Abstract:

Protocol buffers (Protobuf) is a novel way of serializing structured data, which can be used in communication protocols or data storage. In this project, I implement a programming language, PF, to manipulate Protobuf data directly. It treats Protobuf objects as its fundamental elements and provides built-in operators and functions to operate on these objects. By using PF, we do not need to worry about the different access functions to handle different fields of a Protobuf data. PF is an new language aiming to facilitate the data operations with Protobuf and to provide a intuitive idea of how to use it.

Introduction:

Protocol buffers is powerful and useful in developing programs for communicating and storing structured data, given the trends of big data. It is language-neutral and platform-neutral. Google developed Protobuf with the goal of providing a better way to communicate between systems. It shares some commons with JSON or XML but it is more efficient and flexible.

To use Protobuf, we can add a message contains several name-value pairs in the .proto file. The message describes the data structure we want to serialize, which is an aggregate containing a set of typed fields. The types could be bool, int32, float, double, string and even message types. A .proto file can contain several message as desired. Fig.1 is an example of a address.proto file. After defining the how our data is structured in Protobuf, we easily write and read to and from our structured data using a variety of languages, such as C++, Java, JavaScript, Ruby, Python and Go. We can use Protocol Compiler to generate corresponding classes or objects and the generated classes or objects provide getters and setters to deal with protobuf data with an efficient binary encoding format. Importantly, we can still read data encoded with old format after extending its format.

message Person {

required string name = 1;

required int32 id = 2;

repeated string email = 3;

message PhoneNumber {

required string number = 1;

optional string type = 2;

}

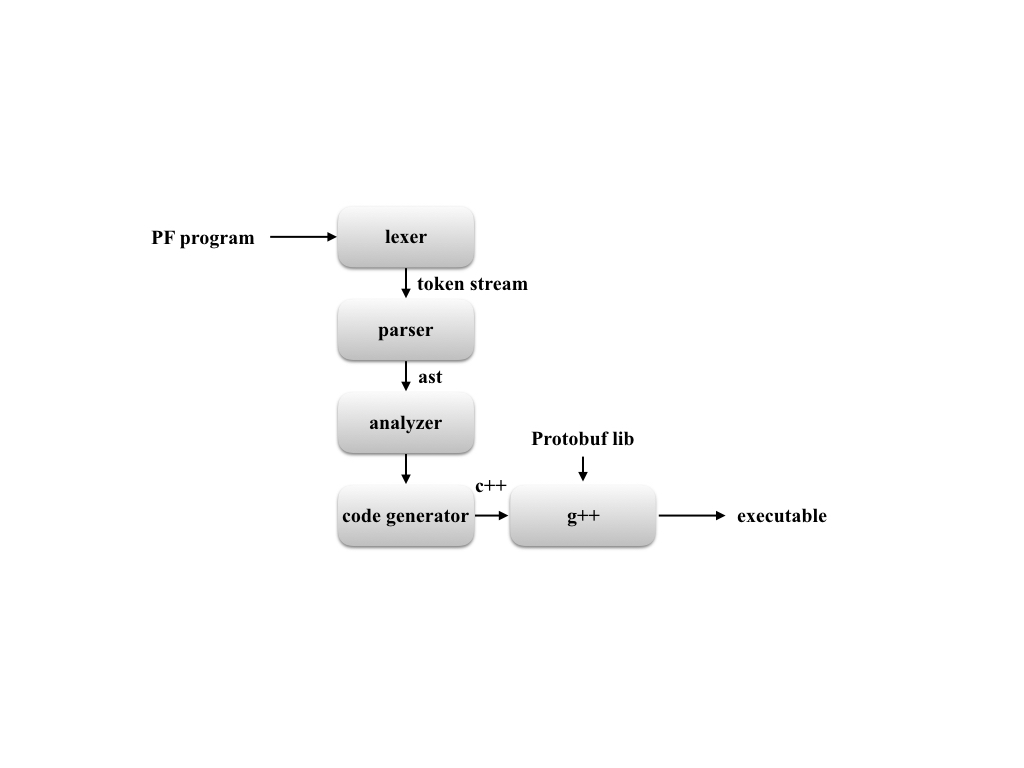
repeated PhoneNumber phones = 4;

