Designing a Type-Safe Remote Procedure Call Protocol

Alisa Vernigor
Faculty of Computer Science
Higher School of Economics
Moscow, Russia
aivernigor@edu.hse.ru

***Abstract*—The goal of this project is to develop a new data serialization format called DependoBuf (DBuf). It supports dependent types and can be used in place of Protocol Buffers (protobuf). We will leverage the latest developments in programming languages and formal methods for type systems to design the DBuf format and use Lex and Yacc to implement a compiler that generates source code to serialize and deserialize DBuf messages in different programming languages. We will evaluate the performance and efficiency of the DBuf format and explore potential integration with gRPC. The aim is to provide developers with a new tool to improve the performance and reliability of their applications. While DBuf is not a drop-in replacement for protobuf, we will ensure that it is as easy as possible for users to transition to DBuf by ensuring that the format is compatible with existing programming languages and type systems.**

***Keywords*—code generation, data formats, dependent types, schema language**

I. Introduction

Data serialization is an essential component of the modern technology landscape, enabling the efficient and accurate transfer of data between different systems and applications. The ability to serialize data, or convert it into a compact and easily transferable format, has become increasingly important as the amount of data generated by systems and devices continues to grow at an unprecedented rate.

However, existing serialization formats such as JSON, XML, and Protobuf are not sufficient to support all use cases. In particular, the lack of dependent types in these formats severely limits the expressiveness of the serialized data. Dependent types, which express constraints on the type of data, are critical for ensuring data consistency and correctness, especially in safety-critical systems and applications.

Research into dependent types has yielded promising results, but so far the only serialization format that supports dependent types is Telegram's Type Language (TL). Unfortunately, TL's syntax is very different from commonly used formats such as Protobuf, and it lacks open source compilers, making it difficult for many developers to adopt.

TL's syntax and limited tools have made it a less-than-ideal choice for developers wishing to use dependent types in their serialization formats. While TL has demonstrated the potential of dependent types, it has not yet achieved widespread adoption due to these issues.

To address these challenges, we propose DependoBuf (DBuf), a new data serialization format that draws inspiration from Protobuf and the dependent type system in Agda. DBuf supports dependent types and has a syntax that is easy to understand and use for developers already familiar with Protobuf. We believe that DBuf will fill a critical gap in the market by providing a serialization format that is both powerful and accessible to a wide range of developers.

II. Literature Review

The need for efficient and standardized methods of transferring data between systems has increased with the advancements in computing technology. Various data formats, including XML [1] and JSON [2], have been created to facilitate data transfer, with text-based formats such as these being some of the first to emerge. While text formats are human-readable, they require a lot of storage space and generate additional network load when sent. While text formats remain popular for tasks where readability is important, binary formats have become increasingly popular, especially for large-scale distributed systems, where performance is critical.

Protocol Buffers (Protobufs) [3] is one of the most popular binary formats today due to its efficiency, interoperability, maintainability, and safety. Protobufs are designed to be language- and platform-neutral, forward- and backward-compatible, and less error-prone than text-based formats because they are strongly typed. The Protoc command-line tool is used to compile Protocol Buffers schema files into source code for various programming languages, freeing developers from the responsibility of serializing and deserializing the data. The adoption of Protobufs has been further accelerated by the widespread adoption of gRPC [4], a high-performance, open-source RPC framework that uses Protobufs for efficient data transfer.

As software systems become more complex, type safety becomes increasingly important. Type safety is the concept that a programming language should prevent operations or expressions from being applied to values of the wrong type. Dependent types take the idea of type safety further by allowing types to depend on values or other types, providing additional safety guarantees such as maintaining invariants or preventing certain types of errors from occurring. Research into dependent types has been going on for a long time, both in theory [5] and in programming languages [6]. However, in the area of data formats, the topic is still rather unexplored.

Current mainstream serialization formats such as JSON, XML, and Protocol Buffers do not support dependent types. Support for dependent types can provide stronger guarantees of correctness and security, particularly in distributed systems handling sensitive or critical data.

However, there has been relatively little research in this area, with the only known work on dependent types in data transfer formats being the Telegram Type Language (TL) [7]. Despite its potential, TL has several limitations. For example, its syntax differs significantly from that of the most common data formats, making it difficult for developers to adopt, and the lack of an open source compiler and comprehensive documentation further limits its usability. There is therefore a clear need for a new open source data transfer format that supports dependent types and is easy for developers to adopt.

