Perldoop2: a Big Data-oriented source-to-source Perl-Java compiler

César Piñeiro, José M. Abuín and Juan C. Pichel
Centro de Investigación en Tecnoloxías da Información (CIITIUS)
Universidade de Santiago de Compostela
Santiago de Compostela, Spain
Email: {cesaralfredo.pineiro, josemanuel.abuin, juanCarlos.pichel}@usc.es

Abstract—Perl is one of the most important programming languages in many research areas. However, the most relevant Big Data frameworks, Apache Hadoop, Apache Spark and Apache Storm, do not support natively this language. To take advantage of these Big Data engines Perl programmers should port their applications to Java or Scala, which requires a huge effort, or use utilities as Hadoop Streaming with the corresponding degradation in the performance. For this reason we introduce Perldoop2, a Big Data-oriented Perl-Java source-to-source compiler. The compiler is able to generate Java code from Perl applications for sequential execution, but also for running on clusters taking advantage of Hadoop, Spark and Storm engines. Perl programmers only need to tag the source code in order to use the compiler. Experimental results demonstrate the benefits of Perldoop2 in terms of ease of use, performance and scalability.

I. INTRODUCTION

We are living in the Big Data era. This data come from all type of sources: sensors used to obtain information on the climate, publications in social networks, blogs, digital images and video, etc. One of the main characteristics of this amount of information is the fact that, in many cases, is not structured. In order to process this information several frameworks have been proposed. Nowadays the de-facto standards for parallel processing of Big Data are Apache Hadoop [1] and Apache Spark [2] engines. Hadoop follows the MapReduce programming model [3], and it was implemented in Java. Even though code developing in Hadoop is largely simplified with its characteristics as the automatic input splitting, task scheduling or fault tolerance mechanism, to write a Java MapReduce program is not straightforward. Spark was designed to overcome some of the Hadoop limitations, especially when considering iterative jobs. It supports both in-memory and on-disk computations in a fault tolerant manner by introducing the idea of Resilient Distributed Datasets (RDDs). Apart from running interactively using Python and Scala, Spark can also be linked into applications in either Java, Scala, or Python. Another important framework is Apache Storm [4], which is focused on processing streaming data in real time. In this case, Java is the only natively supported language.

On the other hand, users in several research areas are not familiar with the languages supported by Hadoop, Spark and Storm, especially Java and Scala. In Bioinformatics and Natural Language Processing (NLP), for example, many applications were developed using the scripting language Perl. Interpreters are generally slow, which makes scripting languages prohibitive for implementing large, data and CPU-intensive applications. As a consequence, Big Data technologies fit in a natural way as solution for processing huge amounts of data in reasonable time. Nevertheless, porting Perl applications to Java or Scala is a relatively difficult task as the differences between both languages are huge. Hadoop gives an opportunity to Perl programmers in order to take advantage of parallel systems providing an utility called Hadoop Streaming. This tool allows to execute in parallel codes written in any programming language. However, the ease of use provided by Hadoop Streaming comes at the expense of important degradations in the performance with respect to Java codes [5]. To overcome those problems we introduce Perldoop2, a Big Data-oriented Perl-Java source-to-source compiler. The main contributions of Perldoop2 are the following:

• To the best of our knowledge, it is the first working source-to-source Perl-Java compiler. In addition, Perldoop2 is also the first effort towards a compiler that automatically generates Big Data codes. In particular, it is capable of producing Java code for Hadoop, Spark and Storm frameworks.
• It is open-source (source code available at a public repository1).
• No knowledge about Java is necessary. Users only need to tag the Perl source code to assign a datatype when a variable is declared.
• The supported Perl syntax is comprehensive enough to compile applications from many scientific areas. Note that the important differences between Perl and Java make a direct and effective translation of the source code impossible. In this way, the syntax of Perl is limited to make the translation possible. If unsupported syntax appears in the source code, Perldoop2 raises an error including debugging information.
• Perldoop2 allows to use Java classes to replace non-translatable Perl module dependencies.
• We must highlight that Perldoop2 Java codes obtain similar performance results with respect to hand-coded Java applications. In addition, we demonstrate the use-

1https://github.com/citiususc/perldoop2
The paper is structured as follows: Section II explains the main difficulties to translate Perl code to Java, presents some related work and discusses about the limitations of the first version of PRLDOOP. Section III describes the PRLDOOP2 compiler in detail. In Section IV the tagging process is described. Section V details the particularities of generating Java code for Hadoop, Spark and Storm using PRLDOOP2. Section VI shows the experimental results. Finally, the main conclusions derived from this work are summarized.

