State of structural typing support in Scala 3.3.0

Julien Richard-Foy

Contributing authors: julien@richard-foy.fr;

Abstract

Scala's type system is primarily based on nominal typing. Scala 3 introduces a special type, `Selectable`, which provides an infrastructure for structural typing. Karlsson and Haller proposed improvements to `Selectable` to support extensible records. In this paper, we review several Scala 3 projects that involve structural typing. We find that their implementation or usability would benefit from the extensible records proposal from Karlsson and Haller. We investigate the remaining common challenges when working with structural types. In particular, we identify that a dedicated syntax for both record types and record literals would be the most beneficial addition to the language.

Keywords: structural typing

1 Introduction

Nominal typing is very practical for most cases but, for some use cases, such as data manipulation involving intermediate transformations, it is not convenient. Indeed, defining a data type for every intermediate result, even if it is used only once, is cumbersome. On the other hand, structural typing is well suited for such cases as it allows developers to manipulate the structure of data on-the-fly without having to declare the type of every intermediate transformation.

A common workaround in Scala is to use a tuple. For instance, a method performing a Euclidean division would typically be defined with a return type of `(Int, Int)`, modeling a pair containing both the quotient and the remainder. This solution has the benefit of being concise. On the other hand, it is not obvious to the users of that method which element of the pair is the quotient and which one is the remainder. It is easy to confuse them. A solution to that drawback would be to define a bespoke data type for the result of the Euclidean division:

```
case class EuclideanDivisionResult(quotient: Int, remainder: Int)
```

Thus, it would be obvious to the users which element is the quotient and which one is the remainder. However, in practice, Scala developers do not do follow this approach because it is too verbose compared to returning a tuple.

Structural typing would bring the best of both worlds: implementers would not need to define a data type such as `EuclideanDivisionResult`, and yet users would get a value with proper fields `quotient` and `remainder` preventing the risk of confusion.

The goal of this paper is to identify the issues that prevent Scala developers from using structural typing, and to propose solutions to them.

In the next section, I describe the current support of structural typing in Scala 3 based on the marker type `Selectable`, and I explain why it does not provide a satisfactory solution to the above example. Additionally, I present the language changes proposed by Karlsson and Haller to support extensible records[1], and I show how it improves the situation.

In their paper, Karlsson and Haller evaluated the performance of their records implementation. They showed that it is competitive with cached reflection for structural field access. However, they did not validate that their design provides a satisfactory solution to the use cases that involve structural types. In section 3, I review several existing libraries that involve structural types (currently based on `Selectable`), and I assess how much their implementation and usage would be simplified by the extensible records proposal.

In section 4, I summarize the pros and cons of the extensible records proposal, and I investigate the remaining common challenges regarding both the implementation and usage of the reviewed libraries. My analysis suggests that the most valuable addition to the extensible records proposal would be a dedicated syntax for record types and record literals.

The last section concludes.

In summary, the contributions of this work are the following:

- I evaluated the impact of the extensible records proposal on the implementation and usage of several real world libraries,
- I rebased the original implementation of the extensible records on the development branch of the Scala 3.3.x series,
- I proposed a new idea to improve further the support of structural typing in Scala 3: a dedicated syntax for record types and record literals.

2 Support of Structural Typing in Scala 3

In this section, I describe the current support of structural typing in Scala 3, and I present the extensible records proposal from Karlsson and Haller.

2.1 Status Quo: the Marker Type `Selectable`

Scala 3 comes with a special type, `Selectable`, which provides a minimal infrastructure for structural typing.

As an example, the result of the Euclidean division can be modeled by the type `Record { val quotient: Int; val remainder: Int }`, provided that type `Record` is defined as a subtype of `Selectable` that declares a method named `selectDynamic`:

```
trait Record extends Selectable:
  def selectDynamic(label: String): Any
```

Assuming we would have a value `result` of type `Record { val quotient: Int; val remainder: Int }`, the compiler would type the expression `result.quotient` as `Int`, and the expression `result.remainder` as `Int`. On the other hand, the compiler would reject any attempt to select any other field, such as `result.fractional`, for instance.

The way `Selectable` works is that the compiler only allows you to select the fields that were declared in the type refinement (between the braces following `Record`). It rewrites fields' selections to calls to the method `selectDynamic`, which takes the name of the selected field as a parameter of type `String`, and returns the value of that field.

