Static Type Checking of Hadoop MapReduce Programs

Jens Dörre
University of Passau
Germany

Sven Apel
University of Passau
Germany

Christian Lengauer
University of Passau
Germany

ABSTRACT

MapReduce is a programming model for the development of Web-scale programs. It is based on concepts from functional programming, namely higher-order functions, which can be strongly typed using parametric polymorphism. Yet this connection is tenuous. For example, in Hadoop, the connection between the two phases of a MapReduce computation is unsafe: there is no static type check of the generic type parameters involved. We provide a static check for Hadoop programs without asking the user to write any more code. To this end, we use strongly typed higher-order functions checked by the standard Java 5 type checker together with the Hadoop program. We also generate automatically the code needed to execute this program with a standard Hadoop implementation.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures, frameworks, inheritance, input/output*

General Terms

Design, Languages, Reliability

Keywords

MapReduce, Hadoop, generic types, static type checking

1. INTRODUCTION

MapReduce has proved to be a practical programming model for cluster computing. Its first promoter, Google, uses it, for example, for building its Web search index [2]. There are many other uses. For example, it can be used for image processing in astronomy [11] and for the analysis of spatial data from sensor networks [5].

MapReduce is a flexible model that allows to combine functions at run time to a *MapReduce program*. One may even create a *workflow* consisting of multiple *MapReduce jobs*, each implemented as a different MapReduce program. For example, Google uses a sequence of over twenty MapReduce jobs to build its search index.

Still, this flexibility comes at a price. In many MapReduce implementations, MapReduce programs are not type checked at compile time. Imagine a programming error in the midst of all the programs used in the Google workflow: you will not notice it until you can schedule the workflow for execution on a cluster, the computations are halfway done, and the MapReduce job affected has started to run. Then you hurry to fix the error, because the second half of the computations need to complete. So, we face the problem of how to detect errors in MapReduce programs as early as possible.

We restrict ourselves to the subproblem of intra-job type safety: we consider errors inside a single MapReduce job, but not between multiple jobs. Nevertheless, a solution to this problem will prove more useful for larger workflows than for single MapReduce jobs: we can use it to check every MapReduce program in a workflow prior to running any one. Then we can fix the error(s) found, validate that no other bugs have been introduced, and prepare to run the complete workflow.

In Section 2, we provide an introduction to MapReduce, how it executes jobs in phases, and to one of its many implementations: Hadoop. We also introduce our running example: a program for solving a shortest path problem. Then, in Section 3, we state the problem of type checking Hadoop programs. In Section 4, we present a solution: a static type checker for Hadoop. In Section 5, we go into more technical detail of type checking, the chaining combinators we use in it, and how we generate legacy Hadoop programs. In Section 6, we present an extension of our type checker for subtyping of function types. Section 7 contains a brief overview of related work. Finally, in Section 8, we draw some conclusions and give a prospect of further work.

2. MAPREDUCE

Technically, a MapReduce system is a framework for processing data in chunks. A *framework* is a collection of functions that can be called by user code, and that may also call user-defined functions, which are then named *callback functions*. A MapReduce system has the following properties:

1. The format of the input data can be chosen freely.
2. The output data consist of pairs of arbitrary keys and values.
3. Processing happens in two consecutive phases, using two user-defined functions: `mapper` and `reducer`.
4. The `mapper` function creates intermediate results (pairs of keys and values of arbitrary type each) from each input chunk.
5. The `reducer` function is applied in unison to all intermediate results with the same key, and produces arbitrarily many final results.

Additionally, there is a main function in a MapReduce frame-

MapReduce'11, June 8, 2011, San Jose, California, USA.

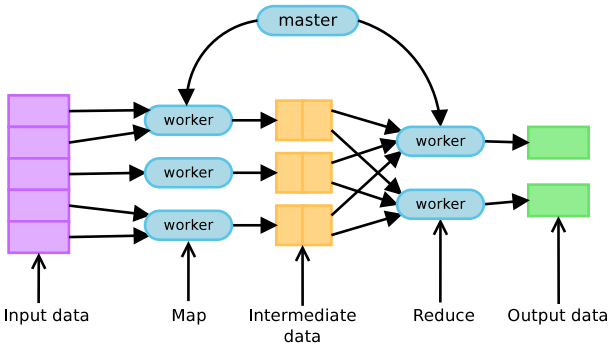

Figure 1: Schematic overview of MapReduce processing.

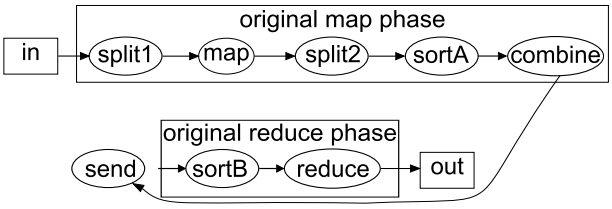

Figure 2: Phases and subphases in MapReduce processing.

work that has to be called by the user to specify the MapReduce program to be run as a job: which `mapper` and `reducer` function to execute and which data to process. MapReduce employs multiple used-defined functions operating on arbitrary data types. For a MapReduce program to be correct, the types of the data items and of the functions need to be compatible with each other.

