A SCALA-BASED DOMAIN SPECIFIC LANGUAGE FOR STRUCTURED DATA REPRESENTATION

Kazuaki Maeda
Department of Business Administration and Information Science, Chubu University, 1200 Matsumoto, Kasugai, Aichi, Japan

Keywords: Data representation, Structured data, Domain specific languages, Scala, Java.

Abstract: This paper describes Sibon, a new representation written in a text-based data format using Scala syntax. The design principle of Sibon is good readability and simplicity of structured data representation. An important feature of Sibon is an executable representation. Once Sibon-related definitions are loaded, the representation can be executed corresponding to the definitions. A program generator was developed to create Scala and Java programs from Sibon definitions. In the author's experience, productivity was improved in the design and implementation of programs that manipulate structured data.

1 INTRODUCTION

A domain-specific language (DSL) is a programming language tailored to a specific application domain(Mernik et al., 2005). It is a special-purpose, and not general-purpose, programming language.

Fowler has explained the difference between external and internal DSLs(Fowler, 2009). An external DSL (e.g., an XML configuration file) is a special-purpose language with a syntax that is different from existing programming languages. In case of an external DSL, the DSL developer has to reuse or build a parser for the domain-specific description. On the other hand, an internal DSL uses the constructs of an existing programming language (called a "host" language) to define the DSL. In case of an internal DSL, the designer extends the host language to the domain specific description, which can improve the development time for building a DSL processor to execute the description.

Scala is a hybrid functional and object-oriented language(Martin Odersky, 2008). It is a simple but powerful programming language. Scala plays an important role as a host language for an internal DSL. One of Scala's appealing features is the fact that parentheses for arguments of methods are optional for infix operator notation; therefore, the descriptions are easier to read and understand than ones in other programming languages.

Another feature is implicit conversion methods. It is used in Scala to extend existing libraries. We can use method calls, for example, capitalize and reverse, to String objects, but these methods are not defined in the String class. A special class RichString in the Scala library wraps the String class, and the Scala compiler converts the String implicitly using the implicit conversion methods.

One more feature in Scala is that functions can be passed as arguments of the methods. It represents a group of program statements; it can be an argument of a method in Scala. It is powerful in its representation of structured data.

This paper describes Sibon[1], an internal DSL for structured data representation in a text-based data format using Scala syntax(Martin Odersky, 2008). An important feature of Sibon is that it is based on Scala and the representation is executable. If Sibon-related definitions are loaded, the representation can be executed corresponding to the definitions. This is useful for Java programs in reconstructing Java objects or for traversing the structured data. Moreover, the author believes that good readability and simplicity of structured data representation are important for all developers.

Section 2 explains structured data representation and Sibon, and Section 3 summarizes this paper.

[1]Sibon stands for Scala Instructions Becoming Object Notation.

Maeda K. (2010).
A SCALA-BASED DOMAIN SPECIFIC LANGUAGE FOR STRUCTURED DATA REPRESENTATION.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 296-299
DOI: 10.5220/0003010502960299
Copyright © SciTePress

2 REPRESENTATION AND MANIPULATION OF STRUCTURED DATA

2.1 Structured Data Representation and XML

Structured data has been widely used in many software development projects. In the case of compiler development, compiler front-ends build abstract syntax trees (ASTs), which represent structured syntactic information of source code(Aho et al., 2006). A variety of representations for structured data have been developed to date(Snodgrass, 1989; Franz and Kistler, 1997; Wilson et al., 1994).

An attractive alternative is XML-based structured data representation. JavaML(Badros, 2000) is a typical XML-based source code representation for ASTs, providing syntactic and semantic information after parsing Java source code. Once software tools are implemented using the JavaML representation, they can easily obtain information about Java source code without the need for analysis. XSDML(Katsuhisa Maruyama, 2004) and srcML(Jonathan I. Maletic and Kagdi, 2004) are other representations in XML that support the representation of formatting information, including white spaces and comments, in addition to ASTs. Therefore, the original source code is restored from the XML representation using the formatting information in XSDML or srcML.