For end-users, `Selectable` only is half satisfactory. On the one hand, once they get the result of the Euclidean division, they can access the quotient and remainder in a concise and type-safe way. On the other hand, the result type of the method is verbose, as illustrated by the following snippet that compares a tuple type expression, a case class reference, and a structural type expression:

```
(Int, Int)
EuclideanDivisionResult
Record { val quotient: Int; val remainder: Int }
```

As you can see, the structural type expression is the most verbose of the three.

For the implementers of the method that performs the Euclidean division, `Selectable` is not satisfactory at all because there is no simple and type-safe way to create a structurally-typed value. The infrastructure in the compiler is only responsible for rewriting fields' selections into calls to `selectDynamic`, but that method still needs to be implemented.

We could define `Record` as a class that internally stores fields' values in a `Map` and implements `selectDynamic` by performing a look-up in that `Map`:

```
case class Record(fields: Map[String, Any]) extends Selectable:
  def selectDynamic(label: String): Any = fields(label)
```

With this `Record` data type defined, the implementers of the method that performs the Euclidean division could return a structurally-typed value as follows:

```
Record(Map("quotient" -> quotient, "remainder" -> remainder))
  .asInstanceOf[Record { val quotient: Int; val remainder: Int }]
```

But that would still be much more verbose than returning a tuple or an instance of the class `EuclideanDivisionResult`. Furthermore, this is error-prone since any mistake in the type passed to the `asInstanceOf` call would not be caught by the compiler (e.g., a mismatch between the `Map` content and the declared record type).

Another important limitation in the current support of structural types that is not illustrated by my example is that there is no way to concatenate two structural types (for example to add new fields to a record). The type intersection operation does almost that, but it can not detect clashes (such as extending a record with a field of the same name but with a different type). The limitations of structural types in Scala 3 have been discussed in depth by Karlsson and Haller, and motivated their proposal for extensible records, which I describe in the next subsection.

2.2 Extensible Records Proposal

As described in the previous section, `Selectable` alone does not provide satisfactory support of structural typing in Scala 3. Karlsson and Haller showed that the key missing piece is the ability to *extend* structurally-typed records in a type-safe way[1]. Indeed, type-safe extension of records also provides type-safe *creation* of records since any record can be constructed by extending the empty record with additional fields.

Here is how one can create a record containing a quotient and a remainder with their proposal:

```
Record() + ("quotient" ->> quotient) + ("remainder" ->> remainder)
```

Or, with the specific constructor for records with two fields:

```
Record("quotient" ->> quotient, "remainder" ->> remainder)
```

The type of both expressions is inferred to `Record { val quotient: Int; val remainder: Int }`.

Without going into details about their proposal, the following points are worth noting:

- to ensure soundness of record extension, a record can only be extended by one field at a time (via the operator `+`),
- most of the proposal is implemented as a library but some typing rules have been changed in the compiler to simplify the type expression `Record { val a: A } & Record { val b: B }` into `Record { val a A; val b: B }`,
- no special syntax is introduced.

3 Evaluation of Extensible Records on Real World Libraries

In this section, I look at three libraries that use structural typing, and I evaluate how much the extensible records proposal would simplify them.

I consider the following axes of simplification:

1. implementation: do extensible records make those libraries easier to implement and maintain? For instance, by subsuming some parts of their implementation.
2. usage: is the user-facing API easier to navigate through? Is it easier to use? For instance, by having clearer type signatures, or a more concise syntax.

I selected the following libraries:

- Chimney, a library for transforming data structures by adding, removing, or renaming fields,

- Tyqu, a library for constructing SQL queries in a composable and type-safe way,
- Iskra, a library that adds more precise types to Spark data frames.

I rebased the implementation of extensible records (published in 2018) on top of Scala 3.3.0, re-implemented a small subset of Chimney, Tyqu, and Iskra as self-contained tests, and looked at possible simplification with records[1].

In the following subsections, I will give an overview of each library, I will explain how they are implemented, and I will assess how much they would be simplified by extensible records.

