CREATING THE BOLT COMPILER: PART 7

A Protobuf tutorial for OCaml and C++

OCTOBER 03, 2020 8 MIN READ

SERIES: CREATING THE BOLT COMPILER

Part 1: How I wrote my own "proper" programming language
Part 2: So how do you structure a compiler project?
Part 3: Writing a Lexer and Parser using OCamllex and Menhir
Part 4: An accessible introduction to type theory and implementing a type-checker
Part 5: A tutorial on liveness and alias dataflow analysis
Part 6: Desugaring - taking our high-level language and simplifying it!
Part 7: A Protobuf tutorial for OCaml and C++
Part 8: A Complete Guide to LLVM for Programming Language Creators
Part 9: Implementing Concurrency and our Runtime Library
Part 10: Generics - adding polymorphism to Bolt
Part 11: Adding Inheritance and Method Overriding to Our Language

---

Now that we've desugared our language into a simple "Bolt IR" that is close to LLVM IR, we want to generate LLVM IR. One problem though, our Bolt IR is an

TABLE OF CONTENTS

OCaml value, but the LLVM API we're using is the native C++ API. We can't import the value directly into the C++ compiler backend, so we need to serialise it first into a language-independent data representation.

> Hey there! If you came across this tutorial from Google and don't care about compilers, don't worry! This tutorial doesn't really involve anything compiler-specific (it's just the illustrative example). If you care about OCaml then the first half will be up your street, and if you're here for C++ you can skip the OCaml section.

Protocol Buffers Schema

Protocol buffers (aka *protobuf*) encodes your data as a series of binary messages according to a given schema. This schema (written in a `.proto` file) captures the structure of your data.

Each message contains a number of fields.

Fields are of the form `modifer type name = someIndex`

Each of the fields have a modifier: `required / optional / repeated`. The `repeated` modifier corresponds to a list/vector of that field. e.g. `repeated PhoneNumber` represents a value of type `List<PhoneNumber>`.

For example, the following schema would define a Person in a contact book. Each person has to have a name and id, and optionally could have an email address. They might have multiple phone numbers (hence `repeated PhoneNumber`), e.g. for their home, their mobile and work.

example.proto

Copy

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
```

```
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}
```

Note here within the schema for a `Person` message, we also **number** the fields (`=1`, `=2`, `=3`, `=4`). This allows us to add fields later without altering the existing ordering of fields (so you can continue to parse these fields without having to know about new fields).

Every time we want to add a new "type" of field, we define a schema (just like how you might define a class to add a new type of object). We can even define a schema within another schema, e.g. within the schema for a `Person`, we defined the schema for the `PhoneNumber` field.

We can also define the schema for an **enum** type `PhoneType`. This enum acted as a "tag" for our phone number.

Now, just as you might want a field to be one of many options (specified by the enum type), you might want the message to contain exactly one of many fields.

For example in our compiler we might want to encode an identifier as *one of two options*. Here we don't want to store data in an index field if we've tagged it as a `Variable`.

frontend_ir.mli

Copy

```
type identifier =
    | Variable of string
    | ObjField of string * index (* name and field index *)
```

In our protobuf, we can add fields for each of these options, and use the keyword `oneof` to specify that only one of the fields in that group will be set at once.

Now, whilst this tells Protobuf that one of those values is set, we have no way of telling which one of them is set. So we can introduce a `tag` enum field, and query its value (`Variable` or `ObjField`) when deserialising the object.

Copy

```
message identifier {
  enum _tag {
    Variable_tag = 1;
    ObjField_tag = 2;
  }

  message _ObjField {
    required string objName = 1;
    required int64 fieldIndex = 2;
  }

  required _tag tag = 1;
  oneof value {
    string Variable = 2;
    _ObjField ObjField = 3;
  }
}
```

Note since the constructor `Variable of string` has only one argument of type `string`, we can just set the type of that field to `string`. We equally could have defined a message `_Variable` with schema:

Copy

