Runtime Serialization of Higher-Order Functions

Guoye Zhang

December 2017

1 Introduction

Normally when we talk about serialization, we are referring to serializing objects and structures with basic data types such as integers and strings in JSON, XML, or binary formats. Our framework aims to provide a generalized solution to serialize higher-order functions of existing programming languages, and we present several use cases that demonstrate the benefits such as type-safety and extensibility of our approach.

We generalize higher-order function as a sequence of function calls and optionally a return value. To serialize higher-order functions, we first need to establish a shared symbol table between encoder and decoder. Then our framework generates stub symbols with stub types with the encoder, and calls to stub symbols are recorded in thread-local storage and put into protobuf message.

In the decoder side, our framework generates a typechecker and an interpreter for the protobuf message. Typechecking is performed during decoding, which checks if the function is valid given its signature. Interpreter runs the decoded function when it is called, and uses the actual implementations of the symbols provided to the decoder.

Both encoder and decoder are generated by our framework from a user-provided specification file. Given this specification file, our framework produces (1) a protobuf message definition, and (2) encoders and decoders in target programming language.

2 Background

Many new programming languages and new versions of existing languages have made higher-order functions first-class citizen which behaves just like any other values. However, a significant part missing from programming languages is the ability to

serialize and inspect functions. Our framework aims to add this capability to every language that supports both protobuf and higher-order functions.

2.1 Protobuf

Protocol Buffers [1] are a language-neutral, platform-neutral extensible mechanism for serializing structured data. The structure of data is described in a proto file which is then fed into a compiler to generate source code in target programming language.

Our framework is built upon protobuf and utilizes the same code generation technique. Instead of data structure, our specification file describes types of symbols and functions to serialize. It is then compiled into both proto file and source code in target programming language. Protobuf is our underlying serialization format.

3 Serialization Format

Our serialization format is based on protobuf. It is a sequence of function calls and raw values followed by an optional return step number. Two examples are given below for a quick introduction.

3.1 Basic Arithmetic Example

The following is an example of a serialized function pretty-printed from its protobuf message:

```
%0:0 = (int32)2
%0:1 = add($0:0, %0:0)
%0:2 = add($0:1, $0:2)
%0:3 = subtract(%0:1, %0:2)
return %0:3
```

This is encoded from function:

```
let num: Int32 = 2
func f(a: Int32Producer, b: Int32Producer, c: Int32Producer)
       -> Int32Producer {
    return subtract(
        add(a, num),
        add(b, c)
    )
}
```

Let's first look at the serialized format. There are two types of values, one starting with `$` is an argument and one starting with `%` is a result. 3 arguments to `f` are `$0:0`, `$0:1`, and `$0:2`. Each step in `f` stores its result in `%0:x` where `x` is the step number. At the end, the result of step 3 (`%0:3`) is returned. The number before the colon represents the level of the value which is useful for capturing, and the next example will focus on this level.

For function `f`, types of arguments and return value are all `Int32Producer` instead of `Int32`. `Int32Producer` is a stub type generated by our framework that works with stub symbols. It is something that produces an `Int32`, and `Int32` itself is an `Int32Producer` (shown by `num` whose value is passed into add symbol) but not vice versa. Calls to `add` and `subtract` are instrumented during the encoding of `f`.

3.2 Nested Functions Example

In this example, nested functions with value capturing are encoded. `ifThenElse(Bool, () -> Int32, () -> Int32) -> Int32` is a user-provided symbol that performs the if operation. It calls the "then" clause or the "else" clause based on whether the first argument is true or false, and forwards the return value.

```
func addOrSubtract(lhs: Int32Producer, rhs: Int32Producer)
                    -> Int32Producer {
    let performAdd = randomBool()
    return ifThenElse(
        performAdd,
        { return add(lhs, rhs) },
        { return subtract(lhs, rhs) }
    )
}
```

This function is serialized into:

```
%0:0 = randomBool()
%0:1 = {
    %1:0 = add($0:0, $0:1)
    return %1:0
}
%0:2 = {
    %1:0 = subtract($0:0, $0:1)
    return %1:0
}
```

```
%0:3 = ifThenElse(%0:0, %0:1, %0:2)
return %0:3
```

As can be seen on the example above, "then" clause and "else" clause both capture the arguments of `addOrSubtract`.