All the code examples assume following class definitions:

```scala
case class UserV1(name: String)
case class UserV2(name: String, age: Option[Int])
```

3.1 Chimney

Chimney transforms data structures by adding, removing, or renaming fields. For instance, here is a snippet that converts a value of type `UserV1` into a value of type `UserV2`:

```scala
val userV1 = UserV1("Martin")
val userV2 =
  userV1
    .into[UserV2]
    .withFieldComputed(_.age, _ => None)
    .transform
```

We start with the value `userV1`, of type `UserV1`, and we call the Chimney extension method `into` on it, with the type of the target class `UserV2`.

Chimney tries to create a mapping from `UserV1` to `UserV2` by looking at the fields in both classes that have the same name and type. In this example, the field `name` of type `String` is present in both classes, but the field `age` is present only in the target class. Chimney creates a partial mapping (handling only the field `name`), which we need to complete to cover all the fields of the target class. We call `withFieldComputed(_.age, _ => None)` to tell Chimney to map the target field `age` to the value `None`.

Finally, when the mapping covers all the fields of the target class, we can call the method `transform` to perform the conversion.

The current implementation of Chimney relies on macros that analyze the structure of the source and target types to create the mapping between them. The operation `withFieldComputed` is also implemented via a `transparent inline def` that analyzes the first argument (`_.age` in the example above) to extract the name of the field.

[1] The result is available at https://github.com/lampepfl/dotty/compare/main...julienrf:dotty:dotty-records-final.

On the usage side, the Chimney API is rather easy to navigate through and IDEs provide helpful suggestions. For instance, it is impossible to make typos on field names (e.g., `agee` instead of `age`) or assigning a value of an incompatible type to a field.

Furthermore, the authors of the library made a great job at providing sensible error messages. For instance, if we omit to provide the mapping for the `age` field:

```
Chimney can't derive transformation from UserV1 to UserV2

UserV2
  age: Option[Int] - no accessor named age in source type UserV1

Consult https://chimney.readthedocs.io for usage examples.
```

There is a lot of overlap between what Chimney does and extensible records. Indeed, with both tools developers transform data structures by adding new fields to them.

Here is a possible sketch of how Chimney could look like if extensible records were part of the language:

```
val userV2 =
  (userV1.toRecord + ("age" ->> None))
    .into[UserV2]
```

We first convert the value `userV1` to an extensible record, which we extend with a field `age`, and then we convert the resulting record into the case class `UserV2`.

Thus, the implementation of Chimney would be only responsible for providing conversions between case classes and records. The user-facing API would be reduced to the following two extension methods:

```
extension [A](value: A)
  def into[B](using Conversion[A, B]): B =
    value.convert
extension [A](recordLike: A)
  def toRecord[B <: Record](using Conversion[A, B]): B =
    recordLike.convert
```

In addition to those methods, Chimney would still be responsible for synthesizing `given` conversions between case classes and their corresponding record type.

A drawback of the records-based approach is that typos and type-mismatch errors in added fields would be caught only at the point of calling the finalizer method `.into[UserV2]`.

3.2 Tyqu

Tyqu provides an embedded DSL to describe SQL queries from aggregation, projection, and selection operations. Here is a snippet that shows how to look up in a database the users whose name is “Martin”:

```
object UsersV1 extends Table:
  val name = Column[String]()
```

```scala
val results: Seq[Result { def firstName: String }] =
  from(UsersV1)
    .filter(user => user.name === "Martin")
    .map(user => user.name.as("firstName"))
    .run(dbConnection)
```

We start by defining the schema of the `UsersV1` table via the object of the same name. It has just one column, `name`, of type `String`.

Then, we create a query that looks up in the `UsersV1` table by calling `from(UsersV1)`, and we transform that query by calling its `filter` and `map` methods. Ultimately, we execute the query by calling the `run` method with an actual database connection.

Tyqu provides a type-safe way to reference and manipulate table columns. Both operations `filter` and `map` take as parameter a function that takes a model of the table schema and returns a SQL expression.

For example, we can refer to the column `name` by writing `user.name`, which has type `Expression[String]`. This lets us write the condition `user.name === "Martin"`, which has type `Expression[Boolean]` as expected by the `filter` operation. Tyqu will translate those expressions into proper SQL when it runs the query.

To achieve this, Tyqu synthesizes a type that models the table schema (ie, the type of the `user` parameter). It does that via a macro that introspects the content of the object `UsersV1` and finds all its fields of type `Column`. It returns the structural type `Selectable { def name: Expression[String] }`.