```
message _Variable {
    required string varName = 1;
  }
  ...
  oneof value {
    _Variable Variable = 2;
    _ObjField ObjField = 3;
  }
```

Mapping OCaml Type definitions to Protobuf Schema

Now we've looked at the `identifier` schema, let's flesh out the rest of the frontend "Bolt IR" schema.

If you notice, whenever we have an OCaml variant type like `identifier` we have a straightforward formula to specify the corresponding schema. We:

- Add a tag field to specify the constructor the value has.
- Define a message schema if a constructor has multiple arguments e.g for `ObjField`. If the constructor has only one argument (`Variable`) we don't need to.
- Specify a `oneof` block containing the fields for each of the constructors.

We could manually write out all of the schema, but we'd have to rewrite these every time our language changed. However we have a hidden weapon up our sleeve - a library that will do this for us!

> There's the full Protobuf Schema Guide if you want to learn more about other message types.

PPX Deriving Protobuf

OCaml allows libraries to extend its syntax using a ppx hook. The ppx libraries can preprocess the files using these hooks to extend the language functionality. So we can tag our files with other information, such as which types need to have a protobuf schema generated.

We can update our Dune build file to preprocess our `ir_gen` library with the ppx-deriving protobuf library:

dune

Copy

```
(library
 (name ir_gen)
 (public_name ir_gen)
 (libraries core fmt desugaring ast)
 (flags
  (:standard -w -39))
 (preprocess
  (pps ppx_deriving_protobuf bisect_ppx --conditional))
```

Note the flags command suppresses warning 39 - "unused `rec` flag" - since the code the ppx_deriving_protobuf library generates raises a lot of those warnings. I would highly recommend that you also suppress these warnings!

> An aside, `bisect_ppx` is the tool we use to calculate test coverage. It has a `--conditional` flag since we don't want to preprocess the file with coverage info if we're not computing the test coverage.

Telling PPX deriving protobuf that we want to serialise a type definition is easy - we just stick a `[@@deriving protobuf]` at the end of our type definition. For variant types, we have to specify a `[@key 1]`, `[@key 2]` for each of the variants.

For example, for our `identifier` type definition in our `.mli` file:

frontend_ir.mli

Copy

```
type identifier =
  | Variable of string [@key 1]
  | ObjField of string * int [@key 2]
  [@@deriving protobuf]
```

We do the same thing in the `.ml` file, except we also specify the file to which we want to write the protobuf schema.

frontend_ir.ml

Copy

```
type identifier =
  | Variable of string [@key 1]
  | ObjField of string * int [@key 2]
[@@deriving protobuf {protoc= "../../frontend_ir.proto"}]
```

This path is relative to the src file `frontend_ir.ml`. So since this `frontend_ir.ml` file is in the `src/frontend/ir_gen/` folder of the repo, the protobuf schema file will be written to `src/frontend_ir.proto`. If you don't specify a file path to the `protoc` argument, then the `.proto` file will be written to the `_build` folder.

One extra tip, the ppx deriving protobuf library won't serialise lists of lists. For example you can't have:

Copy

```
type something = Foo of foo list list [@key 1] [@@deriving prot
```

This is because if we have a message schema for `foo`, then to get a field of type `foo list` is straightforward - we use `repeated foo` in our field schema. But we can't say `repeated repeated foo` to get a list of a list of type foo. So to get around this, you'd have to define another type here:

Copy

```
type list_of_foo = foo list [@@deriving protobuf]
(*note no @key since not a variant type *)

type something = Foo of list_of_foo list [@key 1] [@@deriving p
```

Finally, to serialise our Bolt IR using this schema, PPX deriving protobuf provides us with `<type_name>_to_protobuf` serialisation functions. We use this to get a binary protobuf message that we write to an output channel (`out_chan`) as a sequence of bytes.

ir_gen_protobuf.ml

Copy

```
let ir_gen_protobuf program out_chan =
    let protobuf_message = Protobuf.Encoder.encode_exn program_
    output_bytes out_chan protobuf_message
```

In our overall Bolt compiler pipeline, I write the output to a `.ir` file if provided, or to `stdout`:

compile_program_ir.ml

Copy

```
...
 match compile_out_file with
    | Some file_name ->
        Out_channel.with_file file_name ~f:(fun file_oc ->
            ir_gen_protobuf program file_oc)
```