II. BACKGROUND & RELATED WORK

A. Translating Perl to Java

In general, the automatic translation of a Perl script to an equivalent Java source code is an almost impossible task since differences between both languages are too large. Perl is an interpreted programming language with a very permissive syntax and with weakly typed variables. On the other hand, Java is a pseudo-compiled language with a very strict syntax and strongly typed variables.

Perl has a Turing-complete grammar [6], [7] which allows almost total customization of its syntax through directives and libraries. These customizations are carried out at runtime which makes it impossible to predict the behavior of a Perl script until it is executed. In this way, in order to make the translation process possible, the Perl syntax has been reduced to a context-free language recognizable by a Look-Ahead LR (LALR) parser, which allows us to recognize the procedural syntax of Perl (objects are not supported) without any kind of personalization directive. PRLDOOP2 uses a Java implementation of Lex and Yacc [8] to analyze the Java source code and produce a Java compatible code.

Our approach to deal with the translation from a weakly typed language (Perl) to a strong typed one (Java) requires to tag the Perl source code including information about the type of each declared variable. Note that the validation of types is similar to the one used in Flow and TypeScript but the goal is completely different. Flow and TypeScript include static type annotations to increase the readability of JavaScript codes and avoid type errors, while in our case types are necessary to translate from Perl to Java. Other weakly typed languages like PHP have been compiled to static languages (C++) [9], but C++ is not type secure as Java which facilitates the translation.

There are a number of differences between Perl and Java that must be taken into account when writing a compatible Perl code with a Java translation:

- **Variable declaration:** It is mandatory to declare a variable before using it for the first time.
- **Variable type:** Each variable has only one type and it can not be changed. That is, if you declare a variable as Integer, it cannot store a String. Of course it is possible to redefine the variable in order to use a new type.
- **Collection initialization:** Collections must be initialized before they can be used. In particular, collections can be initialized as an empty collection, copying an existing collection or with the return value from a function.
- **Array Access:** When accessing arrays, indexes should be positive numbers with a value smaller than the size of the collection. In the case of lists this restriction does not apply.
- **Boolean values:** Perl does not have boolean values, which can be assigned to variables. Values are converted into boolean during the translation process instead. Constants are replaced by 'true' or 'false' whereas the expressions are evaluated by functions at runtime.

B. Language migration

As an approximation we can consider that our work deals with the migration from Perl to Java API. In that field we can find a related tool that automatically mines API mapping (MAM) relations from Java to C# [10]. We must highlight that translating Java to C# is much easier than translating Perl to Java because in the first case both languages have similar characteristics and only differ in their syntax. MAM does not need any modification of the source code like PRLDOOP2 because Java and C# have similar datatypes and they can be directly mapped.

In other work the author shows an example of the translation of COBOL source code to Java using an automatic translator [11]. This tool was focused on modernizing a particular system written in COBOL using a programming language more modern and maintainable, so the tool is not for general purpose as PRLDOOP2. In this case the source code is known and fixed. However, both tools focus on adapting a non-object-oriented source code to Java. It is worth noting that a complete translation from COBOL to Java is possible although due to the syntax it is a big challenge. However, the complete translation from Perl to Java is impossible. Unlike PRLDOOP2, this tool does not require modifying the source code because COBOL is a low-level language and it has more information than Java source code. For example, Java hides and manages automatically several aspects such as memory that in COBOL depend on the user. In our case Perl handles the type of variables while Java does not, so it is impossible to predict the types without including additional information.

C. Limitations of PRLDOOP1

PRLDOOP [12] (from now on PRLDOOP1) was proposed by the authors in a previous work. This tool automatically translates NLP Perl scripts prepared to be executed using Hadoop Streaming into Hadoop-ready Java codes. NLP scripts consist of many regular expressions. In this way, the tool required a well defined structure and very little syntax of Perl was supported. Although the usefulness of PRLDOOP1 was demonstrated, it has important limitations.