3.3 Protobuf Format

In our protobuf message, we store a sequence of steps and a return step number. Each step can either be a raw value, a call to a higher-order function (not shown on examples above), or a call to a symbol. It is represented in an `oneof` type in protobuf. Since protobuf uses numbers to label each case, we use odd numbers for raw values or higher-order function calls, and even numbers for symbol calls, so that adding declaration to specification file does not break backward compatibility.

Symbols and higher-order functions can take arguments, and arguments are represented in two numbers: level number and step number. Level number indicates the level where the value is captured. Non-negative step number locates the result of a step (`%x:x`) while negative step number locates an argument (`$x:x`).

4 Specification File

Similar to proto file which describes the structure of protobuf messages, our specification file describes the type of functions to serialize and defines a set of shared symbols. It consists of 5 sections: Import, Type, Subtype, Symbol, and Function (mandatory). Types and subtypes should be declared before their usage. And each section can appear multiple times.

Here is the specification file used in examples above:

```
# Example 1
Type:
int32

Symbol:
add(int32, int32) -> int32
subtract(int32, int32) -> int32

Function:
f3: (int32, int32, int32) -> int32
```

```
# Example 2 Additions
Type:
bool
f = () -> int32

Symbol:
randomBool() -> bool
ifThenElse(bool, f, f) -> int32

Function:
f2: (int32, int32) -> int32
```

4.1 Types

We support 4 categories of types as argument types or function return types.

1. Basic types: Basic protobuf types such as `int32` and `bool`.
2. Custom protobuf types: Imported custom protobuf message types.
3. Non-backing types: Types without protobuf representation that can only be produced and consumed by symbols.
4. Higher-order function types: Named function types used in argument or return type.

	Imported	Declared in Type	Used in Subtype
Basic types		✓	
Custom protobuf types	✓	✓	
Non-backing types			✓
Higher-order function types		✓	

4.2 Function Signature

Function signatures are used in higher-order function types, symbol types, and serializable function types. It is a C-style function with 0 or more argument types and optionally 1 return type which is supported by all target programming languages.

Function signature takes the form of `(type, type, ...) -> type` or `(type, type, ...)`. Each type is one of 4 categories described above. It is worth noting that in order to take or return a higher-order function, the higher-order function type needs to be named beforehand in the Type section.

4.3 Sections

1. Import: List of proto files that would be imported into generated proto file. These are used for importing custom protobuf messages definitions.

2. Type: List of protobuf-backed types and higher-order function types. Protobuf-backed types can be either basic types or imported custom message types. Higher-order function types are represented by a name, equal sign, and its function signature, for example `binaryop = (int32, int32) -> int32`.

3. Subtype: List of subtyping relationships between non-backing types. It only affects generated stub types and typechecker, and it is independent as to how subtyping is implemented in the decoder side, which can be subclassing, interface/protocol implementation, or prototype-based inheritance.

4. Symbol: List of symbols provided to the decoder. It is represented by a name followed by its function signature, for example `add(int32, int32) -> int32`. Stub symbols are generated for symbols declared here.

5. Function: List of serializable functions. It is represented by a name, colon, and its function signature, for example `f: (int32) -> int32`. Encoder and decoder are generated for each function type declared here.

4.4 Backward Compatibility

Backward compatibility of serialized data is maintained when declarations are added to the bottom while existing declaration are kept in the original order. This is achieved by the even/odd scheme devised in our project (See Serialization Format section above).

It is recommended that new declarations being added along with new sections. On the specification file above, "Example 2 Additions" are added to the end of "Example 1" which guaranteed that existing example 1 serializations are still compatible to the decoder generated for the new specification.