```
    | None          -> ir_gen_protobuf program stdout )
```

Auto-generated Protobuf Schema

The README of the repository for the PPX Deriving Library has a thorough explanation of the mapping. We've already gone through an example of the autogenerated schema, so the schema shouldn't be too unfamiliar. There was a slight modification I made though. For (`ObjField of string * int`) I said the generated output was:

Copy

```
message _ObjField {
    required string objName = 1;
    required int64 fieldIndex = 2;
  }
```

This is unfortunately not quite true. I put in the objName and fieldIndex for clarity, but the library doesn't know what the semantic meaning of `string * int` is, so instead it looks like:

Copy

```
message _ObjField {
    required string _0 = 1;
    required int64 _1 = 2;
  }
```

That's the downside of the autogenerated schema. You get the generic field names `_0` , `_1` , `_2` and so on instead of `objName` and `fieldIndex` . And for `type block_expr = expr list` , you get the field name `_` :

Copy

```
message block_expr {
  repeated expr _ = 1;
}
```

Decoding Protobuf in C++

Alright, so far we've looked at Protobuf schema definitions and how PPX Deriving Protobuf encodes messages and generates the schema. Now we need to decode it using C++. As with the encoding section, we don't need to know the details of how Protobuf represents our messages in binary.

Generating Protobuf Deserialisation Files

The `protoc` compiler automatically generates classes and function definitions from the `.proto` file: this is exposed in the `.pb.h` header file. For `frontend_ir.proto`, the corresponding header file is `frontend_ir.pb.h`.

For Bolt, we're building the C++ compiler backend with the build system Bazel. Rather than manually running the `protoc` compiler to get our `.pb.h` header file and then linking in all the other generated files, we can instead take advantage of Bazel's integration with `protoc` and get Bazel to run `protoc` during the build process for us and link it in.

The Bazel build system works with many languages e.g. Java, Dart, Python, not just C++. So we specify two libraries - a language-independent `proto_library`, and then a specific C++ library (`cc_proto`) that wraps around that library:

BUILD

Copy

```
load("@rules_cc//cc:defs.bzl", "cc_library")

# Convention:
# A cc_proto_library that wraps a proto_library named foo_proto
# should be called foo_cc_proto.
cc_proto_library(
    name = "frontend_ir_cc_proto",
    deps = [":frontend_ir_proto"],
    visibility = ["//src/llvm-backend/deserialise_ir:__pkg__"],

)

# Conventions:
# 1. One proto_library rule per .proto file.
# 2. A file named foo.proto will be in a rule named foo_proto.
proto_library(
    name = "frontend_ir_proto",
    srcs = ["frontend_ir.proto"],
```

```
)
```

We include the `cc_proto_library(...)` as a build dependency to whatever files that need to use the `.pb.h` file, and Bazel will compile the protobuf file for us. In our case, this is our `deserialise_ir` library.

BUILD

Copy

```
load("@rules_cc//cc:defs.bzl", "cc_library")

cc_library(
    name = "deserialise_ir",
    srcs =  glob(["*.cc"]),
    hdrs = glob(["*.h"]),
    visibility = ["//src/llvm-backend:__pkg__", "//src/llvm-bac
    deps = ["//src:frontend_ir_cc_proto", "@llvm"]
)
```

Deserialising a Protobuf serialised file

The `frontend_ir.pb.h` file defines a namespace `Frontend_ir`, with each message mapping to a class. So for our Bolt program, represented by the `program` message in our `frontend_ir.proto` file, the corresponding Protobuf class is `Frontend_ir::program`.

To deserialise a message of a given type, we create an object of the corresponding class. We then call the `ParseFromIstream` method which deserialises the message from the input stream and stores it in the object. This method returns a boolean value indicating success/failure. We've defined our own custom `DeserialiseProtobufException` to handle failure:

deserialise_protobuf.cc

Copy

```
Frontend_ir::program deserialiseProtobufFile(std::string &fileP
  Frontend_ir::program program;
  std::fstream fileIn(filePath, std::ios::in | std::ios::binary
  ...
  if (!program.ParseFromIstream(&fileIn)) {
```