This compatibility can be expressed as type correctness if we have a means of typing *all* the data and functions involved. It turns out that we can order them by the complexity of typing: ordinary data items have simple types, callback functions have first-order function types (they consume and produce data items), and the main function is higher-order, as it takes callback functions as parameters. As a consequence, we need generic types (to take advantage of *parametric polymorphism* [9]) if we want to type check a MapReduce program.

2.1 Phases

MapReduce owes its name to the two main phases into which its execution can be divided: the Map phase (in which mainly the `mapper` function executes) and the Reduce phase (executing the `reducer` function). This is illustrated in Figure 1. Rectangles represent chunks of distributed (input, intermediate, and output) data, and ovals labelled "worker" represent nodes executing user-defined functions. In each phase, processing can happen in parallel. In contemporary applications, the computation is typically distributed over a cluster of hundreds to thousands of worker nodes, controlled by a single master node. In this distributed setting, large sets of data have to be *serialized* (converted to a representation suitable for transport over a network), communicated over the network, and deserialized during a MapReduce computation. As this may incur huge communication costs, MapReduce also contains a feature that optimizes locality in the Map phase: the `mapper` function, operating on a particular chunk of input data, is typically computed *on the same node* on which the chunk is stored.

There is an extension to MapReduce that allows to reduce communication even further: the `combiner` function. A `combiner` function is a third user-defined function that processes the intermediate data after the `mapper` function has produced it, and that is executed on the same node using the data in memory [2]. This leads us to subdivide the two phases of a MapReduce computation further, as illustrated in Figure 2. The outer rectangles represent the two phases and the inner ovals subphases in the execution of a distributed MapReduce program with `combiner` function.

In the Map phase, the input is split up into chunks (split1), which are fed to the `mapper` function (map). The intermediate results produced are then split up again (split2; this is needed to create the partitioning desired for the final output) and sorted (sortA) to guarantee a deterministic order of processing. All intermediate results stored on one node can then be pre-reduced (a.k.a. combined by the `combiner` function, combine), which may reduce the size of the data to be sent over the network.

In the Reduce phase, the prereduced intermediate results received by a node have to be sorted (or merely merged, sortB) to establish the desired order, and can then be processed by the `reducer` function (reduce), which in turn produces a partition of the final output.

To make use of this optimization, the programmer has to write a third function besides a `mapper` and a `reducer` function. Since it has been noted that the `reducer` function can often be reused as the `combiner` function in the same MapReduce program [2], additional code might not be necessary.

2.2 Hadoop

There are many implementations of MapReduce in various programming languages. We look at Apache's *Hadoop* MapReduce,[1] a widely used framework for MapReduce, whose Java source code is freely available. For serialization, it uses an interface (called `Writable`) to which all data types must adhere.

In Hadoop, there is no explicit representation for MapReduce programs; one can only create jobs (class `Job`) to be submitted to a Hadoop installation for execution. We call the sequence of Java statements that create a job the *main program*. Jobs are represented as a combination of byte code (the user-defined functions) and XML data or simple strings (the glue "code"). Thus, user-defined functions are written in Java, while the glue code is either written in XML, given as string parameters, or written in Java using a thin layer that encapsulates the XML data in a Java data type (a class called `Configuration`, which stores a job's data).

2.3 An Example

A running example shall serve to illustrate where type errors can occur in Hadoop programs, and how to correct them. This example is not construed; we wrote the program for another purpose and were surprised by the run-time type error, which motivated us to address this research question in the first place.

We want to use Hadoop to compute the shortest paths in a large undirected graph. Our solution makes use of two different MapReduce programs: the first program forms new paths by concatenating any two shortest paths of which the end of one is the start of the other, while the second program removes paths that are not shortest. Jobs from the two programs are created and executed in alternation until all shortest paths have been computed.

We discuss only the second program. Input are paths that may or may not be shortest. Only the shortest paths are output. Paths are represented as pairs of keys (the source node) and values (the list of nodes on the path, including the source and target node).

Let us specify the two user-defined functions.

- The `mapper` function computes, for each path in its input, a

[1] `http://hadoop.apache.org/mapreduce/`; we use version 0.20.2.

key/value pair whose key is the pair of a source and a target node and whose value is the list of nodes on the path.

- The `reducer` function processes each group of intermediate values with the same key, filters out the paths which do not have the minimal length, and produces, for each remaining input item, a pair of the source node as the key and the list of nodes on the path as the value.

This program works, but it runs unnecessarily slowly, since all the intermediate results computed by the other program have to be communicated over the network to the node processing the matching group of intermediate values. Fortunately, MapReduce provides the `combiner` functions, which we have described above, to cope with this problem. We modify the (second) main program by introducing a `combiner` function to accelerate it. The `mapper` and `reducer` function remain unchanged. We note that we need not write code to implement a new `combiner` function: we can just reuse the code of the existing `reducer` function.

```
1  static int runMinLength(Job job, Path iPath,
2                                   Path oPath) throws ...
3  {
4      FileInputFormat.addInputPath(job, iPath);
5      FileOutputFormat.setOutputPath(job, oPath);

7      job.setMapperClass(MinMapperForText.class);
8      job.setCombinerClass(MinLengthReducer.class);
9      job.setReducerClass(MinLengthReducer.class);

11     job.setOutputKeyClass(Pair.class);
12     job.setOutputValueClass(Text.class);

14     return (job.waitForCompletion(true) ? 0 : 1);
15 }
```

Figure 3: The main program with `combiner`.

