Edelweiss: Decentralized Protocol Compiler

Milestone 1 (MVP): RPC compiler for Go

Petar Maymounkov

petar@protocol.ai

Roadmap

- **Milestone 1 (Q1 2022, MVP)**: RPC compiler for Go **(THIS TALK)**
- **Milestone 2 (Q2 2022)**: Feature parity with IPLD schema, performance squared, policies and transforms
- **Milestone 3 (Q3 2022)**: Lambdas across networks, Filecoin/FVM actors API
- **Milestone X**: Multiple target languages, packaging, github integration, doc generation, cli generation, type interoperability checks at compile time, etc.

Plan

- Definitions
- Types (semantics, representations, generated runtime code)
- Interoperability
- Usage

Definitions

AST interface

User builds type definition AST in Go.

Syntax to come later, when the language matures.

```
import "github.com/ipld/edelweiss/defs"

Types{
   Named{ Name: "MyLink", Type: Link{To: Int{}} }, // link to int
   Named{ Name: "MyList", Type: List{Element: Any{}} }, // list of any
   ...
}
```

Named definitions

Wrap a type definition in `Named{}`

```
Named{
    Name: "MyStructure"
    Type: Structure{
        Fields: Fields{
            Field{ Name: "Foo", Type: Int{} },
            Field{ Name: "Bar", Type: Any{} },
        }
    }
}
```

Inline definitions

```
Named{
   Name: "MyStructure"
   Type: Structure{
      Fields: Fields{
         Field{
            Name: "MyFieldFoo",
            Type: List{Element: Int{}}, // <-- inline type definition, list of int
         },
      }
   }
}
```

Inline types are named generically, e.g. `AnonListXXX`

Named inline definitions

```
Named{
    Name: "MyStructure"
    Type: Structure{
        Fields: Fields{
            Field{
                Name: "MyFieldFoo",
                Type: Named { // <-- named inline type definition, list of int
                    Name: "MyInlineListOfInt",
                    Type: List{Element: Int{}},
                },
            },
        }
    }
}
```

Type references

Use `Ref{Name: "TypeName"}` to refer to any `Named` type

```
Named{
      Name: "MyList",
      Type: List{ Element: Int{} }
}

Named{
      Name: "MyListOfList",
      Type: List{ Element: Ref{Name: "MyList"} }
}
```

Types

Significance of types

1. Semantics of data (agnostic to programming language)
2. Representation of data in IPLD Data Model (encoding/decoding)

   Note: *"Transforms" (introduced later) can alter representation.*
3. Representation of data in user's programming language

Types

- **Non-parametric**
  - **Builtin:** Bool, Float, Int, Byte, *Char*, *String*, *Bytes*
  - **Special:** Any, Nothing
- **Parametric**
  - **Composite:** Link, List, Map, Structure, *Inductive*, *Singleton*, *Union*
  - **Functional:** *Function*, *Service*, *Method*

Italicized types are new or different from IPLD Schema types.

Non-parametric types

Builtin types

Definitions:

```
Bool{} // represented as IPLD bool
Float{} // represented as IPLD float
Int{} // represented as IPLD int
Byte{} // represented as IPLD int
Char{}
String{}
Bytes{}
```

Runtime implementations in package `github.com/ipld/edelweiss/values`:

```
type Byte byte
// etc.
```

Char

Semantically:

- a character is not an integer

Representationally:

- encoded as an IPLD integer which is a valid UTF8

Programmatically:

- Implemented by `type Char rune` in package `edelweiss/values`

String

Semantically:

- `String{}` is equivalent to `List{Element: Char{}}`

Representationally:

- Encodes to IPLD string
- Decodes from a UTF8 IPLD string or the IPLD encoding of `List{Element: Char{}}`

Programmatically:

- Implemented by `type String string` in package `edelweiss/values`

Bytes

Semantically:

- `Bytes{}` is equivalent to `List{Element: Byte{}}`

Representationally:

- Encodes to IPLD bytes
- Decodes from IPLD bytes or the IPLD encoding of `List{Element: Byte{}}`

Programmatically:

- Implemented by `type Bytes []byte` in package `edelweiss/values`

Special types

Nothing

Semantically:

- `Nothing{}` holds no value

Representationally:

- Encodes as IPLD nothing

Programmatically:

- Implemented by `type Nothing struct{}`

E.g. use in conjunction with `Inductive` types to describe enumerations.
E.g. use in conjunction with `Union` types to describe optional values.