An XML representation is just static data, while the representation in Sibon is dynamic, executable code. Once a representation in Sibon is successfully loaded, the parsing of the data is already done and the representation can be executed. It is useful in reconstructing Java objects from the representation.

2.2 Sibon as an Object Notation

The representation of structured data in Sibon is composed of several elements. Each element has a value and a name. Basically, one element ends with semicolon at the end of line, according to Scala syntax. For example,

```
"April 1" .date;
```

which represents the value as “April 1” and the name of the element as *date*. The element is not just a data representation, but internally, it is also an executable method invocation without parentheses in Scala. The method *data* can be executed using implicit conversion methods.

In Sibon, a structure is represented using functions as arguments. For example, Figure 1 shows that the *config* without the value has three child elements: *time*, *drink*, and *place*. The *time* element’s value is “programming,” the *drink* element’s value is “coffee,” and the *place* element’s value is “coffee stand.”

We can use multiple elements in one line. This action is not recommended, however, since the author believes that simplicity is very important in representing structured data in Sibon.

```
config {
  "programming"  .time;
  "coffee"       .drink;
  "coffee stand" .place;
}
```

Figure 1: An element with three child elements.

```
config {
  pay {
    340       .price;
    0.05      .tax;
    true      .togo;
    "April 1" .date;
  }
  "programming"  .time;
  "coffee"       .drink;
  "coffee stand" .place;
  options {
    "grande"      .option;
    "hot"         .option;
    "double-shot" .option;
  }
}
```

Figure 2: Elements including primitive data types and a collection.

Sibon supports nine primitive data types: *byte*, *char*, *short*, *int*, *long*, *float*, *double*, *boolean*, and *string*. For example, Figure 2 shows that the *pay* element has four child elements: *price*, *tax*, *togo*, and *date*. The *price* element has an integer value 340, the *tax* element has a float value 0.05, the *togo* element has a boolean value true, and the *date* element has a string value “April 1.”

Basically, an element in Sibon cannot have more than one child element with the same name. Recall that a Java class has no more than one field with the same name. An element in Sibon is a similar construct as a Java class. In comparison, an element to represent a collection can have more than one element with the same name. In Figure 2, the *options* element has three child elements with the name *option*. The first *option* element's value is "grande," the second *option* element's value is "hot," and that of the last is "double-shot."

If we need to represent an element linked to another element across the structure, a unique identifier is given to the element, and another element refers to the element using the identifier. Figure 3 shows that a uuid element with the identifier u1da16a5e_c767_ddc7b5077855 is given to the *place* element. The identifier in the figure is calculated using java.util.UUID, which is represented using a symbol in Scala; the *place* element has a reference to the unique identifier. This means that Sibon supports graph-structured data. In the case that a Java program writes graph-structured data to a file, after another Java program reads the data from the file and it constructs a graph data structure, both *place* elements become only one Java object. The value of the *uuid* element can be freely modified using a predefined method.

```
shops {
  place {
    'u1da16a5e_c767_ddc7b5077855 .uuid;
    "Seattle, Washington" .location;
  }
}
config {
  "programming" .time;
  "coffee" .drink;
  'u1da16a5e_c767_ddc7b5077855 .place;
}
```

Figure 3: Elements including a uuid and the reference.

2.3 Definition of Structured Data

Structured data is defined using symbols in Scala and predefined keywords as shown in Table 1. Figure 4 shows the definitions of the representation in Figure 2. The definitions are as follows:

- *config* element has five child elements: *pay*, *time*, *drink*, *place*, and *options*.
- *options* element is a collection of *option* elements
- *pay* element has four child elements: *price*, *tax*, *togo*, and *date*
- *price* element has an integer value
- *tax* element has a float value
- *togo* element has a boolean value
- *date* element has a string value

Table 1: Keywords to define structured data.

keyword	meaning of the keyword
is_child_of	composition of elements
is_element_in_seq	collection of elements
is_a	specialization of an element
is_type_of	primitive data type (int, float, bool, string, et al.)

```
('pay,'time,'drink,'place,'options)
                     .is_child_of 'config;
'option .is_element_in_seq 'options;
('price,'togo,'tax,'date)
                     .is_child_of 'pay;
'int    .is_type_of 'price;
'float  .is_type_of 'tax;
'bool   .is_type_of 'togo;
'string .is_type_of 'date;
```

Figure 4: An example of Sibon definitions.