5 Implementation

Provided a specification file, our framework generates stub types, stub symbols, typechecker, and interpreter.

5.1 Encoder

Stub types and stub symbols are both part of the encoder. All stub types are represented in two numbers, level and step. The reason we generate a stub type for each concrete type is to take advantage of typechecker in target programming language. Subtyping relationships are also reflected in stub types. Stub symbols are global functions that record its arguments in thread-local storage, and return its level and step number in stub type.

```
func addOrSubtract(lhs: Int32Producer, rhs: Int32Producer)
                    -> Int32Producer {
    let performAdd = randomBool()
    return ifThenElse(
        performAdd,
        { return add(lhs, rhs) },
        { return subtract(lhs, rhs) }
    )
}
```

Let's look at the `addOrSubtract` example again. Encoder starts by creating a runtime and passing `level: 0, step: -1` (`$0:0`) and `level: 0, step: -2` (`$0:1`) as arguments to this function. When `randomBool` is called, it finds the runtime in thread-local storage and records the call, and then returns `level: 0, step: 0` (`%0:0`) which is stored in `performAdd`. When `ifThenElse` is called, it also records the call and all 3 arguments. Since some of its arguments are functions, encoders for those types of functions are called recursively with the level being incremented to 1. Finally, the result of `ifThenElse` is passed out of `addOrSubtract` and recorded by the encoder.

Pseudocode of an encoder:

```
encode(f) {
    env = createEncoderRuntime()  // env is thread-local var
    let function = createFunction()
    env.pushFunction(function)

// Call "f" with stub arguments and record the return step
    function.returnStep = f($0:0, $0:1)

    env = nil
    return function
}
```

Pseudocode of a stub symbol with two arguments:

```
symbol(a, b) {
    let f = env.topFunction  // env is thread-local var

    if (a is rawValue) {
// Based on its type, "a" is a protobuf value or a function
//  - Protobuf value is stored directly as a step
//  - Function is encoded recursively then stored as a step
        f.pushStep(encodeRaw(a))

// Convert "a" to a producer
        a = %(env.currentLevel):(f.lastStepNumber)
    }    // Otherwise "a" is already a producer

// Do the same for "b"
    if (b is rawValue) {
        f.pushStep(encodeRaw(b))
        b = %(env.currentLevel):(f.lastStepNumber)
    }

    f.pushStep(symbol(a, b))
    return %(env.currentLevel):(f.lastStepNumber)
}
```

5.2 Decoder

Decoder consists of a typechecker and an interpreter. Typechecker inspects the protobuf message during decoding to check if it is properly typed, which ensures no type error can occur in the interpreter. Interpreter runs the function when it is called with real arguments. Both typechecker and interpreter keep a stack of runtimes which contain arguments and results of each step. Typechecker uses types while interpreter uses real values.

Pseudocode of an interpreter:

```
// function, arguments, symbol implementation
run(f, args, imp) {
    var results = []    // Empty list
```

```
    for step in f.steps {
        match (step) {
        rawValue(v) =>    // can be data or function
            results.append(v)

        higherOrderedCall(stubFunc, stubArgs) =>

// findValueFor() locates the value from args or results
            let function = findValueFor(stubFunc)

// Locate all arguments
            let realArgs = stubArgs.map(findValueFor)

// Recursively interpret inner function
            let result = run(function, realArgs, imp)

            results.append(result)

        symbolCall(symName, stubArgs) =>
            let realArgs = stubArgs.map(findValueFor)
            let result = imp.call(symName, realArgs)
            results.append(result)
        }
    }
    return findValueFor(f.returnStep)
}
```

6 Optimization

2 optimizations we applied in our framework are described below.

6.1 Inline Unit Functions

Unit function is a function that passes all arguments to another function, and passes the return value back. If an argument is a higher-order function, encoder provides it as a unit function. For example, when we are encoding `(binaryop) -> int32`, `binaryop` in our stub symbol needs to be an actual function that can be called, so

a unit function which calls the argument is created. However, suppose we have a symbol that takes `binaryop` as an argument, it needs to be able to encode arbitrary functions created by users, so it calls an encoder internally to encode it. But what if we pass the argument directly to this symbol? Unit function is encoded and stored as an independent function which is unnecessary.