First, PRLDOOP1 is not a compiler, it is just a translator. In addition, the tool only translates the section of the source code which contains the regular expressions, while the remainder code such as class and constructor declarations, imports of libraries, auxiliary functions, among others, should be implemented by a programmer as a Java template. As a consequence, knowledge about Java is required to use PRLDOOP1. Second, the Perl source code should be tagged in order to facilitate the translation to Java. The tagging process is sometimes confusing for the users as the number of labels is high. And finally, there are strict rules of programming in PRLDOOP1 that oblige to change certain aspects of the Perl source code. For example:

- Ordered conditional blocks: it means that the conditional expression should appear before the sentences to be executed if the condition is fulfilled.
- Perform string concatenations always with the "." operator.
- Restrict the access to array positions not previously allocated.
- Users must use a different name for each variable.
- Expressions in control block must be a boolean, an integer when accessing an array, and a string when considering a hash.

All those limitations have been removed since PRLDOOP2 has been completely redesigned using compiler construction techniques, which include the creation of a lexical and a syntactic analyzer. Next we summarize the most important differences and optimizations of PRLDOOP2 with respect to PRLDOOP1:

1) Templates are not required anymore, all the code and dependencies are generated by the PRLDOOP2 compiler. Knowledge about programming in Java is not necessary.
2) Simplified and improved tagging process. The labels needed has been reduced to the minimal.
3) All programming rules listed above are no longer required.
4) Automatic casting between defined data types without labels.
5) Modular support allowing the translation of custom libraries and auxiliary functions.
6) Advanced error management that sorts and reports the position and source of errors to the users.
7) Strict formatting of the output Java code, preserving the comments of the original Perl source code.

III. THE PRLDOOP2 COMPILER

Compilation is the process of transforming a source code into a binary program that can be executed by a computer. During the process, optimizations are performed and the final result may change depending on the architecture of the target machine. The advantage of source-to-source compilation is that the process is easier and the responsibility of making binary code and apply optimizations rests with the target source compiler. PRLDOOP2 is a Perl-Java source-to-source compiler, which has been implemented in Java. The compilation process can be divided into the stages depicted in Figure 1. Next we describe them in more detail.

A. Lexer

The lexical analysis consists in converting a sequence of characters into a sequence of tokens. PRLDOOP2 has two types of tokens: Perl tokens and tag tokens. Perl tokens are all tokens that can be found in a normal Perl script or application (variables, functions, reserved words, etc.), while tag tokens only exists in the PRLDOOP2 syntax. Tag tokens are written within comments in the source code between `<` and `>` such as `<string>` or `<array>`. A special case are comments. Perl interpreter ignore them when a source code is analyzed but we want to keep them in the destination Java code, so all comments will be interpreted as tags to separate them from Perl tokens.

Note that the lexical analyzer in PRLDOOP2 was implemented using Jflex3, a Lex implementation for Java.

B. Tag Preprocessor

The goal of this stage is to convert a series of Perl tokens into terminals. In this process tag tokens are stored in existing terminals according to the following rules:

1) If tags are at the beginning of the line, tags will be stored in all compatible terminals listed below until the end of the sentence. For instance,
   
   ```perl
   #<integer>
   my $=2;
   ```

2) If tags are at the end of the line, tags will be stored in all compatible terminals in the same line.
   
   ```perl
   my ($x, #<integer>
   $y); #<string>
   ```

3) If a tag is a comment, it will be stored in the last terminal to preserve the position in the translated code. If there is not a previous terminal, an empty terminal will be created.

As a result of this process, the syntactic analysis is not affected by the position of the tags and the parser grammar reduces significantly its complexity.

3http://www.jflex.de
C. Parser

A parser or syntax analyzer is a software component that takes terminals and builds a syntax tree. The parser is responsible for validating the tokens in order to check if the syntax is correct. There are various types of analyzers, ascending and descending, as well as different types of grammars for each of them. The parser in PERLDOOP2 was implemented using BYACC/J\(^4\), a Yacc implementation for Java. In particular, PERLDOOP2 uses an ascending parser with a LALR(1) grammar. This grammar allows to define fast analyzers and do not consume as much memory as the LR counterpart. Perl has an incredibly large syntax and, according to its documentation, cannot be recognized by a parser. However, if we limit the syntax in some cases, an LALR(1) grammar is more than enough to accomplish this task.

A complete syntax tree is constructed to provide the maximum information as possible about the source code. In the translation stage the syntax tree will be traversed to infer the use of expressions and thus be able to address some of Perl’s ambiguities. For example, when using the Perl function

\[
\text{print}\{\}\;
\]  

\[
\text{print }\{\text{STDERR}\} \text{"error";}\quad\# \text{ Case A (Pipe)}
\]

\[
\text{print }\{\text{"key"},\text{"value"}\};\quad\# \text{ Case B (Hash)}
\]

In both cases the behavior of \(\text{print}\{\}\) is totally different. In case A Perl interprets that it should print "error" using the standard error output, while in case B prints the reference to a hash.

