] support record polymorphism through structural subtyping, but strictly through various forms of parametric polymorphism. Without structural subtyping a concrete record type always matches its record value exactly, and this is indeed the core assumption that makes it possible for the compiler to eventually derive the required field indices from static information. In the presence of subtyping, however, this assumption breaks down.

itables. An alternative approach that supports subtyping takes inspiration from a common implementation technique for multiple inheritance and interface calls as described by Alpern et al. [2001]. A record value `r` with static record type `S` but dynamic fields corresponding to type `R` can be compiled to an array of dynamic values together with a interface-table-like structure that maps each statically known field index in `S` to its right index in the dynamic array. Whenever `r` is passed to a reference of static supertype `T` the compiler then automatically updates the table to map from `T` to `R`. The runtime cost of field selection is a single index indirection and the amortized cost of updating the table in every cast.

However, this compilation scheme breaks down in the face of Scala's variant generics; for example, Scala's lists are covariant and so an instance of `List[S]` can at any time be upcasted to `List[T]`, in which case it is unclear where and how the compiler should insert the required table coercion for every record in the collection. One solution would be to forbid implicit upcasts, and always force upcasting to be an explicit operation (as in Whiley [Pearce and Noble 2011] and OCaml), but we argue that such semantics fits badly with the existing semantics for subtyping in Scala.

Wrapper Classes at Runtime. Whiteoak [Gil and Maman 2008] is an extension to the Java language that implements structural typing on the JVM through runtime bytecode generation. Each structural type S is compiled into an interface I_S, but classes that conform structurally to S are not declared to do so at compile time–due to separate compilation, it is not feasible to let a class declare every possible structural supertype interface it implements. Instead, whenever an instance of dynamic type C and structural static type S is used as an instance of I_S, a wrapper class $W_{S,C}$ is generated at runtime that implements the interface I_C and delegates each method call to the wrapped class C. The runtime cost of field selection is then a single interface call with wrapper class delegation, plus the amortized cost of generating the wrapper class. However, bytecode generation introduces a runtime dependency on a bytecode generation framework, and furthermore requires class-loader access [Dubochet and Odersky 2009], thus violating design goal 4.

Reflection with Inline Caching. Scala 2 uses another compilation scheme to achieve structural typing on the JVM without runtime bytecode generation [Dubochet and Odersky 2009]. Instead of representing structural types as interfaces and generating wrapper classes at runtime, structural field access is simply carried out using reflection. To improve runtime performance a strategy using polymorphic inline caches is employed. The cache is implemented as a linked association list using the receiver's dynamic class as key, and so the lookup time grows linearly with the degree of polymorphism at the call-site.

Field-access performance aside, this approach has the drawback that it is unclear how to implement polymorphic record extension. Consider, for example, extending a record `r: R` with a field `f`. If the resulting record is to be constructed by instantiating a class, that class declaration has to be somehow automatically generated first. As long as all fields are known at compile time, such a class declaration can surely be generated by the compiler (as is done for anonymous classes). Due to polymorphism however, this is not always the case. If the extension should be implemented at the call-site, the compiler would have to generate a class declaration extending every possible record type `R`. Likewise, if the extension was to be implemented as a method on the record class of type `R` itself, the compiler would have to generate a class declaration for every possible set of fields the record could be extended with. It is unclear how to solve this difficulty at compile time. Granted, this could be solved by reflection and bytecode generation at runtime, but that would be a violation of design goal 4.

Interface Fields. Both Whiteoak and Scala 2 solve a more general problem than structural typing of records, since they allow retroactive structural typing of any Java class. The next scheme, suggested by Odersky [2015], treats records specially by representing each field `(f: T)` as an interface (or trait in Scala) like `field_f[T](f: T)`. A record type with fields `(f1: T)`, `(f2: T2)`, ... , `(fn: T)` can then be represented as the intersection type

```
field_f1[T1] & field_f2[T2] &| ... & field_fn[Tn]
```

and a record value is an instance of a class that implements the corresponding interfaces. The cost of field selection is a single interface call. However, to preserve separate compilation (design goal 3) the field interfaces would have to be generated at runtime rather than compile time–again violating design goal 4.