The Hadoop code for the main program is shown in Figure 3. Function `runMinLength` takes a rudimentary job description, and input and output paths in the distributed filesystem of Hadoop (HDFS). First it sets these locations in the job description. Then the MapReduce job is created, consisting of reflected class values for a `mapper` (`MinMapperForText`), an additional `combiner` (`MinLengthReducer`, highlighted in bold) and a `reducer` function (`MinLengthReducer`).[2] Hadoop needs additional "type declarations" for the keys (`Pair`) and values (`Text`, our representation of paths in the graph) of intermediate data which are communicated over the network. These statements are not regular Java type declarations, but function calls using reflection to convert types to a reflective representation as values. After these types have been supplied, the job is executed in the last line, and its exit status is returned.

3. THE PROBLEM

The optimized MapReduce program compiles, but it raises a run-time exception, shown in Figure 4. The first line states the type of the exception and the error message. The lines below contain the Java stack trace, listing the procedures executed, from the last callee to the first caller.

The exception is an `IOException`, indicating that *something* went wrong during I/O. Its message text mentions that the class of keys mismatched. The highlighted parts of the stack trace show that the error occurred in the code of class `MinLengthReducer`. The

[2]Since functions have to be modelled as classes in Java to become first-class entities with associated types, we will not distinguish any longer between these two concepts.

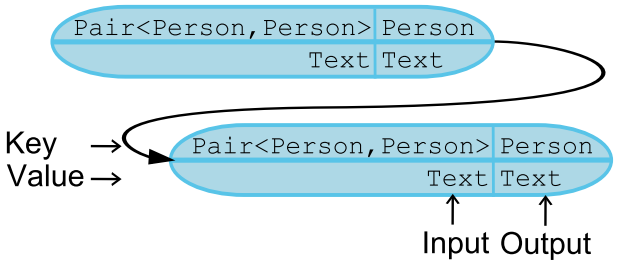

Figure 5: The mismatch between our `combiner` (at the top) and `reducer` function (at the bottom).

exception message states that we made a type error: we confused the types `Person` and `Pair`.

To locate the error, we look at class `MinLengthReducer`. It is used as both `combiner` and `reducer` function. The `combiner` function feeds into the `reducer` function, so its output must match the latter's input (cf. Figure 5). Consider the type parameters of class `MinLengthReducer`. Important are the types of the keys, since they are stated in the exception message: the `combiner` function produces keys of type `Person`, while the `reducer` function expects keys of type `Pair<Person,Person>`. This raises the exception.

To correct the error, we need an extra `combiner` class distinct from the `reducer` class. It must have distinct instantiations of the generic type parameters and a distinct implementation, which produces output in the format of the intermediate results instead of the final results. We name the class `MinLengthCombiner`. Its four type parameters read: `Pair<Person,Person>`, `Text`, **`Pair<Person,Person>`**, and `Text`.

More generally speaking, Java is statically typed, so one might expect a Java compiler to detect this error at compile time. But, as an object-oriented language, Java has subtyping and a universal supertype (or top type), class `Object`.[3] Values may be cast up (safely) and down (unsafely). So, the static type system cannot always detect whether a (down)cast must fail. To be type-safe, Java has to complement its static type system with a dynamic counterpart that detects unsafe downcasts at run time.

Generic types, providing parametric polymorphism, have only recently made it into Java, as so-called Generics. Many APIs and programmers do not use them, but resort to using *subtype polymorphism* instead [9]. This often defers the type check to run time, and leads to exactly the kind of run-time type error encountered here, which is normally associated with dynamic languages.

Hadoop does not require the programmer to use generic types for the user-defined functions written and assembled to form a job. As a consequence, a MapReduce program consisting of more than one user-defined function—for example, any program containing both `mapper` and `reducer` functions—will not be type checked at compile time! This is the reason why the Java compiler did not warn us about the type error just described. Having been made aware by the run-time system, we isolated the error by inspection and corrected *this* program. In the following section, we generalize our approach to detecting similar errors automatically in Hadoop MapReduce programs at compile time.

4. A SOLUTION

Static checkers have proved to be valuable tools for writing correct programs. Compilers for functional languages are particularly

[3]Technically, this is only true if we disregard primitive types.

```
java.io.IOException: wrong key class: class socialgraph.Person is not class socialgraph.Pair
        at org.apache.hadoop.mapred.IFile$Writer.append(IFile.java:164)
        at org.apache.hadoop.mapred.Task$CombineOutputCollector.collect(Task.java:880)
        at org.apache.hadoop.mapred.Task$NewCombinerRunner$OutputConverter.write(Task.java:1197)
        at org.apache.hadoop.mapreduce.TaskInputOutputContext.write(TaskInputOutputContext.java:80)
        at socialgraph.MinLengthReducer.workOnInput(MinLengthReducer.java:56)
        at socialgraph.MinLengthReducer.reduce(MinLengthReducer.java:36)
        at socialgraph.MinLengthReducer.reduce(MinLengthReducer.java:1)
        at org.apache.hadoop.mapreduce.Reducer.run(Reducer.java:176)
        at org.apache.hadoop.mapred.Task$NewCombinerRunner.combine(Task.java:1217)
        at org.apache.hadoop.mapred.MapTask$MapOutputBuffer.sortAndSpill(MapTask.java:1227)
        at org.apache.hadoop.mapred.MapTask$MapOutputBuffer.flush(MapTask.java:1091)
        at org.apache.hadoop.mapred.MapTask$NewOutputCollector.close(MapTask.java:512)
        at org.apache.hadoop.mapred.MapTask.runNewMapper(MapTask.java:585)
        at org.apache.hadoop.mapred.MapTask.run(MapTask.java:305)
        at org.apache.hadoop.mapred.LocalJobRunner$Job.run(LocalJobRunner.java:176)
```

Figure 4: Java run-time type error of the example of Section 2.3.

successful in applying static checks, but modern object-oriented languages like C# and Java also have good support for compile-time checks. We can detect errors like the one just described by making the necessary type information available to the compiler.