2.4 Program Generation for Scala and Java

Sibgen reads a Sibon definition, and it generates a Scala program and Java programs. Figure 5 shows the specification as follows;

is_java_package specifies the path "test.sibon" for the Java package

is_java_prefix specifies addition of a prefix "Sbn" at the beginning of the Java class name

is_generated specifies generation of a programming language

is_generated_sibonsetup specifies generation of Sibon-related programs to the file name "setup.scala"

is_generated_diagram specifies generation of a class diagram for generated Java classes to the file name "mydiagram.dot"

```
"test.sibon" .is_java_package;
"Sbn"  .is_java_prefix;
"java" .is_generated;
"setup.scala"   .is_generated_sibonsetup;
"mydiagram.dot" .is_generated_diagram;
```

Figure 5: Specification to generate Scala and Java programs.

Sibon is designed to map the representation to Java classes. The SbnBase class is a base class for all classes generated by Sibgen; it provides three fields:

value, *name* and *uuid*. The Sibon representation and related programs can be compiled and executed. The Scala programs trigger instantiation one after another and instantiate all Java objects.

2.5 Current Implementation

The implementation work has been performed on an Apple MacBook with Mac OS X 10.6.3, Scala 2.7.7, and Java 1.6.0_17. The program generator Sibgen is written in Scala. It reads a Sibon definition file, generates a Scala program to set up, and generates Java classes corresponding to all elements.

The Sibon representation changes to graph-structured data using unique identifiers and references. When a Java program, using Sibon APIs, writes graph-structured objects to a file, the objects with cyclic paths need only be written once. Sibon APIs correctly serialize and deserialize them using the algorithm mentioned in the paper(Birrell et al., 1993).

3 SUMMARY

This paper describes the development of Sibon, a new data representation for graph data structures that uses Scala syntax. One of its important features is that the representation is executable. A program generator Sibgen for Sibon was also developed to create programs from data definitions. It is useful for the development of Java programs to read/write Java objects from/to persistent storage media, or to traverse the structured data.

In the author's experience, Sibon improves productivity in the design and implementation of programs that manipulate graph data structures. Sibon and its related tools are now being used for the development of commercial products, including a compiler front end. Research and development of Sibon will continue to support creation other commercial software products. The results will be published in a future paper.

ACKNOWLEDGEMENTS

This research has been supported by the Kayamori Foundation of Informational Science Advancement.

REFERENCES

Aho, A. V., Lam, M. S., et al. (2006). *Compilers : Principles, Techniques, and Tools*. Pearson Education, 2nd edition.

Badros, G. (2000). JavaML: A Markup Language for Java Source Code. In *9th International World 9th International Wide Web Conference*, http://www9.org/w9cdrom/index.html.

Birrell, A., Nelson, G., et al. (1993). Network objects. In *14th ACM Symposium on Operating Systems Principles*, pages 217–230.

Fowler, M. (2009). *MF Bloki: Domain Specific Language*. http://martinfowler.com/dslwip/.

Franz, M. and Kistler, T. (1997). Slim Binaries. *Communications of the ACM*, 40(12):87–94.

Jonathan I. Maletic, M. C. and Kagdi, H. (2004). Leveraging XML Technologies in Developinging Program Analysis Tools. In *4th International Workshop on Source Code Analysis and Manipulation*, pages 80–85.

Katsuhisa Maruyama, S. Y. (2004). A CASE Tool Platform Using an XML Representation A CASE Tool Platform Using an XML Representation of Java Source Code. In *4th International Workshop on Source Code Analysis and Manipulation*, pages 158–167.

Martin Odersky, Lex Spoon, B. V. (2008). *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344.

Snodgrass, R. (1989). *The Interface Description Language: Definition and Use*. Computer Science Press.

Wilson, R. P., French, R. S., et al. (1994). SUIF: an Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Notices*, 29(12):31–37.