III. Methodology

A. Syntax

DependoBuf (DBuf) is a proposed new language that aims to address the issues with the current serialization formats described above. To make it easier for programmers to learn the language, it is based on the syntax of the popular Protobuf data format.

The syntax of a Protobuf file consists of messages, which are structures with fields. Similarly, the basis of DependoBuf is also structured around messages. This ensures that programmers who are already familiar with Protobuf can quickly adapt to the new language.

Dependent types are an important feature of the language, allowing more expressive and accurate type checking. The construction of dependent types is achieved using constructors and pattern matching. Both are included in the language.

The type Nat on Fig. 1 is used as an example to demonstrate how DependoBuf uses constructors. The Nat type has two constructors, Zero and Succ, the latter of which has a field, prev. The curly braces after the constructor list the fields of the structure if created by that constructor.

```
message Nat {
    enum {
        Zero,
        Succ {
            prev Nat
        }
    }
}
```

Listing 1. Type Nat in DependoBuf.

In DependoBuf, dependencies between messages are expressed by listing them in parentheses after the message name. For example, RandomMessage (dep1 DepOne) (dep2 DepTwo) ... (depn DepN).

Pattern matching is performed using the constructors of the message dependencies. If there are multiple dependencies, the constructors are listed in the order described after the dependent message name, separated by commas. The fields of a message created by the dependency constructor can be captured within curly braces following the constructor's name. In this case, these fields are also used in pattern matching. For each case described, the corresponding dependent message constructors with fields are listed in the enum. If a case does not match any of the above, it is matched by the special character *. It is important to note that cases are evaluated in the order in which they are declared.

In the pattern matching example provided in Fig. 2, the RandomMessage type depends on a single Nat value called n. The pattern matching rules specify how the program should handle different cases of the n value. For example, if n is Zero, the program should use the Constructor1, while if n is a Succ constructor with a field prev, the program should use the Constructor2 with a single field number of type Nat equal to prev.

```
message RandomMessage (n Nat) {
    Zero => enum {
        Constructor1 {
            ...
        }
    },
    Succ {number : prev} => enum {
        Constructor22 {
            number Nat
        }
    }
}
```

Listing 2. RandomMessage depend on Nat message, pattern-matching example.

For a more complete understanding of the DependoBuf syntax, a strict EBNF description of the language is included in the Appendix A.

B. Stages of Work and Tools

The first step in creating a schema language is to define its syntax, which involves specifying the rules for how the language's code is written. To implement these rules, we will use a scanner and a parser.

The scanner, also known as a lexer, breaks down the source code into individual tokens. These tokens represent the basic building blocks of the language, such as keywords, identifiers, operators, and literals. We will use the Lex tool to create the scanner, as it allows us to define patterns that match specific tokens in the input text.

Once the scanner has identified the tokens in the input code, the parser takes these tokens and generates a tree-like structure called an Abstract Syntax Tree (AST). This tree represents the structure of the code and its relationships. It serves as the basis for subsequent processing steps. We will use the tool Yacc to create the parser, as it provides a convenient way of specifying the grammar of the language using a BNF-like notation.

To ensure that the syntax of the language is correct, the parser will perform a syntax check by comparing the input code with the rules specified in the grammar. If the input code violates any of these rules, the parser will generate an error message indicating the location and nature of the error.

Once we have a working scanner and parser we can start adding more features to the language, such as type checking, code optimization and code generation. These features will allow us to create a robust and useful schema language, building on the foundation laid by the scanner and parser. In the code generation phase, we will generate code in two popular programming languages: Go and TypeScript. Go is a popular choice for back-end systems due to its efficient performance and seamless integration with protobufs and gRPC, while TypeScript is a widely-used language for front-end development that provides strong typing and supports modern web development practices.

IV. Results

The main expected results of this project are the successful implementation of the parser and scanner stages of the DependoBuf language using Lex and Yacc, and the generation of code in the Go and Typescript programming languages. We aim to demonstrate that the DependoBuf language is capable of generating valid code in multiple languages and that it is a valuable tool for developers who require type safety and dependent types in their projects.

V. Conclusion

We have developed DependoBuf, a data serialization format that supports dependent types. The format offers a secure transport mechanism that reduces the number of errors that occur during data transfer. The security comes from the nature of our language, which ensures type safety and prevents data corruption.

However, DependoBuf has some limitations, including a lack of support for gRPC technology and limited compatibility with programming languages. We plan to address these issues in the future.

Our primary objective is to provide developers with a reliable and secure data format that supports dependent types. We plan to achieve this by defining the language syntax, creating a language scanner, parser, and type checker, and developing code generation for Go and Typescript languages. Testing our language on real systems will help us evaluate its effectiveness.