```
      throw DeserialiseProtobufException("Protobuf not deserialis
    }
    return program;
  }
```

So we're done!

Well, one caveat... this autogenerated class is a dumb data placeholder. We need to read the data out from the `program` object.

Reading out Protobuf data from a protobuf class

If you try to read the protoc-generated file `frontend_ir.pb.h`, you'll realise it's a garguantuan monstrosity, which is not nearly as nice as the standard example in the official tutorial. So instead of trying to read the methods from the file, here is an explanation of the structure of the header file.

> TIP: Make sure you have code-completion set-up in your IDE - it means you won't need to manually search through `frontend_ir.pb.h` for the right methods.

We'll be deserialising messages using the schema defined below:

frontend_ir.proto

Copy

```
package Frontend_ir;

message expr {
  enum _tag {
    Integer_tag = 1;
    Boolean_tag = 2;
    Identifier_tag = 3;
    ...
    IfElse_tag = 11;
    WhileLoop_tag = 12;
    Block_tag = 15;
  }
  message _Identifier {
    required identifier _0 = 1;
    optional lock_type _1 = 2;
```

```
  }
  ...
  message _IfElse {
    required expr _0 = 1;
    required block_expr _1 = 2;
    required block_expr _2 = 3;
  }
  message _WhileLoop {
    required expr _0 = 1;
    required block_expr _1 = 2;
  }
  ...
  required _tag tag = 1;
  oneof value {
    int64 Integer = 2;
    bool Boolean = 3;
    _Identifier Identifier = 4;
    ...
    _IfElse IfElse = 12;
    _WhileLoop WhileLoop = 13;
    block_expr Block = 16;
    ...
  }
}

message block_expr {
  repeated expr _ = 1;
}
```

Protobuf message classes and enums

Each Protobuf message definition maps to a class definition. Protobuf enums map to C++ enums. To give each class/enum a globally unique name, they are prepended by the package name (`Frontend_ir`) and any classes they're nested in.

So message `expr` corresponds to class `Frontend_ir_expr` , and enum `_tag` which is nested within message `expr` maps to `Frontend_ir_expr__tag` .

In the repo, the `bin_op` message also has a nested `_tag` enum, and this maps to `Frontend_ir_bin_op__tag` (note how this namespacing means it doesn't clash with the `_tag` definition in the `expr` message).

This holds for arbitrary levels of nesting, e.g. the nested message `_IfElse` in the `expr` message maps to the class `Frontend_ir_expr__IfElse`.

Specific enum values for the enum `_tag` e.g. `Integer_tag` can be referred to as `Frontend_ir_expr__tag_Integer_tag`.

Rather than writing classes\enums by concatenating the package and class names with `_`, Protobuf also has a nested class type alias, so you can write these nested message classes as `Frontend_ir::expr` and `Frontend_ir::expr::_IfElse`.

Accessing Specific Protobuf Fields

Each protobuf *required* field `field_name` has a corresponding accessor method `field_name()` (where the field name is converted to lower-case). So the field `tag` in the `expr` message would map to the method `tag()` in the `Frontend_ir::expr` class, and the field `Integer` would map to the method `integer()`, `IfElse` to `ifelse()` etc.

> NB: to reiterate, don't get confused between the field name e.g `IfElse` and the type of the field `_IfElse` in the Protobuf message (note the prepended underscore). This is only because the OCaml PPX deriving protobuf library gave them those names - we could have chosen less confusing names if we were writing this proto schema manually.

For *optional* fields, we can access the fields in the same way, but we also have a `has_field_name()` boolean function to check if a field is there.

For *repeated* fields, we instead have a `field_name_size()` function to query the number of items, and we can access item i using the `field_name(i)` method. So for a field name `_1` the corresponding methods are `_1_size()` and `_1(i)`. For field name `_` in the autogenerated schema, the methods would correspondingly be `__size()` and `_(i)`.

Sanitising our Frontend IR

Let's put this in practice by sanitising our protobuf classes into C++ classes directly mapping our OCaml type definitions. Each of the OCaml `expr` variants maps to a subclass of an abstract `ExprIR` class.

Copy