D. Translator

The translation stage is responsible for validating and generating Java source code for each node in the syntax tree following a post-order path. The translation phase is divided into two parts: semantic checking and code generation.

1) Semantic checking is responsible for assigning a type to each node and validating if it meets the requirements to generate Java code. For example, if the node is a variable, it must check that the variable exists and then assign its type to the node.

2) The code generator is responsible for generating Java code. The code is created using the direct children of the node and the type assigned in the semantic checking phase. If the node to be analyzed is a terminal, the generated code corresponds to the value of the stored Perl token. In addition, if the terminal contains a comment, it will be copied using the Java comment syntax.

E. Formatter

Finally, the formatter is responsible for making code human readable. To deal with this PERLDOOP2 reformats the Java source code to comply with Google Java Style. This process is optional and can be omitted if the final code will be compiled directly without user modification.

\(^4\)http://byaccj.sourceforge.net

IV. Tagging the Perl source code

Perl codes must be tagged to be compatible with PERLDOOP2. As we have mentioned previously the tagging process has been simplified and improved with respect to PERLDOOP1, reducing the required labels to the minimal. Next we explain the different types of existent tags and how they work.

A. Datatype tags

Perl variables do not have datatypes. For this reason it is necessary to tag the source code in order to define their type. There are three ways to declare the type of a variable:

1) In the line before the statement: datatype tags will affect all variable declarations of the statement, that is, up to the semicolon.

2) At the end of variable declaration: datatype tags only affect variable declarations of this line.

3) Assigning a type before declaring: preceding a tag with the name of a variable, it is possible to assign a type before its declaration.

The basic datatypes supported by PERLDOOP2 are the following: \(<\text{boolean}, <\text{integer}, <\text{long}, <\text{float}, <\text{double}, <\text{number}\) (generic type to store any of the four previous types), \(<\text{string}, <\text{file}\) and \(<\text{box}\) (generic type to store any type of scalar context '$').

Collection datatypes are \(<\text{array}, <\text{list}\) (used instead of an array when the size is unknown), \(<\text{hash}\) and \(<\text{map}\) (both define a collection accessed by a key). Collections must be initialized assigning an empty collection and specifying a size using a tag with a number or a variable after the collection tag. For instance:

\[
\text{my } @a=();\quad\# <\text{array}<3><\text{string}\\
\text{my } @x=3;\quad\# <\text{integer}\\
@a = ();\quad\# <\$x>\\
\]

In addition, nested collections are allowed but they should always end with a basic datatype. Unlike hash and list datatypes, array allows to define all the dimensions in a single initialization:

\[
\text{my } @a=();\quad\# <\text{array}<10><\text{array}<10><\text{string}\\
\text{my } @h=();\quad\# <\text{hash}<\text{hash}<\text{string}\\
\$("key")=\{\};\\
\]

Finally, references are a special case. In Java there is no access to memory so a programmer can reference the value of a variable but if its value changes the reference will not be updated. For that reason, scalar references are not allowed because they do not work. References are defined by \(<\text{ref}\) tag before the first collection tag.

B. Block tags

Block tags are used by PERLDOOP2 to perform in a different way the translation of a particular block of code. The tag must be placed immediately before or after the opening brace of the block. In the current implementation PERLDOOP2 has four types of block tags:

- Main tag: By default in Java all the code outside of the functions (global code) is translated as a block of static
The Mapper class in Java requires four datatypes: two for the input (key, value) pair and two for the output (key, value) pair. Since the input is always a text file, it is not necessary to specify the input types in the Perl code. Output types are specified next to the \texttt{<mapper>} tag (Perl, line 3). Nevertheless, this is not mandatory. In case output types are not specified, strings will be used as default type.

The automatic translation of the WordCount reducer is more complex than the translation of the mapper (see Figure 3). It is not possible to apply a direct translation and it is necessary to specify a translation by sections. Just as the mapper, the Reducer reads from the standard input (Perl, line 2). The translated code of this section includes the definition and initialization of all the variables (Perl, lines 3-6 - Java, lines 12-15). Nested blocks inside this code section are ignored. The only blocks not ignored are those marked as combination (\texttt{<combine>}) or reduction (\texttt{<reduction>)} sections. The first section defines the block containing the reduction operation (Perl, line 13 - Java, line 18) that is performed on all values with the same intermediate key. The reduction section (Perl, line 17 - Java, line 21) is executed after the last execution of the combine block. This section includes the final calculations and stores the result. Both sections can use the variables defined in the main block.