An optimization we implemented is to inspect every encoded function, and if it is a unit function, the body is discarded and the wrapped inner function is used instead.

6.2 Result Destruction

In our format, we can use the result of any step before us as the argument to the current step. However, we do not want to keep all results forever in memory. This optimization is to record the last place where every result is used during typechecking, so when the decoded function is called, we can destroy results as early as possible.

7 Use Cases

We listed 3 use cases below that demonstrate the expressibility of our approach.

7.1 Query

It is intuitive to write queries like `persons.filter { person in person.lastName == "Zhang" && person.age >= 20 }` for a local array. However for remote databases, we are required to write query in languages like SQL which provides no type-safety while being generated from a general-purpose language. A type-safe querying interface built with our framework is introduced below.

```
Import:
db.proto     # Provides columnID

Type:
int32
string
columnID

Symbol:
getStringColumn(columnID) -> StringColumn
```

```
getIntColumn(columnID) -> IntColumn
lessThanInt(IntColumn, int32) -> Predicate
equalToInt(IntColumn, int32) -> Predicate
equalToString(StringColumn, string) -> Predicate
and(Predicate, Predicate) -> Predicate
or(Predicate, Predicate) -> Predicate
not(Predicate) -> Predicate

Function:
query: () -> Predicate
```

With the specification above, we can further wrap the C-style API with the help of operator overloading to achieve the similar interface of a local query, providing a type-safe alternative to SQL.

7.2 Drawing

Graphics formats such as PDF and SVG are complex textual file that is hard to create or parse. Here is an example of a vector format specification for our framework that is easy to extend:

```
Type:
double

Subtype:
Rect <: Shape
Circle <: Shape

Symbol:
createRect(double, double) -> Rect     # width, height
createCircle(double) -> Circle         # radius
setShapePosX(Shape, double)
setShapePosY(Shape, double)
setShapeColor(Shape, double, double, double)
draw(Shape)

Function:
drawInRegion: (double, double)
```

This example uses subtyping to provide an object-oriented interface to a graphics

format. To draw a circle:

```
drawInRegionEncode { x, y in
    let circle = createCircle(5.0)
    setShapePosX(circle, 5.0)
    setShapePosY(circle, 5.0)
    draw(circle)
}
```

7.3 JIT

Just-in-time compilation has been gaining popularity recently to facilitate faster execution of dynamic languages. However, JIT compilers typically have a complex interface, such as Java and .NET bytecode, or LLVM C++ API to build an intermediate representation [2], which makes it hard for a trivial program to take advantage of JIT compilation. This example introduces the concept of multi-stage programming [3] and imagines new ways of building JIT engine API.

Computation heavy program might have hot-spots where some parts of them are executed many times, suppose this fragment is the hot-spot:

```
for _ in 0..<n {
    if someCondition {
        performA()
    } else {
        performB()
    }
}
```

`n` is a small number that does not change after initialization, and `someCondition` does not change, either. JIT would reduce the overhead by eliminating those branches.

The solution is very simple in our framework. First, we create symbols for `performA` and `performB` (`performAsym` and `performBsym`) assuming JIT knows how to call the original function. Then, we can serialize the fragment by encoding:

```
let fragment = fragmentEncode {
    for i in 0..<n {
        if someCondition {
            performAsym()
        } else {
            performBsym()
```

```
        }
    }
}
let code = JIT.compile(fragment)
code()
```

Suppose `n` is 3 and `someCondition` is true, the encoded function would only be 3 calls to `performAsym()`, and the JIT engine would be able to compile this program without any branches. This works because our framework encodes the function by calling it and recording the invocations of stub symbols. In this case, `for` and `if` are both executed during encoding, so in effect, we used loop and condition to build this program.

