Type System, Type Extraction, Serialization

Flink handles types in a unique way, containing its own type descriptors, generic type extraction, and type serialization framework. This document describes the concepts and the rationale behind them.

Type handling in Flink

Flink tries to know as much information about what types enter and leave user functions as possible. This stands in contrast to the approach to just assuming nothing and letting the programming language and serialization framework handle all types dynamically.

- To allow using POJOs and grouping/joining them by referring to field names, Flink needs the type information to make checks (for typos and type compatibility) before the job is executed.
- The more we know, the better serialization and data layout schemes the compiler/optimizer can develop. That is quite important for the memory usage paradigm in Flink (work on serialized data inside/outside the heap and make serialization very cheap).
- For the upcoming logical programs (see roadmap draft) we need this to know the “schema” of functions.
- Finally, it also spares users having to worry about serialization frameworks and having to register types at those frameworks.

Flink’s TypeInformation class

The class TypeInformation is the base class for all type descriptors. It reveals some basic properties of the type and can generate serializers and, in specializations, comparators for the types. (Note that comparators in Flink do much more than defining an order - they are basically the utility to operations that depend on keys, like grouping or joining)

Internally, Flink makes the following distinctions between types:

- **Basic types**: All Java primitives and their boxed form, plus `void`, `String`, and `Date`.
- **Primitive arrays** and `Object arrays`
- **Composite types**
  - Flink Java Tuples (part of the Flink Java API)
  - Scala case classes (including Scala tuples)
  - POJOs: classes that follow a bean-like pattern
- **Scala auxiliary types** (Option, Either, Lists, Maps, …)
- **Generic types**: This is the fallback type if a class matches none of the aforementioned cases. These types will not be serialized by Flink itself, but delegated to Kryo (default) or Avro (upon activation)

POJOs are of particular interest, because they support the creation of complex types and the use of field names in the definition of keys: `dataSet.join(another).where("name").equalTo("personName")`. They are also transparent to the runtime and can be handled very efficiently by Flink.

Rules for POJO types

Flink recognizes a data type as a POJO type (and allows “by-name” field referencing) if the following conditions are fulfilled:

- The class is public and standalone (no non-static inner class)
- The class has a public no-argument constructor
- All fields in the class (and all superclasses) are either public or or have a public getter and a setter method that follows the Java beans naming conventions for getters and setters.

Type Information in the Scala API

Scala has very elaborate concepts for runtime type information though `type manifests` and `class tags`. In general, types and methods have access to the types of their generic parameters - thus, Scala programs do not suffer from type erasure as Java programs do.

In addition, Scala allows to run custom code in the Scala Compiler through Scala Macros, which we use quite a lot in Flink. That means that some Flink code gets executed inside the Scala Compiler whenever you compile a Scala program written against Flink’s Scala API.

We use the Macros to look at the parameter types and return types of all user functions during compilation - that is the point in time when certainly all type information is perfectly available. Within the macro, we create a `TypeInformation` for the function’s return types (or parameter types) and make it part of the operation.

No Implicit Value for Evidence Parameter Error

In the case where TypeInformation could not be created, programs fail to compile with an error stating “could not find implicit value for evidence parameter of type TypeInformation”.

A frequent reason if that the code that generates the TypeInformation has not been imported. Make sure to import the entire flink.api.scala package.

```scala
import org.apache.flink.api.scala._
```

Another common cause are generic methods, which can be fixed as described in the following section.
Generic Methods

Consider the following case below:

```scala
def[T] selectFirst(input: DataSet[(T, _)]) : DataSet[T] = { input.map { v => v._1 } } val data : DataSet[(String, Long)] = ... val result = selectFirst(data)
```

For such generic methods, the data types of the function parameters and return type may not be the same for every call and are not known at the site where the method is defined. The code above will result in an error that not enough implicit evidence is available.

In such cases, the type information has to be generated at the invocation site and passed to the method. Scala offers implicit parameters for that.

The following code tells Scala to bring a type information for \( T \) into the function. The type information will then be generated at the sites where the method is invoked, rather than where the method is defined.

```scala
def[T : TypeInformation] selectFirst(input: DataSet[(T, _)]) : DataSet[T] = { input.map { v => v._1 } }
```

Type Information in the Java API

Java, in general, erases generic type information. Only for subclasses of generic classes, the subclass stores the type to which the generic type variables bind.

Flink uses reflection on the (anonymous) classes that implement the user functions to figure out the types of the generic parameters of the function. This logic also contains some simple type inference for cases where the return types of functions are dependent on input types, such as in the generic utility method below:

```java
public class AppendOne<T> extends MapFunction<T, Tuple2<T, Long>> { public Tuple2<T, Long> map(T value) { return new Tuple2<T, Long>(value, 1L); } }
```

Not in all cases can Flink figure out the data types of functions reliably in Java. Some issues remain with generic lambdas (we are trying to solve this with the Java community, see below) and with generic type variables that we cannot infer.

Type Hints in the Java API

To help cases where Flink cannot reconstruct the erased generic type information, the Java API offers so called type hints from version 0.9 on. The type hints tell the system the type of the data set produced by a function. The following gives an example:

```java
DataSet<SomeType> result = dataSet .map(new MyGenericNonInferrableFunction<Long, SomeType>()).returns(SomeType.class);
```

The \( \text{returns} \) statement specifies the produced type, in this case via a class. The hints support type definition through

- Classes, for non-parameterized types (no generics)
- Strings in the form of \( \text{returns("Tuple2<Integer, my.SomeType>"), which are parsed and converted to a TypeInformation.} \)
- A TypeInformation directly

Type extraction for Java 8 lambdas

Type extraction for Java 8 lambdas works differently than for non-lambdas, because lambdas are not associated with an implementing class that extends the function interface.

Currently, Flink tries to figure out which method implements the lambda and uses Java’s generic signatures to determine the parameter types and the return type. However, these signatures are not generated for lambdas by all compilers (as of writing this document only reliably by the Eclipse JDT compiler 4.5 from Milestone 2 onwards)

Improving Type information for Java Lambdas

One of the Flink committers (Timo Walther) has actually become active in the Eclipse JDT compiler community and in the OpenJDK community and submitted patches to the compiler to improve availability of type information available for Java 8 lambdas.

The Eclipse JDT compiler has added support for this as of version 4.5 M4. Discussion about the feature in the OpenJDK compiler is pending.