Runtime Search. As a last resort, field access can be implemented as a runtime search for the value corresponding to the accessed field. One possibility is to store the fields with both labels and values in a sorted association list. Using binary search, field access can then be done in time $O(\log n)$ in the size of the record.

Another alternative is to use Scala's immutable hash map from the standard collections library. The hash map is implemented as a hash trie with *effectively constant* execution time for selection, extension, and update. This approach is used by both scala-records and Compossible, two macro-based record libraries for Scala 2, as well as our proposed implementation of the `Record` trait.

Shapeless HList-based Records. The Shapeless library provides an example of structurally-typed records with ordered fields (violating design goal 1) for which the index of the accessed field is known at compile time. Since the values are stored as a linked list, however, access times are still linear in the size of the record.

7 Conclusion

This paper shows that a novel combination of new features of Scala 3 significantly simplifies the implementation of typesafe, extensible records; essentially, the only required compiler extension is the automatic synthesis of two new type classes. In Scala, certain type classes are already automatically synthesized, such as `Eq` and `ClassTag`. Thus, we argue that our approach fits nicely with the overall design principles of the language. We evaluate our implementation experimentally using a new benchmarking suite generator, supporting a range of compilers and record implementations for the JVM. Performance-wise, our approach is competitive with cached reflection for structural field access, as well as solutions using the deprecated, experimental macro system of Scala 2. In summary, the paper presents the first implemented design for records in Scala which enables typesafe record operations also in polymorphic contexts while only requiring minimal, idiomatic compiler support, and satisfying design goals such as separate compilation without runtime bytecode generation.

References

Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. 2001. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, USA, 108–124. https://doi.org/10.1145/504282.504291

Luca Cardelli and John C Mitchell. 1991. Operations on records. *Mathematical structures in computer science* 1, 1 (1991), 3–48.

Gilles Dubochet and Martin Odersky. 2009. Compiling structural types on the JVM: a comparison of reflective and generative techniques from Scala's perspective. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*. ACM, New York, NY, USA, 34–41. https://doi.org/10.1145/1565824.1565829

Benedict R. Gaster and Mark P. Jones. 1996. *A Polymorphic Type System for Extensible Records and Variants*. Technical Report NOTTCS-TR-96-3. Department of Computer Science, University of Nottingham.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

Joseph Gil and Itay Maman. 2008. Whiteoak: Introducing Structural Typing into Java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*. ACM, New York, NY, USA, 73–90. https://doi.org/10.1145/1449764.1449771

Vojin Jovanovic, Tobias Schlatter, Hubert Ploczniczak, et al. 2014-2018. scala-records, Labeled records for Scala based on structural refinement types and macros. https://github.com/scala-records/scala-records.

Martin Odersky. 2015. Add Records To Dotty #964. https://github.com/lampepfl/dotty/issues/964. [Online; accessed 22-May-2017].

Martin Odersky and Heather Miller. 2017. Scala collection library performance characteristics. https://docs.scala-lang.org/overviews/collections/performance-characteristics.html. [Online; accessed 29-July-2018].

Atsushi Ohori. 1995. A Polymorphic Record Calculus and Its Compilation. *ACM Transactions on Programming Languages and Systems* 17, 6 (Nov. 1995), 844–895. https://doi.org/10.1145/218570.218572

Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes As Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 341–360. https://doi.org/10.1145/1869459.1869489

David J Pearce and James Noble. 2011. Implementing a language with flow-sensitive and structural typing on the JVM. *Electronic Notes in Theoretical Computer Science* 279, 1 (2011), 47–59.

Miles Sabin et al. 2011-2018. Shapeless, Generic programming for Scala. https://github.com/milessabin/shapeless.

Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr Tuma. 2017. Unit Testing Performance in Java Projects: Are We There Yet?. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 401–412. https://doi.org/10.1145/3030207.3030226

Jan Christopher Vogt. 2015. Compossible, Extensible records and type-indexed maps. https://github.com/cvogt/compossible.