}

message AddressBook {

repeated Person people = 1;

}

While these languages have libraries to handle Protobuf data, they are not native. This means that Protobuf is not something fundamental to the language, like the function to the functional language. Moreover, sometimes we may find that accessing elements of a Protobuf data is difficult especially when it contains message types since the access functions could be different for different field types and for repeated fields. PF, a language I implemented in this project, is oriented to handle and manipulate Protobuf data directly. It is designed to be simple, intuitive and specific, and to help developers quickly master how to handle Protobuf data. It has some primitive data types, number, string, bool and a complex data type to store protobuf data object. It provides some built-in functions to operate these data types.

Approach:

The project was developed on a combination of Mac OS X and Ubuntu. The complier was written in OCaml and was developed using vim. The PF code is compiled to C++, allowing for an high-performance execution, and then g++ was used to compile the code to an executable file. Makefiles and Bash scripts were used to compile and test these codes automatically. The architecture on the PF language compiler is shown in Fig.2.

1. Lexer

The lexer was implemented using Ocamllex and the corresponding file is lex.mll. The lexer will take a PF program as input and generate token streams according to the defined lexical rules as output. It will reject programs with illegal syntax. It will also discard unnecessary information such as white space and comments.

2. Parser

The parsers was implemented using ocamlyacc and the corresponding file is parse.mly. It will take the token streams provided by the lexer as input and generate an abstract syntax tree(AST) as output. The AST was parsed using the defined grammatical rules and this process will reject programs which cannot meet he rules.

3. Analyzer

The analyzer was implemented in Ocaml and the corresponding file is analyzer.ml. The analyzer take the AST provided by the parser as input and check the semantic requirement. The semantic checking includes checking on :

1) type requirements:

e.g. We can only assign an expression to a variable which have been declared to the same type;

Arithmetic and comparison operators work on number types only;

Logical operators work on boolean types only;

Protobuf element access operator works on Protobuf variables only.

2) declaration requirements:

e.g. We can only use a variable that has been declared;

We cannot declare a variable that has been declared before.

3) source file requirement:

e.g. The program should provide valid .proto files to declare Protobuf variavles.

4) scope requirements:

e.g. We cannot use a variable that is declared in another scope but has not been declared in its own scope.

5) language-specific requirements:

e.g. The Protobuf elements access statement should be valid.

4. Generator

The generator was implemented in Ocaml and the corresponding file is pf.ml. After checking the semantic requirements by analyzer.ml, pf.ml take the AST as input and generate C++ code as its output.

**Implementation and result**

1. language tutorial

1.1 Program Execution

To word with protocol buffers, you should install the Protobuf complier protoc first. Since the PF compiles pf codes to C++, you need to run protoc to compile the .proto files to C++ classes first. PF includes two scripts which allow you to compile and run pf program easily.

To compile and run a PF program, run the following script:

./PF.sh <file\_name>.pf

If there is any parse error then the compilation process fails and the script displays parse error.

If there is any type error then the compilation process fails and the script displays the corresponding type checking error.

If the program compiling correctly, the script displays Ocaml to c++ succeeded.

Then it will generate <file\_name>.cpp and then you can use the g++ to compile and then run the program. Here, you need to provide the when you use g++ to compiling the program.

Alternatively, you can use the following script on the test programs:

./PF\_test.sh <file\_name>.pf

Besides the errors listed above, if there is any error during running the c++ code, the script displays these errors.

If it runs successfully, it will generate <file\_name>.output files and print the result on the script directly.

1.2 Lexical Conventions

1.2.1 Comments

The character // introduce a single line comment while the character /\* introduce a multi-line comments. Comment with character /\* must terminates with the character \*/.

// This is the single line comments.

/\* This is the multi-line

comments. \*/

1.2.2 Identifiers

An identifier is a string of letters, digits and underscores. The letters are character a-z and A-Z and digits are 0-9. The identifier is case-sensitive. A valid identifier should begin with an letter or an underscore.