On the other hand, note that next to the \texttt{<reducer>} tag it is necessary to specify two variables that store the (key, value) pair (Perl, line 1). The value will be updated every execution of the combine section. Like the Mapper, the Reducer class in Java requires four datatypes but in this case input and output should be defined by tags. If types are not
In addition to Hadoop codes, PERLDOOP2 is able to generate Java functions that can be used inside Apache Spark applications. Spark is a framework with its own Mapper and Reducer classes, but it has more natural and simple ways of distributing the work thanks to RDDs. RDDs can be created by distributing a collection of objects (e.g., a list or set) or by loading an external dataset from any storage source supported by Hadoop, including the local file system, HDFS, HBase, etc. On created RDDs, Spark supports two types of parallel operations: transformations and actions. Transformations are operations on RDDs that return a new RDD, such as map, filter, join, groupByKey, etc. On the other hand, actions are operations that kick off a computation, returning a result to the Driver program or writing it to storage. Examples of actions are collect, count, take, etc. Note that transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action.

As example we will focus on flatMap transformation (see Figure 4). We must highlight that the process detailed next can be applied to any supported Spark transformation. The first step should be the creation of an RDD from the input data (lines 5-6). flatMap applies a custom function to all the elements of the RDD, so it has a single input argument. This function can be generated by PERLDOOP2 without using special tags like in the case of Hadoop. Note that the invocation of the function generated by PERLDOOP2 is different from a typical Java function (line 7), so it is necessary to explain how the PERLDOOP2 API works.

All functions take as argument an array of Box, which is an abstract datatype that allows casting a datatype without knowing the source type (see Section IV-A). For example, it is possible to store “1” as string and read the Box as an integer. If you need to pass a collection to a function, the <array>, <list> and <hash> tags are equivalent to classes [], PerlList and PerlMap, respectively. The easiest way to pass the collections to a function is by reference following the Perl style. So we store the collection within the class Ref, Ref is a scalar and can be converted into a Box.

Ref ref = new Ref(new PerlList());
class.function(new Box[] {new RefBox(ref)});

The return of the function follows the same principle and it is only necessary to read the Box to get the type of Java data from the Box[] return.

Integer n =
class.functA(..)[0].intValue();
PerlMap m = (PerlMap)
class.functionB(..)[0].refValue().get();

Finally, Storm requires the definition of topologies, which are computational graphs where every node represents individual processing tasks. In the Storm terminology, spouts are the sources of a stream within a topology, which usually read data from an external source. Bolts are the consumers of the streams and they perform calculus and transformation tasks on the received data. PERLDOOP2 generates a generic Storm Bolt template when the <storm> tag is written above a function. If the user wants to create a more complex Bolt, the generated code can be invoked using the procedure explained for Spark.
In the codes in terms of the execution time for the calculation of
a string of written language into its component sentences.

is a NLP module called Sentences [13]. This script performs
the table is sorted by value and printed. The last algorithm
count the number of repetitions of a word in a hash table,
is a simple WordCount: a loop reads from standard input and
third applications process text. In particular, the second script
obtained using the average of 50 executions for each applica-
tion. We have considered three small applications (due to space
limitations the source codes are not displayed). The first Perl
script considered in the experimental evaluation calculates a
set of prime numbers and stores them in a file. The second and
third applications process text. In particular, the second script
is a simple WordCount: a loop reads from standard input and
count the number of repetitions of a word in a hash table,
the table is sorted by value and printed. The last algorithm
is a NLP module called Sentences [13]. This script performs
sentence segmentation, which is the problem of dividing a
string of written language into its component sentences.

Figure 5 shows the performance of the different source
codes in terms of the execution time for the calculation of
prime numbers, WordCount and Sentences, respectively. In the
last two applications we have used as input text the English
Wikipedia (file size 2.1 GB). For all the considered cases Java
outperforms Perl, reaching speedups higher than $2 \times$.

In this way, the performance of applications which mostly
contains those kind of operations is similar when comparing
PERLDOOP2 and hand-coded Java codes. Sentences script is
a purely regular expression NLP module, so as Perl and Java
have different regular expression engines, the performance
may vary depending on the type of expressions used.