Another point worth noting is how projection operations can transform the structural type of the underlying schema. For instance, in the call to `map` we return `user.name.as("firstName")`, which creates the alias `firstName` for the column `name`. Subsequent transformations of the query would now have to use the identifier `firstName` to refer to the column instead of `name`.

When we run the query, Tyqu returns a result of the structural type `Result { def firstName: String }` according to the alias we created (the type `Result`, defined by Tyqu, is also a subtype of `Selectable`).

There is some overlap between what Tyqu does and extensible records. The most obvious one is the `Result` type defined by Tyqu, which is literally the same as `Record`. Additionally, both Tyqu and extensible records implement a way to create values of structural types. Tyqu achieves that via the `as` operation (as in `user.name.as("firstName")`).

Here is a possible sketch of how Tyqu could look like if extensible records were part of the language:

```scala
object UsersV1Table:
  val name = Column[String]()

val results: Seq[Record { def firstName: String }] =
  from(UsersV1Table)
    .filter(user => user.name === "Martin")
```

```
  .map(user => Record("firstName" ->> user.name))
  .run(connection)
```

The differences with status quo are the following:

- executing a query now returns standard records of type `Record` instead of `Result`,
- the creation of the alias `firstName` is achieved by constructing a standard record.

The benefits of the second points are debatable, as some users may prefer using the original `as` method because it looks more similar to SQL. I will elaborate further on that point in the Discussion section.

Extensible records would not have a significant impact on the implementation of Tyqu. Indeed, the main tasks related to structural types in Tyqu consists of computing the type modeling table schemas from table object definitions, and computing the type of a query result based on the structural type of the query schema. Both tasks might be achieved with some form of generic programming support on records, but such features are not included in the extensible records proposal.

3.3 Iskra

Spark SQL allows developers to perform relational operations on data structured in columns. A drawback of Spark SQL is that its main abstraction, `DataFrame`, does not provide a precisely typed schema of the manipulated data, making it easy to do mistakes such as typos in column names, or applying an operation incompatible with the underlying column type.

The goal of Iskra is to be as source compatible as possible with Spark's `DataFrame` while providing more precise types in order to catch the type errors mentioned above at compile-time. Here is a simple example of Spark SQL program that prints the users whose name is "Martin", and then the same program with Iskra:

```
// Spark SQL
(users: DataFrame)
  .select($"name".as("user_name"))
  .where($"user_name" === "Martin")
  .show()

// Iskra
(users: DataFrame[UserV1])
  .select($.name.as("user_name"))
  .where($.user_name === "Martin")
  .show()
```

The only difference is that column selection is typechecked in Iskra: the expression `$.name` has type `Column[String]`.

Also shown in this example is the definition of column aliases, similar to the Tyqu example. The result of the `select` operation is a `DataFrame` with one column named `user_name` and whose type is `String`. As a consequence, we refer to that column in the following `where` call by using `$.user_name`.

Iskra computes a structural type modeling the schema of the manipulated `DataFrame` that is pretty similar to what Tyqu does. Iskra would marginally benefit from extensible records, like Tyqu.

Here is a possible sketch of how Iskra could look like if extensible records were part of the language:

```
(users: DataFrame[UserV1])
  .select(Record("user_name" ->> $.name))
  .where($.user_name === "Martin")
  .show()
```

Like with Tyqu, we use standard records to manipulate structurally-typed values, which provides the same debatable benefit: the resulting user-facing API would be pretty different from the original Spark SQL API. I will discuss further that point in the next section.

The other main tasks related to structural types in Iskra consists of computing the type modeling the `DataFrame` schema, and computing the type of the result of evaluating a computation. Like with Tyqu, both may be achievable with some support of generic programming on records, but such a feature is not included in the extensible records proposal.

4 Discussion

In this section, I summarize the impacts of the extensible records proposal on the three studied libraries, and I summarize the remaining common challenges when working with structural types. Then, I make new proposals to address those challenges.

4.1 Pros and Cons of Extensible Records

The following table summarizes the impact of extensible records on the studied libraries ("+" means a marginal positive impact, and "++" means a significant positive impact):

	Chimney	Tyqu	Iskra
Implementation	++	+	+
Usage	=	+	+

4.1.1 Impact on Implementation

The only case where the benefits are clear is the implementation of Chimney. Indeed, 80% of Chimney's implementation is a DSL to extend data structures. That part is subsumed by the extensible records proposal. The remaining 20% is an infrastructure to convert between case classes and records.