1.2.3 Keywords

The following identifiers are reserved by PF and cannot be re-defined by users.

|  |  |
| --- | --- |
| Keywords | Description |
| Integer, Boolean, String, Protobuf | Data types |
| true, false | Literals |
| if, then, else, while, do, end | Statement constructs |
| print, pprint, readfrom, writeto, sizeof | Built-in functions |

1.2.4 Separator

Semicolons “;” are used to terminate statements. An operation starts from the end of the previous one and will end at a semicolon.

e

1.2.5 Whitespace

A sequence of ‘ ‘, ‘\012‘, ‘\r‘, ‘\t‘, ‘\n’ does not mean anything. They are used to separate tokens. The compiler will ignore all the whitespace.

1.3 Types

PF supports the primitive types of integer, boolean and string. It is not hard to extend the types to include float and double. But here we only include integer for simplicity. The boolean type can either carry a value of true or false. It is considered as its own type and applying any boolean operators to non-boolean variables will cause an error during compilation. For instance,

if ( 3 || 4 ) // This will cause an type error.

A string is a sequence of characters with double quotes.

str := “abc” ; // str is a string type variable.

All variables have to be declared before they are used. PF does not support conventions between different types. The types should match when the variables are used.

PF also introduce a complex type, Protobuf. A Protobuf is an object of the classes generated by .proto file.

1.4 Operators

1.4.1 Mathematical Operators

All mathematical operators work on integer types only and they will result in integer types. Trying to apply mathematical operators to other types will cause errors during compilation.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| + | Addition | 1 + 2 results in 3 |
| - | Substraction | 3 - 1 results in 2 |
| \* | multiplication | 2 \* 3 results in 6 |
| / | Division | 6 / 2 results in 3 |
| % | Modulo | 7 % 2 results in 1 |

1.4.2 Comparison Operators

All comparison operators work on integer types only and they will result in boolean types. Trying to apply comparison operators to other types will cause errors during compilation.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| == | Equal | 1 == 1 results in true |
| < | Less than | 2 < 1 results in false |
| <= | Less than or equal to | 1 <= 4 results in true |
| > | Greater than | 3 > 1 results in true |
| >= | Greater than or equal to | 1 >= 1 results in true |

1.4.3 Logical Operators

All logical operators work on boolean types only and they will result in boolean types. Trying to apply logical operators to other types will cause errors during compilation.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| && | And | true && true results in true |
| || | Or | true || false results in true |
| ! | Not | ! true results in false |

1.4.4 Protobuf Operators

Protobuf operators work on protobuf types only. Trying to apply protobuf operators to other types will cause errors during compilation. These operators will simply the operations with Protobuf variables comparing to C++ using methods.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| $ | Access element | person $ name |
| [+] | Add element | person $ phones [+] |
| ? | Has element or not | person $ name ? |

The Protobuf access operator $ can be used to access each elements in a Protobuf variable. In c++, the methods to access message fields are different from those to access primitive type fields. Therefore, during the access process, we have to differentiate between message fields and other fields, which is not easy especially when there are multiple levels of nesting and may easily introduce an error.

In PF, $ is a general operator, this means when using $, we do not need to consider whether the element is a message type and how this is implemented in detail. If the element is annotated as repeated, we can use index to specify which one we are using. For example,

person $ email [3] // This results in the third email of person.

person $ phones [2] $ number // This results in the the number of the second phones of

// person.

A Protobuf variable person has a repeated message field phones and phones has a singular integer type field number. We will find it is much easier to use the element access operator $ than using methods in c++.

The Protobuf add operator works on the repeated fields only. It will add a new element to the repeated fields. If the field does not have a message type, the new value should be provides. For example,

// This will add a new email “[new\_email@ucdavis.edu](mailto:new_email@ucdavis.edu)” to person.

person $ email [+] “[new\_email@ucdavis.edu](mailto:new_email@ucdavis.edu)” ;

// This will add a new phone to person.

Person $ phones [+] ;

The operator ? works on singular fields only. It will return true if that field has been set. For example, the following statements check whether name has been set. If not, it will set “my\_name” to it.

If (person $ name ?) then

end

1.5 Statements

Statements were used to cause actions and to control flow in the program.