Our solution works with any compiler for Java 5 or higher. Since we do not create a new language or a language extension, the programmer need not give up the abundant tool support available for Java. The solution consists of two parts: the user-defined code explained in this section, and the generic code of our tool SNITCH (StatIc Type Checking for Hadoop), explained in the next section.

Figure 6 contains a version of the main program that uses our combinators. Compare this code with the previous main program (`runMinLength` in Figure 3). It also first sets input and output paths. Then instance objects for the `mapper`, the erroneous and the correct `combiner`, and the `reducer` class are created. Next, if we comment the code in, we call a `check` function on the (three constituting functions of the) original MapReduce program, including the erroneous `combiner` function. At run time, the `check` function does not do much. At compile time, it ensures the correct typing of the three instance objects passed as parameters; in this case, it will fail. Then we (type) check the (functions of the) new MapReduce program successfully. Next we use function `configureTypeSafeJob` to generate a job object, which internally stores the configuration of a Hadoop program. This replaces the five calls of method `job.set`*`Whatever`*`Class` in the previous main program. The job object is then submitted to Hadoop for execution as before.

In the following section, we will explain the functions `check` and `configureTypeSafeJob` used in this code. For now, it suffices to know that the signature of function `check` alone guarantees correctness of the types of the instance objects passed as parameters.[4] The generation step uses reflection to produce a job equivalent to the one created in the original main program, and also performs a type check internally. Thus, in a real program, one could omit the two checks above and the four long lines with generic type declarations, and instead pass the objects directly to function `configureTypeSafeJob`. We include these lines to show the exact types, and because we use the declared variables three times.

In summary, the user has to spend some additional effort, but, in exchange, saves effort in another place: the Hadoop job configuration is generated automatically by the combinator code. The generic combinator code is provided by us, and it can be reused for all Hadoop programs. It guarantees that all type errors (arising from incompatibilities between the `mapper`, `combiner`, and `reducer` functions) will be detected statically using a standard Java compiler. The combinator code can be written non-invasively, without modifying the MapReduce framework used (Hadoop).

5. TECHNICAL DETAILS

This section is about the implementation details of SNITCH: the type checking function for Hadoop programs, the chaining combinators used therein, and how it automatically generates a job configuration.

5.1 Type Checking

The code for type checking is shown in Figure 7. We use indentation here and in the following figures to show matching type parameters in the same column, if possible. Function `check` takes a `mapper`, a `combiner`, and a `reducer` object as input. For type checking, *only the signature* of function `check` is important, its body (implementation) does not matter. The implementation only serves to understand better why the signature is correct. For those interested, let us describe how we derived the types, in particular, the generic type parameters.

Function `check` uses one of our chaining combinators (class `AutoChain` with six type parameters, described in Section 5.2) to check the interface between the `mapper` and the `combiner` function, and another one to check the interface between the result and the `reducer` function. In total, this function has eight type parameters (K1 to K4 and V1 to V4). It returns a pair (`Tuple`) of `null` values; what makes them useful is only their type information: they are declared to be of the generic type that the intermediate keys and values should have.

Using methods `toMChain` and `toRChain`, the chaining combinators can generate objects that simulate internal Hadoop objects (of types `Mapper.Context` and `Reducer.Context`, respectively). The resulting objects are subtypes of the Hadoop types. In the code section commented out, we call these methodes to verify that the types of our chaining combinators match those used by Hadoop: method `run` of each user-defined function is called with the appropriate Hadoop object as parameter.[5] This code should be

[4]There is one case in which SNITCH will only issue a warning instead of an error: if you call the `check` function with raw types, for example with `Reducer` instead of `Reducer<A,B,C,D>` for some types A, B, C and D. This is due to Java's backward compatibility with code that was written before Generics were introduced.

[5]This does not yet prevent all errors. There is another restriction in Hadoop that is not expressed in the type system: a `combiner` function is cast to `Reducer<K,V,K,V>`, identifying the types of input and output parameters. This simplifies the testing of MapReduce programs, as any program with `combiner` function is also type-correct when omitting the `combiner` function. To model this restriction in our type checker, we would simply have to identify K2 with K3 and V2 with V3.

```
1  static int minLengthTyped(Job job, Path iPath, Path oPath) throws ...
2  {
3      FileInputFormat.addInputPath(job, iPath);
4      FileOutputFormat.setOutputPath(job, oPath);

6      Mapper<Object              ,Text, Pair<Person,Person>,Text> map      = new MinMapperForText();
7      Reducer<Pair<Person,Person>,Text, Person             ,Text> combine1 = new MinLengthReducer();
8      Reducer<Pair<Person,Person>,Text, Pair<Person,Person>,Text> combine2 = new MinLengthCombiner();
9      Reducer<Pair<Person,Person>,Text, Person             ,Text> reduce   = new MinLengthReducer();

11     //check(map, combine1, reduce);    // This runtime error will now be detected at compile time.
12     check(map, combine2, reduce);      // The corrected version type checks.

14     configureTypeSafeJob(job, map, combine2, reduce, Pair.class, Text.class);

16     return (job.waitForCompletion(true) ? 0 : 1);
17 }
```