The impact of the extensible records on the implementation of Tyqu and Iskra is only marginal. Indeed, most of the work related to structural types performed by

Tyqu and Iskra consists of computing the types of SQL table schemas (or `DataFrame` schemas, respectively), and computing the result types of running the queries (or evaluating the computations, respectively).

For example, in Iskra, calling the `map` method with a function that returns a record of type `Record { val a: Column[A]; val b: Column[B] }` produces a `DataFrame[Schema { val a: Column[A]; val b: Column[B] }]`. ie, the schema of the resulting `DataFrame` has the same columns as the provided record. Note that it might be possible to just reuse the `Record` type itself to model a schema instead of using a bespoke `Schema` type, but the current implementation in Iskra requires the `Schema` type to have specific additional methods that are not on `Record`.

Likewise, evaluating a computation on a `DataFrame[Schema { val a: Column[A]; val b: Column[B] }]` produces a result of type `Seq[Record { val a: A; val b: B }]`. Here, the structural type of the result has the same field names as the `DataFrame` schema, but their types do not contain the `Column` type constructor anymore.

Similar examples could be constructed for Tyqu.

Unfortunately, the extensible records proposal is of no help to compute such types.

4.1.2 Impact on Usage

In all the cases, the impact of extensible records on the usability of the libraries is marginal.

Nevertheless, a general improvement is that once records become standard in Scala, then there is a unique, standard way to create and extend records that is shared by all the libraries. The user experience is more consistent across the libraries, and fewer concepts need to be learned to get started with a new library.

Other than that, the current design of the extensible records proposal still suffers from usability issues.

A minor issue is that in the proposed design, records can only be extended by using the `+` symbolic operation. This does not work well when mixed with non-infix calls. Compare for instance the following typical usage of Chimney:

```
foo
  .into[Bar]
  .withFieldConst(_.baz, 42)
  .withFieldComputed(_.quux, _.x + 1)
  .transform
```

With the following equivalent program based on my records-based adaptation of Chimney:

```
(foo
  .toRecord
    + ("baz" ->> 42)
    + ("quux" ->> foo.x + 1))
  .into[Bar]
```

The combination of infix operators and dot-notation does not play well.

This problem can easily be solved by providing a non-symbolic alias to the operation `+`, as I will show in section 4.2.1.

Another, more important issue with the extensible records design is that the syntax to define a field is not intuitive. To define a field name, developers write a `String` literal. This is inconsistent with the usual way of creating bindings in Scala, which consists of writing an unquoted identifier in a binding position. Fixing this problem would require changes to the language. Obviously, the current implementations of Tyqu and Iskra also suffer from this problem.

The last important issue is also related to syntax. As shown in the introduction, the syntax of structural types is very verbose. As a reminder, here is the type of records containing the quotient and remainder of a division:

```
Record { val quotient: Int; val remainder: Int }
```

It is obviously much more verbose than a tuple containing the same data.

This problem is important since, unlike classes, structural types can not be referred to by a simple identifier such as `EuclideanDivisionResult`. Defining a type alias would not really solve this issue because structural type expressions would still show up in signatures in the API documentation or in IDEs. Structural types are meant to be referred to by their type expression. Defining such an alias would defeat their purpose.

In the next section, I propose some ideas to fix this issue and the other aforementioned issues.

Finally, another minor usability issue with the current implementation of extensible records is that the type name `Record` clashes with `java.lang.Record`, introduced in JDK 16.

4.2 Proposals to Address the Remaining Issues

I propose three possible additions to the original extensible records proposal to address the issues described in the previous section: a non-symbolic operation to extend records, a dedicated syntax for record types and record literals, and compiler-synthesized mirrors for record types.

4.2.1 Non-Symbolic Record Extension Operation

As mentioned in the previous section, in the extensible records proposal, the only way to extend a record is to use a symbolic `+` operation, which does not play well with the dot-notation. We can fix this issue by adding a non-symbolic alias `withField` to the `+` operation on records. With this change, the initial example becomes:

```
foo
  .toRecord
  .withField("baz" ->> 42)
  .withField("quux" ->> foo.x + 1)
  .into[Bar]
```

4.2.2 Dedicated Syntax

The syntax for records should be concise, readable, and lead to as few ambiguities as possible.

