Programmatic Structural Types

Motivation

Some usecases, such as modelling database access, are more awkward in statically typed languages than in dynamically typed languages: With dynamically typed languages, it's quite natural to model a row as a record or object, and to select entries with simple dot notation (e.g. `row.columnName`).

Achieving the same experience in statically typed language requires defining a class for every possible row arising from database manipulation (including rows arising from joins and projections) and setting up a scheme to map between a row and the class representing it.

This requires a large amount of boilerplate, which leads developers to trade the advantages of static typing for simpler schemes where colum names are represented as strings and passed to other operators (e.g. `row.select("columnName")`). This approach forgoes the advantages of static typing, and is still not as natural as the dynamically typed version.

Structural types help in situations where we would like to support simple dot notation in dynamic contexts without losing the advantages of static typing. They allow developers to use dot notation and configure how fields and methods should be resolved.

Example

Here's an example of a structural type `Person`:

```
class Record(elems: (String, Any)*) extends Selectable:
    private val fields = elems.toMap
    def selectDynamic(name: String): Any = fields(name)
```

```
  type Person = Record { val name: String; val age: Int }
```

The type `Person` adds a *refinement* to its parent type `Record` that defines the two fields `name` and `age`. We say the refinement is *structural* since `name` and `age` are not defined in the parent type. But they exist nevertheless as members of class `Person`. For instance, the following program would print "Emma is 42 years old.":

```
val person = Record("name" -> "Emma", "age" -> 42).asInstanceOf[Person]
  println(s"${person.name} is ${person.age} years old.")
```

The parent type `Record` in this example is a generic class that can represent arbitrary records in its `elems` argument. This argument is a sequence of pairs of labels of type `String` and values of type `Any`. When we create a `Person` as a `Record` we have to assert with a typecast that the record defines the right fields of the right types. `Record` itself is too weakly typed so the compiler cannot know this without help from the user. In practice, the connection between a structural type and its underlying generic representation would most likely be done by a database layer, and therefore would not be a concern of the end user.

`Record` extends the marker trait `scala.Selectable` and defines a method `selectDynamic`, which maps a field name to its value. Selecting a structural type member is done by calling this method. The `person.name` and `person.age` selections are translated by the Scala compiler to:

```
person.selectDynamic("name").asInstanceOf[String]
  person.selectDynamic("age").asInstanceOf[Int]
```

Besides `selectDynamic`, a `Selectable` class sometimes also defines a method `applyDynamic`. This can then be used to translate function calls of structural members. So, if `a` is an instance of `Selectable`, a structural call like `a.f(b, c)` would translate to

```
a.applyDynamic("f")(b, c)
```

Using Java Reflection

Structural types can also be accessed using Java reflection. Example:

```
type Closeable = { def close(): Unit }

  class FileInputStream:
```

```
    def close(): Unit


  class Channel:
    def close(): Unit
```

Here, we define a structural type `Closeable` that defines a `close` method. There are various classes that have `close` methods, we just list `FileInputStream` and `Channel` as two examples. It would be easiest if the two classes shared a common interface that factors out the `close` method. But such factorings are often not possible if different libraries are combined in one application. Yet, we can still have methods that work on all classes with a `close` method by using the `Closeable` type. For instance,

```
import scala.reflect.Selectable.reflectiveSelectable

  def autoClose(f: Closeable)(op: Closeable => Unit): Unit =
    try op(f) finally f.close()
```

The call `f.close()` has to use Java reflection to identify and call the `close` method in the receiver `f`. This needs to be enabled by an import of `reflectiveSelectable` shown above. What happens "under the hood" is then the following:

- The import makes available an implicit conversion that turns any type into a `Selectable`. `f` is wrapped in this conversion.
- The compiler then transforms the `close` call on the wrapped `f` to an `applyDynamic` call. The end result is:

  ```
  reflectiveSelectable(f).applyDynamic("close")()
  ```

- The implementation of `applyDynamic` in `reflectiveSelectable`'s result uses Java reflection to find and call a method `close` with zero parameters in the value referenced by `f` at runtime.

Structural calls like this tend to be much slower than normal method calls. The mandatory import of `reflectiveSelectable` serves as a signpost that something inefficient is going on.

Note: In Scala 2, Java reflection is the only mechanism available for structural types and it is automatically enabled without needing the `reflectiveSelectable` conversion. However, to warn against inefficient dispatch, Scala 2 requires a language

import `import scala.language.reflectiveCalls`.

Before resorting to structural calls with Java reflection one should consider alternatives. For instance, sometimes a more a modular *and* efficient architecture can be obtained using type classes.

Extensibility

New instances of `Selectable` can be defined to support means of access other than Java reflection, which would enable usages such as the database access example given at the beginning of this document.

Local Selectable Instances

Local and anonymous classes that extend `Selectable` get more refined types than other classes. Here is an example:

```
trait Vehicle extends reflect.Selectable:
  val wheels: Int

val i3 = new Vehicle: // i3: Vehicle { val range: Int }
  val wheels = 4
  val range = 240

i3.range
```

The type of `i3` in this example is `Vehicle { val range: Int }`. Hence, `i3.range` is well-formed. Since the base class `Vehicle` does not define a `range` field or method, we need structural dispatch to access the `range` field of the anonymous class that initializes `id3`. Structural dispatch is implemented by the base trait `reflect.Selectable` of `Vehicle`, which defines the necessary `selectDynamic` member.

`Vehicle` could also extend some other subclass of `scala.Selectable` that implements `selectDynamic` and `applyDynamic` differently. But if it does not extend a `Selectable` at all, the code would no longer typecheck:

```
trait Vehicle:
  val wheels: Int

val i3 = new Vehicle: // i3: Vehicle
  val wheels = 4
  val range = 240
```

```
i3.range // error: range is not a member of `Vehicle`
```

The difference is that the type of an anonymous class that does not extend `Selectable` is just formed from the parent type(s) of the class, without adding any refinements. Hence, `i3` now has just type `Vehicle` and the selection `i3.range` gives a "member not found" error.

Note that in Scala 2 all local and anonymous classes could produce values with refined types. But members defined by such refinements could be selected only with the language import `reflectiveCalls`.

Relation with `scala.Dynamic`

There are clearly some connections with `scala.Dynamic` here, since both select members programmatically. But there are also some differences.

- Fully dynamic selection is not typesafe, but structural selection is, as long as the correspondence of the structural type with the underlying value is as stated.
- `Dynamic` is just a marker trait, which gives more leeway where and how to define reflective access operations. By contrast `Selectable` is a trait which declares the access operations.
- Two access operations, `selectDynamic` and `applyDynamic` are shared between both approaches. In `Selectable`, `applyDynamic` also may also take `java.lang.Class` arguments indicating the method's formal parameter types. `Dynamic` comes with `updateDynamic`.

More details

‹ Numer... Progra... ›