In summary, we believe that DependoBuf will be an important tool for developers who value type safety and security. As we continue to work on the language, we will strive to address its limitations and extend its compatibility with programming languages.

References

[1] G. C. M. Garcia-Martin, "Extensible markup language (xml) format extension for representing copy control attributes in resource lists," Internet Requests for Comments, RFC Editor, RFC 5364, October 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5364.txt

[2] E. T. Bray, "The javascript object notation (json) data interchange format," Internet Requests for Comments, RFC Editor, RFC 8259, December 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5364.txt

[3] Google LLC, "Protocol buffers documentation," last accessed 21 February 2023. [Online]. Available: https://protobuf.dev/

[4] ——, "gRPC," last accessed 21 February 2023. [Online]. Available: https://grpc.io/

[5] Martin-Löf, *Intuitionistic Type Theory*. Bibliopolis, 1984.

[6] B. C. Pierce, *Types and Programming Languages*. The MIT Press, February 2002.

[7] Telegram Messenger Inc., "Tl language," last accessed 21 February 2023. [Online]. Available: https://core.telegram.org/mtproto/TL

Word Count: 1685

Appendix A
EBNF description of the language

```
schema = {message_definition | service_definition};

message_definition = dependent_message | independent_message;
dependent_message = "message" type_identifier {type_dependency} dependent_message_body nl;
type_dependency = "(" variable_identifier type_spec ")";
dependent_message_body = "{" nl {pattern_matching {"," pattern_matching} "=>" {constructors_block | fields_block}} "}" nl;
pattern_matching = "*" | value | (constructor_identifier ["{" {field_identifier {":" variable_identifier}} "}"] );

independent_message = "message" type_identifier independent_message_body nl;
independent_message_body = fields_block | ("{" {constructors_block | fields_block} "}" nl);

constructors_block = "enum" "{" nl {constructor_identifier {fields_block | nl}} "}" nl;
fields_block = "{" nl {field_declaration} "}" nl;
field_declaration = field_identifier ws type_spec nl;

service_definition = "service" service_identifier "{" nl {rpc_definition} "}" nl;
rpc_definition = "rpc" rpc_identifier "("  [arguments]  ")" "returns" "(" type_spec ")" nl;
arguments = argument {"," argument};
argument = variable_identifier ws type_spec;

uppercase_identifier = uppercase_letter {letter | digit};
lowercase_identifier = lowercase_letter {letter | digit};

type_identifier = uppercase_identifier;
constructor_identifier = uppercase_identifier;
field_identifier = lowercase_identifier;
service_identifier = uppercase_identifier;
rpc_identifier = uppercase_identifier;
variable_identifier = lowercase_identifier;

type_spec = primitive_type | (type_identifier {ws expression});
primitive_type = "Int8" | "Int16" | "Int32" | "Int64"
               | "Float16" | "Float32" | "Float64"
               | "String" | "Bool";

uppercase_letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
                 | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
                 | "U" | "V" | "W" | "X" | "Y" | "Z";
lowercase_letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j"
                 | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t"
                 | "u" | "v" | "w" | "x" | "y" | "z";
letter = uppercase_letter | lowercase_letter;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

expression = value
           | variable_identifier
           | variable_identifier "." field_identifier
           | expression binary_op expression
           | prefix_op expression
           | "(" expression ")"
           ;

value = literal_value | constructed_value;

literal_value = bool_literal
        | float_literal
        | int_literal
        | string_literal
        ;

bool_literal = "true" | "false" ;
float_literal = {digit} "." digit {digit} [ ("e"|"E") [ "+" | "-" ] digit {digit} ] ;
int_literal = digit {digit} ;
string_literal = "\"" { character } "\"";
```

```
65
66 character = letter | digit | symbol | ws;
67 symbol = "~" | "!" | "@" | "#" | "$" | "%" | "^"
68        | "&" | "*" | "(" | ")" | "-" | "_" | "+"
69        | "=" | "{" | "}" | "[" | "]" | "|" | ";"
70        | ":" | "'" | "," | "." | "<" | ">" | "/" | "?"
71        ;
72
73 ws = (" " | "\t") {ws};
74 nl = "\n" | "\r\n";
75
76 constructed_value = constructor_identifier "{" {field_identifier ":" expression} "}";
77
78 binary_op = "+" | "-" | "*" | "/" | "<" | "<=" | ">" | ">=" | "=" | "!=" | "&&" | "||" ;
79 prefix_op = "-" | "!";
```