JIT is only one form of multi-stage programming, while other forms include OpenGL Shading Language, OpenCL kernel, etc. Our goal is to provide a better way to generate programs within programs that can replace these domain-specific languages.

8 Limitation

Two limitations of our approach are described below, and multiple workarounds are purposed.

8.1 Side-Effects

Since we can only instrument calls to symbol and the return value, operations performed outside symbol calls which produce side-effects cannot be recreated. The following is an example of an incorrect use of `ifThenElse`:

```
func f(b: BoolProducer) -> Int32Producer {
    var t: Int32Producer = 0 as Int32
    ifThenElse(
        b,
        { t = 1 as Int32 },
        { t = 2 as Int32 }
    )
    return t
}
```

This kind of side-effect produces unexpected result (3 in this case), or even corrupts the serialization output in some cases.

Two ways to fix this:

8.1.1 Change code style

Change `ifThenElse` to return an `Int32`, and rewrite the code as:

```
func f(b: BoolProducer) -> Int32Producer {
    return ifThenElse(
        b,
        { return 1 as Int32 },
        { return 2 as Int32 }
    )
}
```

8.1.2 Create a wrapper

```
Symbols:
int32CreateWrapper() -> Int32Wrapper
int32Set(Int32Wrapper, int32)
int32Get(Int32Wrapper) -> int32

func f(b: BoolProducer) -> Int32Producer {
    let t = int32CreateWrapper()
    ifThenElse(
        b,
        { int32Set(t, 1 as Int32) },
        { int32Set(t, 2 as Int32) }
    )
    return int32Get(t)
}
```

8.2 Recursion

Our encoded format itself supports recursion. However, it is impossible to directly encode recursive functions since there is no way to terminate during encoding. One solution is to add helper functions for creating recursive functions. For example, if we need a recursive `ii = (int32) -> int32`, we can create a helper function `iiRecHelper(function: (ii, int32) -> int32) -> ii`, which provides an additional argument of type `ii` that can be used to express recursion.

This solution can be implemented manually in symbols, or generated by our framework automatically as a future feature. Recursion is not always wanted by API authors though since user can abuse it to cause non-termination, so we should require explicit marking to function signatures in the specification file in order to enable recursive helpers.

9 Future Work

This section describes a few ideas for the future development of our framework.

9.1 Generics

Generics can be used to create more powerful symbols that work with any type. For example, `ifThenElse` can be typed as `ifThenElse<T>(bool, () -> T, () -> T) -> T` to support every type. However, generics implementation varies drastically from language to language, and a thorough study is needed for a cross-language generics design.

9.2 Standard Library

Basic operations such as arithmetic, comparison, and type conversion for basic protobuf type, and control flow expressions such as `ifThenElse` and loops can be provided in a built-in standard library. We can even provide a built-in list type with element-wise operations like `map`, `reduce`, `filter`, etc.

This will simplify the work for API authors and make our framework useful even without any custom symbols. We will also have the opportunity to adapt the standard library API to fit into each target programming language natively.

Generics are essential to standard library, but it is possible to use generics inside standard library without exposing them to users which is the approach taken by Go. [4]

9.3 More Languages

Currently, our framework is developed in Swift with Swift as the only target programming language for code generation. However, our approach does not rely on any specific features in Swift as it can be extended to support all protobuf languages with higher-order functions. We are planning to support C++, Java, Rust and more in the future.

10 Conclusion

In this project, we developed a framework that serializes higher-order functions in runtime. It was based on protobuf with support for existing programming languages. We presented several use cases that demonstrated the benefit of type-safety and simplicity of our approach, and offered several examples to showcase the extensibility of our framework. We hope to provide an alternative to query languages like SQL, graphics formats like SVG, and general-purpose bytecode, and become the last stop to a type-safe world.

References

[1] Google. Protocol buffers. `https://developers.google.com/protocol-buffers/`.

[2] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[3] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[4] Google. The go programming language. `https://golang.org`.