Figure 6: The new, statically checked main program.

```
1  public static<K1,V1, K2,V2, K3,V3, K4,V4>
2  Tuple<K3,V3> check(
3         Mapper<K1,V1, K2,V2> map,
4         Reducer      <K2,V2, K3,V3> combine,
5         Reducer             <K3,V3, K4,V4> reduce)
6  {
7      AutoChain<K1,V1, K2,V2, K3,V3> mapChain = new
8      AutoChain<K1,V1, K2,V2, K3,V3>(map, combine);

10     AutoChain       <K2,V2, K3,V3, K4,V4> combineChain=
11        new AutoChain<K2,V2, K3,V3, K4,V4>
12        (mapChain.next(), reduce);

14     Chain                  <K3,V3, K4,V4> reduceChain =
15        combineChain.next();

17     /* try {
18         map.run(mapChain.toMChain());
19         combine.run(combineChain.toRChain());
20         reduce.run(reduceChain.toRChain());
21     } catch (IOException e) { // empty
22     } catch (InterruptedException e) { // empty
23     } */

25     return new Tuple       <K3,V3>
26         (combineChain.getNullAsOutputKey(),
27          combineChain.getNullAsOutputValue());
28 }
```

Figure 7: Generic code for type checking Hadoop programs.

used only at compile time to verify changes to the type checking code; at run time, the generated internal objects are dysfunctional and will always fail. Therefore, the code is left in comments.

5.2 Chaining Combinators

The chaining combinators are conceptually higher-order functions, because they are functions that take functions as input. In Java, they must be coded as classes. They must be generically typed to allow for static type checking. In fact, they are special cases of the function composition combinator used in functional programming. In Figure 8, this is made evident by the highlighted type parameters: the first `AutoChain` constructor takes a (`mapper`) function from type (K1,V1) to (K2,V2) and a (`reducer`) function from type (K2,V2) to (K3,V3) as parameters. This is the type signature of function composition. So, an `AutoChain` models a function composition. A single function is represented by class `Chain`. Since an `AutoChain` instance represents two functions, there are two ways to convert it to a `Chain`:

- To select the first function, just use `AutoChain`, because it is a subtype of `Chain` and has in its subtype declaration the generic type parameters K1, V1, K2, V2 of the first function.
- To select the second function, call method `next`, which returns a `Chain` for the second function.

In Hadoop, there are types for both `mapper` (`Mapper`) and `reducer` (`Reducer`) functions, but there is no dedicated type for `combiner` functions: they are also of type `Reducer`. As a consequence, we can also use the first constructor of class `AutoChain` to compose a `mapper` and a `combiner` function, as illustrated in Figure 7. The second constructor then allows us to compose the result of the composition with a `reducer` function. This way, we have modelled and typed a complete MapReduce job.[6]

```
1  class Chain                <K1,V1, K2,V2>
2  {
3      public Chain(Mapper    <K1,V1, K2,V2> mapper) {...}
4      ...
5      public MChain          <K1,V1, K2,V2> toMChain()...
6      public RChain          <K1,V1, K2,V2> toRChain()...

8      K2 getNullAsOutputKey()   { return null; }
9      V2 getNullAsOutputValue() { return null; }
10 }

12 class AutoChain            <K1,V1, K2,V2, K3,V3>
13     extends Chain          <K1,V1, K2,V2>
14 {
15     public AutoChain(Mapper<K1,V1, K2,V2> mapper,
16                      Reducer      <K2,V2, K3,V3>
17                      reducer) {...}

19     public AutoChain(Chain <K1,V1, K2,V2> chain,
20                      Reducer      <K2,V2, K3,V3>
21                      reducer) {...}

23     public Chain                  <K2,V2, K3,V3> next()
24         {...}
25 }
```

Figure 8: Generic code of the chaining combinators.

The constructor of class `Chain` simply wraps a `mapper` as a combinator. Methods `toMChain` and `toRChain` create objects of subtypes of internal Hadoop types, as we have described in Subsection 5.1. The remaining two methods `getNullAsOutputKey` and `getNullAsOutputValue` in this class do what their name suggests: they return a `null` value having the type of output keys and values, respectively. In this code, it is obvious that these types must be K2 and V2, respectively. They allow us to propagate the information about the types needed to their callers.

[6]Since `combiner` functions are optional, we also need a `check` function without a `combiner` parameter. We have left it out as it does not add any insight.