```
struct ExprIR {
  virtual ~ExprIR() = default;
```

```
  ...
};

struct ExprIfElseIR : public ExprIR {
  std::unique_ptr<ExprIR> condExpr;
  std::vector<std::unique_ptr<ExprIR>> thenBlock;
  std::vector<std::unique_ptr<ExprIR>> elseBlock;
  ExprIfElseIR(const Frontend_ir::expr::_IfElse &expr);
  ...
};
struct ExprWhileLoopIR : public ExprIR {
  std::unique_ptr<ExprIR> condExpr;
  std::vector<std::unique_ptr<ExprIR>> loopExpr;
  ExprWhileLoopIR(const Frontend_ir::expr::_WhileLoop &expr);
  ...
};
...
```

> I use `std::unique_ptr` all over the compiler backend to avoid explicitly managing memory. You can use standard pointers too - this is just a personal preference!

Remember how we had a specific `tag` field to distinguish between variants of the `expr` type. We have a `switch` statement on the value of the `tag` field. This `tag` tells us which of the `oneof{}` fields is set. We then access the correspondingly set field e.g. `expr.ifelse()` for the `IfElse_tag` case:

expr_ir.cc

Copy

```
std::unique_ptr<ExprIR> deserialiseExpr(const Frontend_ir::expr
  switch (expr.tag()) {
    case Frontend_ir::expr__tag_IfElse_tag:
      return std::unique_ptr<ExprIR>(new ExprIfElseIR(expr.ifel
    case Frontend_ir::expr__tag_WhileLoop_tag:
      return std::unique_ptr<ExprIR>(new ExprWhileLoopIR(expr.w
    ...
  }
}
```

Looking at the message schema for the `_IfElse` and `block_expr` messages:

frontend_ir.proto

Copy

```
message _IfElse {
    required expr _0 = 1;
    required block_expr _1 = 2;
    required block_expr _2 = 3;
  }

  message block_expr {
  repeated expr _ = 1;
}
```

Our constructor reads each of the `_IfElse` message's fields in turn. We can then use our `deserialiseExpr` to directly deserialise the `_0` field. However, because the message `block_expr` has a repeated field `_`, we have to iterate through the list of expr messages in that field in a for loop:

expr_ir.cc

Copy

```
ExprIfElseIR::ExprIfElseIR(const Frontend_ir::expr::_IfElse &ex
  Frontend_ir::expr condMsg = expr._0();
  Frontend_ir::block_expr thenBlockMsg = expr._1();
  Frontend_ir::block_expr elseBlockMsg = expr._2();

  condExpr = deserialiseExpr(condMsg);
  for (int i = 0; i < thenBlockMsg.__size(); i++) {
    thenExpr.push_back(deserialiseExpr(thenBlockMsg._(i)));
  }
  for (int i = 0; i < elseBlockMsg.__size(); i++) {
    elseExpr.push_back(deserialiseExpr(elseBlockMsg._(i)));
  }
}
```

The rest of the deserialisation code for the Bolt schema follows the same pattern.

Wrap up

In this post we've converted from our OCaml frontend IR to the equivalent C++ classes. We'll use these C++ classes to generate LLVM IR code in the next part of the tutorial.

Share This On Twitter

If you liked this post, please consider sharing it with your network. If you have any questions, tweet away and I'll answer :) I also tweet when new posts drop!

PS: I also share helpful tips and links as I'm learning - so you get them well before they make their way into a post!

Post Follow @mukulrathi_

SERIES: CREATING THE BOLT COMPILER

Part 1: How I wrote my own "proper" programming language
Part 2: So how do you structure a compiler project?
Part 3: Writing a Lexer and Parser using OCamllex and Menhir
Part 4: An accessible introduction to type theory and implementing a type-checker
Part 5: A tutorial on liveness and alias dataflow analysis
Part 6: Desugaring - taking our high-level language and simplifying it!
Part 7: A Protobuf tutorial for OCaml and C++
Part 8: A Complete Guide to LLVM for Programming Language Creators
Part 9: Implementing Concurrency and our Runtime Library
Part 10: Generics - adding polymorphism to Bolt
Part 11: Adding Inheritance and Method Overriding to Our Language

← Desugaring - taking our high-level language and simplifying it!

17 Tips to Convert Your Internship to a Full-time Offer! →