Any

Semantically:

- `Any{}` can hold any IPLD value
- IPLD kinds are in one-to-one mapping with types in this type sytem:
  - IPLD bool, int, float, string, bytes map to `Bool{}`, `Int{}`, `Float{}`, `String{}`, `Bytes{}`
  - IPLD link maps to `Link{To: Any{}}`
  - IPLD list maps to `List{To: Any{}}`
  - IPLD map maps to `Map{Key: Any{}, Value: Any{}}`
  - IPLD nothing maps to `Nothing{}`

Programmatically:

- Implemented by `type Any struct{ Value }` where `Value` is an interface

Parametric types

Link

Semantically:

- `Link{To: TYPE_DEF_OR_REF}`

Representationally:

- Encodes as IPLD link

Programmatically:

- Code-generated Go `struct` which holds a `Cid`

Use `Link{To: Any{}}` when the link target is of unknown type.

List

Semantically:

- `List{Element: TYPE_DEF_OR_REF}`

Representationally:

- Encodes as IPLD list

Programmatically:

- Code-generated Go alias for a slice type

Map

Semantically:

- `Map{Key: TYPE_DEF_OR_REF, Value: TYPE_DEF_OR_REF}`

Representationally:

- Encodes as IPLD list of key/value pairs or an IPLD map, if the key is a string

Programmatically:

- Code-generated Go slice of key/value pairs or a Go map, if the key is a string

Structure

Semantically:

- A list of named and typed fields, written as

```
Structure{
	Fields: Fields{
		Field{Name: "NAME", Type: TYPE_DEF_OR_REF},
		...
	}
}
```

Representationally:

- Encodes as IPLD map

Programmatically:

- Code-generated Go `struct`

Singletons

Semantically:

- A builtin value that always equals a given constant, written as

```
SingletonBool{BOOL_VALUE}
SingletonInt{INT_VALUE}
SingletonByte{BYTE_VALUE}
SingletonChar{CHAR_VALUE}
SingletonFloat{FLOAT_VALUE}
SingletonString{STRING_VALUE}
```

Representationally:

- Encoded as the correspoding IPLD kind

Programmatically:

- Code-generated as an empty Go `struct`

Inductive

Semantically:

- One of a list of name/value pairs *distinguished by their name*, written as

```
Inductive{
      Cases: Cases{
            Case{Name: "NAME", Type: TYPE_DEF_OR_REF},
            ...
      }
}
```

Representationally:

- Encoded as an IPLD map, wrapping the case name and its value

Programmatically:

- Code-generated as a Go `struct` with one pointer field per case

"Inductive" types correspond to IPLD Schema "union" types.

Union

Semantically:

- One of a list of name/value pairs *distinguished by their value*, written as

```
Union{
	Cases: Cases{
		Case{Name: "NAME", Type: TYPE_DEF_OR_REF},
		...
	}
}
```

Representationally:

- Encoded as the value of the active case
- The union itself has *no representational footprint*

Programmatically:

- Code-generated as a Go `struct` with one pointer field per case

Inductive ≠ Union

Note that inductive and union types are fundamentally different:

- Both types constitute cases that have a name and a value
- Inductive cases are distinguished by their names
- Union cases are distinguished by their values