5.3 Generating Legacy Job Configurations

For flexibility, Hadoop jobs are assembled from the three functions (`mapper`, `combiner`, and `reducer`) using run-time reflection. We can use this facility to generate the references to reflected classes from the objects we use in our type checking function (cf. Figure 9). This way, we can generate the Hadoop job configuration that the user would otherwise have to write by hand. In our approach, the user replaces the code for manually creating a job configuration with code to enable static type checking, which, in turn, automatically generates the job configuration needed.

```
1  static    <K1,V1, K2,V2, K3,V3, K4,V4>
2  Tuple                   <K3,V3>
3  configureTypeSafeJob(Job job,
4      Mapper<K1,V1, K2,V2> map,
5      Reducer      <K2,V2, K3,V3> combine,
6      Reducer             <K3,V3, K4,V4> reduce,
7      Class            <?/*K3*/> keyClass,
8      Class               <?/*V3*/> valueClass) throws...
9  {
10     job.setMapperClass(map.getClass());
11     job.setCombinerClass(combine.getClass());
12     job.setReducerClass(reduce.getClass());

14     job.setOutputKeyClass(keyClass);
15     job.setOutputValueClass(valueClass);

17     return check(map, combine, reduce);
18 }
```

Figure 9: The code for generating Hadoop job configurations.

To create reflective `Class` objects from the function objects for `mapper`, `combiner`, and `reducer`, we use the predefined Java method `getClass`, available for every object (Figure 9). This gives us the "class values" needed to create a Hadoop job.

Then we replicate the additional "type declarations" from the original main program. Their values are parameters to function `configureTypeSafeJob`, but they are effectively untyped: the type `Class<?>` stands for any instantiation of the generic type `Class` used by the Java reflection mechanism.

We would like to supply the generic type parameters K3 and V3 here, as the comments suggest. This is impractical in Java: if we use nested type parameters in our user-defined functions, such as `Pair<Person,Person>` for K3, the type parameter K3 has itself type parameters. Java does not allow a "class value" for generically typed classes.[7] In consequence, a strongly typed variant of this method would not be usable in general. Nevertheless, one could write an additional strongly typed variant to be used only for flat generic type parameters.

Remember that, for a static check of the parameters, the signature of function `configureTypeSafeJob` alone suffices. Nevertheless, we call function `check` in the last line of of function `configureTypeSafeJob` for two reasons: as a sanity check, and to be able to put the type parameters K3 and V3 in the return type of function `configureTypeSafeJob`. This enables the user to match them manually with the parameter objects for the additional "type declarations" discussed above.

6. SUBTYPING

In this section, we describe an additional feature of our implementation that is of interest to programmers who assemble Map-

[7] This is due to *type erasure*: in Java, all generic type information is erased at compile time [4]. Because of type erasure, in Figure 3, we have to use `Pair.class` without any reference to its two type parameters of class `Person`.

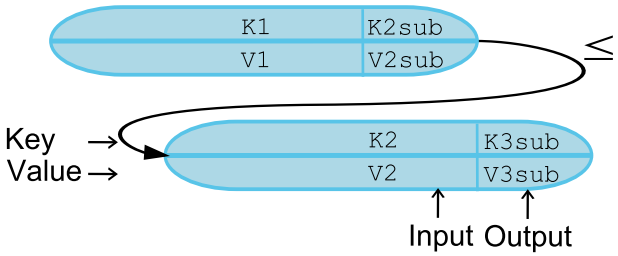

Figure 10: The match with subtyping ($\leq$) between `combiner` (top) and `reducer` function (bottom).

Reduce programs from a pool of already implemented `mapper`, `combiner`, and `reducer` functions.

Suppose you have written a Hadoop program with a `reducer` function that counts the number of key/value-pairs in each group. This `reducer` function is fairly general. Consequently, if you want to reuse it in another (arbitrary) Hadoop program, it should accept any value of the top type `Object` as input value. This means that you must declare it to have `Object` as the type of input keys, and of input values. (You could have declared it to have more specialized input types.) In the type checker developed so far, this means that we would have to *change* the types of the two `mapper` functions involved: they must now return values of *exactly the top type*. This, in turn, would render them incompatible with most other, more specialized `reducer` functions. All in all, this constitutes an unwanted interaction with the type system, hindering the reuse of the user-defined functions.

The restriction is imposed by the Java 5 compiler used to check the Hadoop program: it cannot detect safe upcasts between types with different instantiations of generic type parameters. So, at some point, a human must decide whether to allow a type cast, and will then have to guarantee that the cast will not be an unsafe downcast in any instantiation. We have relegated this task to our type checker. We have included the safe casts, and omitted the others.

Fortunately, the decision which casts are safe is relatively straightforward, since a similar concept has long been present in object-oriented programming: defining subtypes of function types. There, we need *contravariance* for function arguments and *covariance* for function return values [9]: function l *is a subtype* of function u iff l accepts at least the values accepted by u and u returns at least the values returned by l. So, a subtype may additionally accept as parameters (read, in our case) supertypes of the original parameter types, and return (write, in our case) subtypes of the original return types.

With this theoretical armament, it is easy to extend our checker to support subtyping. In the modified checker, instead of mandating exactly matching instantiations of the type parameters of the user-defined functions, we relax this constraint to an implication: when chaining two user-defined functions, the first may produce more specific intermediate results than the second has declared to accept (cf. Figure 10). Implementing this, we must make use of generic type bounds in Java, which allows us to define upper or lower bounds (sub- or supertypes) on the generic type variables in a declaration. In the type checker, in the chaining combinators on which it is built and in the generator for job configurations, we need additional type parameters for the types of keys and values before upcasting them. The remaining modifications are straightforward.