B. Big Data frameworks: Hadoop and Spark

Next we will show the benefits of using PERLDOOP2 in
order to automatically generate Java codes ready to take
advantage of the Big Data frameworks Hadoop and Spark.
We have included a comparison with respect to combining
Perl codes and Hadoop Streaming. The experiments shown in
this section were performed on a Big Data cluster installed at
the Galicia Supercomputing Center (CESGA), which consists
of 64 nodes. Each node has an Intel Xeon E5520 processor
and 1 GB of RAM memory. The Hadoop version is the 1.1.2,
while Java and Perl versions are 1.8.0 and 5.10.1 respectively.

We have selected as representative application for the tests
a Part-Of-Speech (PoS) Tagger. This NLP application process
text and is composed of several chained modules. That is, the
output of one module is used as input of the following one.
In particular, the ordered NLP modules in the PoS tagger are:

- **Sentences**: it splits the input text in sentences.
- **Normalizer**: swaps some elements like abbreviations or
  emoticons, among others, for semantic tags.
- **Tokenizer**: every sentence is transformed in a token se-
  quence.
- **Splitter**: transforms the composed words in contractions.
  E.g., `don't = do + not, we'll = we + will`.
- **NER (Named Entity Recognition)**: it recognizes named
  entities which can contain several words. For instance:
  Santiago de Compostela.
- **PoS Tagger**: the Part-Of-Speech Tagger performs a mor-
  phosyntactic tagging. E.g. **Proper noun, singular** (NNP);
  **Verb, 3rd person singular present** (VBZ).

We must highlight that the Perl source code of the PoS
tagger and the other NLP modules contain thousands of lines
and regular expressions. The tagged Perl codes are available
at Linguakit\footnote{https://github.com/citiususc/Linguakit} repository. We have used the English Wikipedia
as input.

Figure 6 shows the performance in terms of execution time
and speedup of the PoS Tagger on the cluster using different
number of nodes. In this test the application only processes
the first million lines of the Wikipedia. It can be observed that

![Figure 5: Execution time of the applications considering different implementations: Prime numbers (top left), WordCount (top right) and Sentences (bottom).]
there is an important degradation in the performance when considering Perl codes and Hadoop Streaming with respect to using Java both in Hadoop and Spark. This observation agrees with the experiments detailed in [5] where an in-depth analysis of Hadoop Streaming was performed. This confirms the usefulness and good behavior of PERLDOOP analyzing Hadoop Streaming was performed. This confirms the usefulness and good behavior of PERLDOOP.

As a final test to validate the performance and scalability of the Java codes generated by PERLDOOP2 we show the experimental results of using the PoS tagger to process the complete Wikipedia (Figure 7). In this case we have used all the available nodes in the cluster, 64 nodes. Both Java codes outperform again Hadoop Streaming, 1.3× faster on average. In fact, the scalability with respect to sequential Perl are close to the ideal values, 62.2× and 63.1× for Hadoop and Spark respectively.

VII. CONCLUSIONS

In several research areas many applications were implemented using the Perl programming language. However, the most important Big Data engines (Hadoop, Spark and Storm) do not support natively that language. In the particular case of Hadoop, Perl applications could run on clusters using Hadoop Streaming but the performance obtained is far from optimal. On the other hand, porting the applications from Perl to languages natively supported by Big Data frameworks as Java or Scala requires a huge effort. For this reason, we introduce PERLDOOP2, a Big Data-oriented Perl-Java source-to-source compiler. The main goal is to generate quality Java applications from Perl codes with the minimum effort on the part of the users. Note that the unique task required by the PERLDOOP2 users is tagging the Perl source code using a reduced number of labels in such a way that no knowledge about Java is necessary. The translated applications can be directly integrated into the most important Big Data frameworks.

An experimental evaluation was carried out in order to demonstrate the benefits of using PERLDOOP2. First, we have observed that Java codes generated by PERLDOOP2 achieved similar performance than hand-coded applications for sequential execution. In addition, we have generated Java code for Hadoop and Spark engines from several natural language processing applications which consist of thousands of lines of code. Experiments were conducted on a Big Data cluster. Performance results demonstrate the improvements of using Java in Hadoop and Spark with respect to using Perl and Hadoop Streaming in terms of execution time and scalability.

ACKNOWLEDGMENT

This work has been supported by MINECO (TIN2014-54565-JIN and TIN2016-76373-P), Xunta de Galicia (ED431G/08) and European Regional Development Fund.

REFERENCES