1.5.1 Declaration statements

Declarations statements are used to specify the type of a new identifier. For primitive types, declaration has the form

Type-specifier <Identifier>;

The following statements show how to declare variables.

Integer a ; // Declare an integer type variable a.

Boolean b ; // Declare a boolean type variable b.

String c ; // Declare a string type variable c.

To declare a Protobuf variable, the .proto file and the specified message should be provided. The declaration has the form:

Protobuf <Identifier> “<proto\_file>.proto” “<Message>”;

For instance,

Protobuf person “address.proto” “Person” ; // Declare a Protobuf variable person.

“address.proto” was shown in Fig.1 and in this declaration we use its Person message.

1.5.2 Assignment Statements

Assignment statements are used to assign a new value to a variable. The variable could be a declared integer, bool, string and Protobuf variable. We use “:=” to show the assignments and the type at both sides of “:=” should be the same. The assignment statements have the form:

<Identifier> := <expression>;

In PF, the assignment statements could also be applied to the fields in a Protobuf variable. It has the same form as above except that the left side could be an access to an element of a Protobuf variable. For example,

person $ name := “my\_name”; // This will set “my\_name” to the field name in person.

In c++, we use methods to set a field and the process of set could be hard since the message fields have mutable\_ method rather than a set\_ method. Therefore we have to clarify whether we are manipulating a message field or a primitive type field during the setting process. But in PF, the setting process is the same as the setting an expression to a variable. We do not even need to call the methods or differentiate the fields we are manipulating on.

a $ b[4] $ c $ d[2] := 1;

This assignment statement will work correctly if the Protobuf variable a has a repeated message field b and b has a singular message field c and c has a repeated integer field d. This could be much easier than using the methods in c++ programs.

1.5.3 Expression Statements

Expression statements execute the specified expression, which are the building blocks of an program. An expression can be made up of multiple expressions. For example,

3 + 5

a $ b[4] $ c $ d[2] + 2

1.5.4 Conditional Statements

Conditional statements are used to control the flow of a program. Different sets of statements will be executed depending on the boolean value of an conditional expression. In PF, two forms of conditional statements are provided:

form1: If <expression> then

statements

end

form2: If <expression> then

statements1

end else

statements2

end

The expression after if must be evaluated to be a boolean type. The last statement in the statements block should not terminate with a semicolon. And if the if statement is not the last statement in the program, a semicolon should be added to the last end.

1.5.5 While Statements

The while statements have the form:

while <expression> do

statements

end

The expression after while must be evaluated to be a boolean type. The statements in the while loop will be executed until the value of the boolean expression is false.

1.6 Scope

If a variable is declared within an if or while statement block, it can only be used within the block. Trying to access that variable outside its own scope will cause an error during compilation. Otherwise, the variable is visible across the whole program. For example,

Integer x;

x : = 1;

if x < 4 then

x := 4;

Protobuf person “address.proto” “Person”;

person $ id := 12345 // This is ok.

end;

person $ id := 12345; // This is not allowed since person is only visible within the if

//statement block.

1.7 Built-in Functions

PF provides some built-in functions to assist the manipulations with Protobuf objects.

1.7.1 readfrom

The serialized Protobuf message is stored in binary format. The readfrom function can parse a message from a given file and set the parsing result to the Protobuf variable. The form to use this function is:

<Identifier> readfrom “<message\_file>”

The variable should be declared before and the provided message file should be serialized using the same .proto file. Here is an example,

Protobuf person “address.proto” “Person”; // Declare a Protobuf variable person.

/\* Parsing the message from file “person\_information” and then set the result to

variable person.\*/

person readfrom “information”;

1.7.2 writeto

This writeto function serializes a Protobuf variable and then stores the binary bytes in the provided file. The form to use this function is:

<Identifier> writeto “<message\_file>”

For example,

person writeto “information\_modified”;

Since the bytes after serializing are binary, they are not human readable.

1.7.3 print

Print function prints its argument to standard output. It has the form:

print <expression>

1.7.4 pprint

The pprint applies on the Protobuf variables only. It generates a human-readable representation of the message in string type. The form to use this function is:

pprint <Identifier>

For example,

str := pprint person; // str is a string of human-readable representation of person.

2. Example PF programs