The syntax for record types and record literals should be homogeneous, just like the syntax of tuple types and tuple literals, and the syntax of case class definitions and case class construction.

Tuples, records, and case classes are all means of structuring information. Ideally, the syntax of records should share similarities with the syntax of tuples and case classes.

With these principles in mind, I propose the following syntax to define a record containing the quotient and remainder of a Euclidean division:

```
(quotient: Int, remainder: Int)
```

This type would expand to the following:

```
Record { val quotient: Int; val remainder: Int }
```

Constructing a record literal of that type looks as follows:

```
(quotient = dividend / divisor, remainder = dividend % divisor)
```

Extending a record looks as follows:

```
(foo = 42) + (bar = "hello")
```

Or, with the non-symbolic operator:

```
(foo = 42).withField((bar = "hello"))
```

Just like auto-tupling allows developers to call a function taking a tuple as if it was a function taking several parameters, "auto-recording" allows developers to call a function taking a record as if it was a function taking several named parameters:

```
(foo = 42).withField(bar = "hello")
```

"auto-recording" can be a source of ambiguities, though, and more work is needed to find the right set of rules governing its application.

Another benefit of having a dedicated syntax for record literals is that it allows the extension of a record with more than one field at a time, as in:

```
(foo = 42) + (bar = "hello", baz = true)
```

This was not possible in the original extensible records proposal because of soundness issues.

Note that extending a record with a non-literal would still be forbidden:

```
val r = (bar = "hello", baz = true)
(foo = 42) + r // error
```

Last, records should also support named field patterns, as in SIP-44.

With this syntax, the code examples presented earlier can be rewritten as follows:

```
userV1
  .toRecord
  .withField(age = None)
  .into[UserV2]

foo
  .toRecord
  .withField(baz = 42)
  .withField(quux = foo.x + 1)
  .into[Bar]

val results: Seq[(firstName: String)] =
  from(UsersV1Table)
    .filter(user => user.name === "Martin")
    .map(user => (firstName = user.name))
    .run(connection)

users
  .select(user_name = $.name)
  .show()
```

I believe the simpler type signatures and literal syntax would make it easier to work with structural types.

An open question is whether the dedicated syntax should apply to records only, or to structural types in general. Even if records become standard in Scala, the underlying, more general, mechanism based on `Selectable` supports other forms of structurally-typed values. Probably, a more general syntax that would also apply to any form of structural type would be useful. A possibility could be to support a shorthand syntax for type refinements: the type `A(x: X, y: Y)` would be a shorthand for `A { val x: X; val y: Y }`.

4.2.3 Generic Programming Support for Records

The last challenge related to using structural types, faced by both Tyqu and Iskra, is to derive a structural type from another (possibly structural) type.

Let us remind one of the examples presented earlier. In Iskra, the type `DataFrame[Schema { val a: Column[A]; val b: Column[B] }]` describes a computation that produces results of type `Record { val a: A; val b: B }`. Computing the result type is mechanical: for every field `f: Column[T]` of the schema type, there is a corresponding field `f: T` in the resulting record. A generic solution to perform this type computation requires a way to iterate over the fields of a structural type.

In Scala 3, mirrors allow developers to iterate over the fields of case classes. However, this technique can not currently be used with structural types because mirrors are not synthesized for them. In the remainder of this section, I propose new rules to synthesize mirrors for structural types and assess how much that would simplify the implementation of Iskra, Tyqu, and Chimney.

Currently, the compiler synthesizes a subtype of `Mirror.Product` for every case class definition:

```
trait Product extends Mirror:
  type MirroredElemLabels <: Tuple
  type MirroredElemTypes <: Tuple
  type MirroredLabel <: String
  type MirroredMonoType
  def fromProduct(product: scala.Product): MirroredMonoType
```

For instance, here is a case class definition and its mirror type as synthesized by the compiler:

```
case class EuclideanDivisionResult(quotient: Int, remainder: Int)

// Synthesized by the compiler
trait EuclideanDivisionResultMirror extends Mirror.Product:
  type MirroredElemLabels = ("quotient", "remainder")
  type MirroredElemTypes  = (Int, Int)
  type MirroredLabel      = "EuclideanDivisionResult"
  type MirroredMonoType   = EuclideanDivisionResult
  def fromProduct(product: scala.Product): EuclideanDivisionResult =
    EuclideanDivisionResult(
      product.productElement(0).asInstanceOf[Int],
      product.productElement(1).asInstanceOf[Int]
    )
```