Figure 11 contains the code of our subtyping-enabled checker. All differences to Figure 7 are highlighted in bold. The static function `checkSubtypes` has four additional type parameters (in com-

```
1  public static   <K1,V1, K2,V2, K2sub extends K2,V2sub extends V2, K3,V3, K3sub extends K3,V3sub extends V3, K4,V4>
2  Tuple<K3sub,V3sub> checkSubtypes(
3          Mapper  <K1,V1,         K2sub,V2sub> map,
4          Reducer         <K2,V2,              K3sub,V3sub> combine,
5          Reducer                                          <K3,V3, K4,V4> reduce)
6  {
7      AutoChainSub<K1,V1, K2,V2, K2sub,V2sub, K3sub,V3sub> mapChain = new
8      AutoChainSub<K1,V1, K2,V2, K2sub,V2sub, K3sub,V3sub>(map, combine);

10     AutoChainSub       <K2,V2,                                  K3,V3, K3sub,V3sub, K4,V4> combineChain = new
11     AutoChainSub       <K2,V2,                                  K3,V3, K3sub,V3sub, K4,V4>(mapChain.next(), reduce);

13     Chain                                                      <K3,V3,              K4,V4> reduceChain =
14         combineChain.next();

16     /* the code previously shown here remains unchanged */

18     return new Tuple<K3sub,V3sub>(combineChain.getNullAsOutputKey(), combineChain.getNullAsOutputValue());
19 }
```

Figure 11: The generic code for type checking Hadoop programs, allowing for subtyping.

parison with `check`) that are declared to be subtypes of existing ones. So, the existing type parameters are upper type bounds. An example of this is the type parameter `K2sub` with the upper type bound `K2`. These type parameters are then used in a correctly chosen subset of declarations of parameters and local variables. For example, `K2sub` is the new type of the keys the `mapper` function produces, and is therefore used in the declaration of the function parameter `map` and the local variable `mapChain`.

During this adaptation, it is all the more important to use exactly the declared type of the values in question. For example, the type of `combineChain.getNullAsOutputKey()` is K3sub, a proper subtype of K3. If we used Tuple<K3,K3> as the constructor and return type, we would have written code that makes the compiler introduce an unwanted upcast here, thus reducing the amount of type information available to the user.

```
1  /* class Chain    <K1,V1, K2,V2> remains unchanged */

3  class AutoChainSub<K1,V1, K2,V2, K2sub extends K2,
4                 V2sub extends V2, K3,V3>
5      extends Chain <K1,V1, K2sub,V2sub>
6  {
7      public AutoChainSub(
8             Mapper <K1,V1, K2sub,V2sub> mapper,
9             Reducer       <K2,V2, K3,V3> reducer) {...}

11     public AutoChainSub(Chain<K1,V1, K2sub,V2sub>chain,
12                         Reducer     <K2,V2, K3,V3>
13                         nextReducer) {...}

15     public Chain                    <K2,V2, K3,V3>
16            next() {...}
17 }
```

Figure 12: An extract of the generic code of the chaining combinators for use with subtyping.

In Figure 12, we have highlighted the changes with respect to Figure 8. Class `Chain` is exactly the same as before. Compared to class `AutoChain`, we introduce two additional type parameters with type bounds to class `AutoChainSub`. Finally, Figure 13 shows the necessary modifications of the generator shown in Figure 9. These modifications are analogous to those applied to the `checkSubtypes` function in Figure 11. As there are no local variable declarations, the function body is not changed at all.

With this extension, fewer MapReduce programs will be wrongly marked as being erroneous. Since the extension does not let any additional type errors pass undetected, we have made it the default (although the variant without permitting subtyping remains available). In summary, our type checker is now capable of checking Hadoop programs that use subtyping, without producing large amounts of false positives, as it did before.[8]

7. RELATED WORK

Domain-specific languages built on top of MapReduce, such as Pig Latin [8], are frequently compiled to Hadoop code. Thus, it is possible to integrate static type checks in the compilation process. Yet, many programmers write their code for Hadoop directly in Java, and we enable them to find program errors more quickly.

There are also many MapReduce implementations written in dynamically typed programming languages.[9] These languages do not offer any static type check at all. You could also write a static checker for MapReduce programs written in these languages, but this does not show much promise.

The industrial push for the MapReduce programming model came from Google in an imperative setting. The open-source framework Hadoop is also imperative, based on Java, and our work caters directly to the Hadoop and Java community.

However, the roots of MapReduce lie farther back and come from functional programming, where `map` and `reduce` are two of the central higher-order functions. A higher-order function, which takes one or more functions as parameters and/or supplies them as result can be viewed as a program skeleton. A program skeleton can be a precise and convenient way of specifying generic parallelism [10]. Program skeletons come with efficient implementations for specific parallel machines.

One example is MapReduce, whose implementations have so far been aimed at large homogenous cluster platforms. There have been a number of functional specifications of MapReduce [1, 3, 7]. They are naturally type-safe and do not incur the problems which we encounter in the Hadoop setting. And they have not been tuned as seriously for large-scale, high-performance applications as the imperative implementations.