Enumeration = Union + Singleton

Traditional enumerations over any primitive type can be expressed as a union of singletons:

```
Union{
      Cases: Cases{
            Case{Name: "Case1", Value: SingletonInt{1}}
            Case{Name: "Case2", Value: SingletonInt{2}}
            ...
      }
}
```

String-valued enumeration = Inductive + Nothing

Traditional enumerations over strings can also be expressed as an inductive type with nothing values:

```
Inductive{
	Cases: Cases{
		Case{Name: "Case1", Value: Nothing{}}
		Case{Name: "Case2", Value: Nothing{}}
		...
	}
}
```

Services

Service type

- A *service* is a collection of *methods*
- Each *method* is uniquely named and associated with a *functional signature*
- A functional signature specifies the types of the argument and a return values

Service definition

```
Named{
	Name: "MyService"
	Service{
		Methods: Methods{
			Method{
				Name: "MyMethod",
				Type: Fn{
					Arg: TYPE_DEF_OR_REF,
					Return: TYPE_DEF_OR_REF,
				},
			},
			...
		},
	},
}
```

Generated RPC code

- The compiler supports multiple RPC code-generation backends
- Currently, we have a DAGJSON-over-HTTP backend
  - Single URL endpoint per service
  - Method and arguments captured in the DAGJSON body of an HTTP GET request

Cross-version and -capability interoperability

Problem

- Protocols always evolve; never in a finished state
- It is hard to predict the direction of evolution of a protocol
- This causes over-thinking, over-engineering and paralisis in earlier version designs

Solution

- Enable backwards-compatible growth from any state and in any part of a protocol

Structures can grow

A server expecting

```
Structure{
        Fields: Fields{
                Field{ Name: "Foo", Type: Int{} },
        },
}
```

Will accept requests from a client sending

```
Structure{
        Fields: Fields{
                Field{ Name: "Foo", Type: Int{} },
                Field{ Name: "Bar", Type: String{} },
        },
}
```

Structures can shrink

A server expecting

```
Structure{
      Fields: Fields{
            Field{ Name: "Bar", Type: Union{
                        Cases: Cases{
                              Case{ Name: "Missing", Type: Nothing{} },
                              Case{ Name: "String", Type: String{} },
                        },
                  },
            },
      },
}
```

Will accept requests from a client sending

```
Structure{ Fields: Fields{} }
```

This feature is slated for Milestone 2.

Introducing alternatives where there weren't (1/2)

Suppose V1 of a type definition is:

```
Structure{
    Fields: Fields{
        Field{
            Name: "Foo",
            Type: Int{},
        },
    },
}
```

Introducing alternatives where there weren't (2/2)

The next iteration, V2, of the protocol can substitute any given type with a union over old and new alternatives:

```
Structure{
     Fields: Fields{
          Field{
               Name: "Foo",
               Type: Union{   // int is substituted by union of int or float
                    Cases: Cases{
                         Case{ Name: "MyInt", Type: Int{} },
                         Case{ Name: "MyFloat", Type: Float{} },
                    }
               },
          },
     },
}
```

Excess, deficit and unexpected data

At a receiver:

- Data which is in excess of the schema can be captured generically as IPLD data. This applies to: structure fields and union/inductive cases.
- Data which is missing can be captured, by instructing the code-generator to treat the entire schema as optional (at every level of the schema hierarchy)
- Data which contradicts the expected types can also be captured generically as IPLD data. This applies to: structure fields and union/inductive cases.

These features are planned for Milestone 2, based on use case urgency.

Usage

Compiling and code generation

See a complete example in github.com/ipld/edelweiss/examples

Compile type definitions to a Go source file generation plan:

```
x := &GoPkgCodegen{
     GoPkgDirPath: "/home/petar/src/foo/bar", // local directory for generated code
     GoPkgPath:    "github.com/petar/foo/bar", // go package name of generated code
     Defs:         Types{ ... }, // type definitions
}
goFile, err := x.Compile()
```

Materialize the Go file to disk:

```
err = goFile.Build()
```