The type members of `EuclideanDivisionResultMirror` model the structure of the class `EuclideanDivisionResult`. The mirror also implements the method `fromProduct`, which provides a constructor for `EuclideanDivisionResult`. It is worth noting that this constructor is position-based: it iterates over the elements of the provided product and passes them as parameters to the primary constructor of the class `EuclideanDivisionResult`.

That mechanics would not work as is on structural types because position is not significant in structural types. Instead, the constructor should be name-based:

```
def fromFields(fields: Map[String, Any]): MirroredMonoType
```

The complete template type for record type mirrors would be the following:

```
object Mirror:
  trait Record extends Mirror:
    type MirroredElemLabels <: Tuple
    type MirroredElemTypes <: Tuple
    type MirroredLabel = Nothing
    type MirroredMonoType <: scala.Record
    def fromFields(fields: Map[String, Any]: MirroredMonoType
```

Then, the compiler would synthesize the following mirror type for the record type modeling the result of a Euclidean division:

```scala
trait RecordQuotientRemainderMirror extends Mirror.Record:
  type MirroredElemLabels = ("quotient", "remainder")
  type MirroredElemTypes = (Int, Int)
  type MirroredMonoType =
     Record { val quotient: Int; val remainder: Int }
  def fromFields(
    fields: Map[String, Any]
  ): Record { val quotient: Int; val remainder: Int } =
    Record()
      + ("quotient" ->> fields("quotient").asInstanceOf[Int])
      + ("remainder" ->> fields("remainder").asInstanceOf[Int])
```

Note that such mirrors could not be synthesized for `Selectable` in general, because `Selectable` also supports structural method calls via a method `applyDynamic`, not just structural field selection via `selectDynamic`.

The benefits of record mirrors would be to provide a standard, high-level, way to iterate over the fields of a record type.

Currently, it is already possible to iterate over the fields of structural types by iterating over the `Refinement` nodes of the `TypeRepr` model of the structural type, with TASTy reflection. However, such refinements may also include arbitrary refinements such as `{ type X; def f(x: Int): X }`.

Record mirrors would simplify the work of the developers by removing the need to check the structure of the refinements: the compiler would synthesize a mirror only if the refinements all have the form `val <name>: <type>`, and that none of the field types uses another field name as a type prefix (as in `{ val x: X; val y: x.Y }`).

Even though our motivating example only mentions Iskra and Tyqu, it is likely that record mirrors would also benefit to a library like Chimney to implement the conversions from records to case classes. The developers would summon both the mirror of the source record type and the target case class, check that all the fields of the case class are present in the record type and have a compatible type, and finally construct the case class instance by retrieving all the fields' values by calling `selectDynamic` on the underlying record instance.

That being said, record mirrors would also have some drawbacks. Unlike case class mirrors, which have almost no compile-time and run-time footprint, record mirror types cannot be cached by the compiler as efficiently as case class mirrors, and their usage would require the allocation of extra objects.

In conclusion, record mirrors may provide additional benefits on the implementation of libraries manipulating structural types, but more work is necessary to more precisely assess their impact.

5 Conclusion

In this paper, I reviewed the support of structural typing in Scala 3.3.0 and identified possible ways to improve it.

The first issue is that there is no standard and type-safe way to create structurally-typed values. I assessed that that issue is fully solved by the extensible records proposal from Karlsson and Haller.

My analysis showed that the next most important issue is that the syntax of structural type expressions is too verbose. Structural types, by definition, do not have a short name that can be used to reference them. Therefore, they need a concise syntax to be usable at scale. I proposed a syntax for record type expressions, and a consistent syntax for record literals.

I also investigated other minor issues and their possible solutions, such as generic programming on records via compiler-synthesized mirrors.

References

[1] Karlsson, O., Haller, P.: Extending scala with records: design, implementation, and evaluation. In: Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, SCALA@ICFP 2018, St. Louis, MO, USA, September 28, 2018, pp. 72–82. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3241653.3241661