[8] As mentioned in Footnote 5, the genericity of `combiner` functions in Hadoop is lower than described here. This means that, to model this restriction, we would have to identify the type K2 with K3sub, and V2 with V3sub in function `checkSubtypes` in Figure 11. As a consequence, K2sub would be a subtype of K3sub, and V2sub a subtype of V3sub. Nevertheless, all advantages introduced with this extension would be preserved.

[9] One example, written in Erlang and Python, is "disco – massive data - minimal code" (`http://discoproject.org/`).

```
1  public static<K1,V1, K2,V2, K2sub extends K2,V2sub extends V2, K3,V3, K3sub extends K3,V3sub extends V3, K4,V4>
2  Tuple<K3sub,V3sub> configureTypeSafeJobSubtypes(Job job,
3        Mapper <K1,V1,        K2sub,V2sub> map,
4        Reducer       <K2,V2,                                    K3sub,V3sub> combine,
5        Reducer                                              <K3,V3,                                    K4,V4>reduce,
6        Class                                             <?/*K3*/> keyClass,
7        Class                                                <?/*V3*/> valueClass) throws ...
8  {
9      /* the code previously shown here remains unchanged */

11     return checkSubtypes(map, combine, reduce);
12 }
```

Figure 13: The code for generating Hadoop job configurations, for the subtyping variant.

The skeleton approaches most similar to our work here are implemented as frameworks in object-oriented languages with parametric polymorphism (prevalently C++). One such framework is an anonymous C++ skeleton library [6], whose present version is called Muesli.[10] It uses C++ templates to ensure type correctness. C++ templates are a metaprogramming concept. They are checked at compile time; to be precise, the C++ program generated by the template metaprogram is checked when it is compiled to binary code. As a consequence, C++ templates and the skeletons using them are only checked if there is code that instantiates them. This process can be viewed as static checking, but the programmer must assure that the code in question is used in the final program.

8. CONCLUSIONS AND FURTHER WORK

We have shown that type errors can be introduced easily when programming with common MapReduce implementations. Without special treatment, these errors go undetected until a failure occurs at run time. As a consequence, one can imagine failures occurring arbitrarily late in a long, distributed computation. We have proposed and implemented an automatic method to detect this kind of type errors in Hadoop programs at compile time. Using our implementation is as simple as writing the main MapReduce program following the scheme of Figure 6, rather than that of Figure 3.

In further work, we plan to provide better error messages for the type errors found. Presently, our implementation transforms type errors in MapReduce programs to type errors in Java programs with messages that are unspecific regarding MapReduce concepts. We plan to extend Eclipse and its Java compiler `ecj` to using special knowledge of MapReduce concepts like `mapper` and `reducer` functions when providing explanations for type errors detected in MapReduce programs.

Furthermore, we will investigate the problem of the additional type parameters presently supplied by reflection. In a Hadoop program, not only the user-defined functions need to be given by reflection, but also the types of the intermediate keys and values (for communication over the network). We cannot generate these types in the current implementation, as we would first have to create instances, and this may be arbitrarily difficult.

We would like to reduce the amount of user code needed for type checking Hadoop programs to zero. To this end, we need to extract all information needed from main programs represented as Java code or in one of the alternative representations mentioned earlier. This will enable us to check any legacy Hadoop program, for example, any one submitted to a cluster for execution.

Finally, the problem of inter-job type safety has not yet been attacked. With this work, we have presented a solution for intra-job type safety. Yet, as we have explained in Section 1, MapReduce workflows can consist of multiple MapReduce jobs. We would like to consider also the data flow from one job to the next in such a workflow, as incompatibilities at this level can lead to run-time failures as well. We strive to present a practical solution to detect such errors statically, too.

[10] `http://www.wi.uni-muenster.de/PI/forschung/Skeletons/`

9. ACKNOWLEDGEMENTS

We would like to thank the members of our group, Jörg Liebig in particular, for fruitful discussions of earlier drafts of this paper. We thank Johannes Henneberg for providing Figure 1.

10. REFERENCES

[1] J. Berthold, M. Dieterle, and R. Loogen. Implementing Parallel Google Map-Reduce in Eden. In *Proc. Euro-Par*, LNCS 5704, pages 990–1002, 2009.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Comm. ACM*, 51(1):107–113, 2008.

[3] C. A. Herrmann and C. Lengauer. Transforming Functional Prototypes to Efficient Parallel Programs. In Rabhi and Gorlatch [10], chapter 3, pages 65–94.

[4] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Proc. OOPSLA*, pages 132–146, 1999.

[5] C. Jardak, J. Riihijärvi, F. Oldewurtel, and P. Mähönen. Parallel Processing of Data from Very Large-Scale Wireless Sensor Networks. In *Proc. HPDC Workshops*, pages 787–794, 2010.

[6] H. Kuchen and J. Striegnitz. Features from Functional Programming for a C++ Skeleton Library. *Concurrency Computat.: Pract. Exper.*, 17(7–8):739–756, 2005.

[7] R. Lämmel. Google's MapReduce Programming Model – Revisited. *Sci. Comput. Program.*, 70(1):1–30, 2008.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proc. SIGMOD*, pages 1099–1110, 2008.

[9] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[10] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[11] K. Wiley, A. Connolly, J. P. Gardner, S. Krughof, M. Balazinska, B. Howe, Y. Kwon, and Y. Bu. Astronomy in the Cloud: Using MapReduce for Image Coaddition. *CoRR*, abs/1010.1015, 2010.