Overview of Edelweiss
A decentralized protocol compiler

Petar Maymounkov petar@protocol.ai

What is a protocol compiler?

- **Define protocols formally**
  - Verify properties (e.g. different versions are interoperable)
- **Generate implementations**
  - Data serialization
  - RPC services

Challenges of decentralized development

- **Brokering across multiple data models**
  - *IPLD, Protobuf, FlatBuffers, XML, JSON, MessagePack, Apache Avro/Arrow/Parquet, etc.*
  - *read from Git or Bittorrent, write IPLD to IPFS/Filecoin*
- **Implementations in different languages**
  - *Go, JavaScript/TypeScript, Rust, Python, etc.*
- **Variety of RPC networking stacks**
  - *JSON-over-HTTP, CBOR-over-libp2p, Protobuf, GRPC, etc.*
- **New abstractions**
  - *Smart contracts exchange callbacks over network boundaries*

Solution

An extensible, modular, code-generating compiler with a unified type system:

- **Unified type system**

  Strict and expressive type system, decoupled from serialization technology

  Facilitate bridging the same schema from one data model into another

- **Code-generating**

  De/serialization and RPC code is generated for multiple target languages

- **Modular**

  Custom code-generation backends for different (a) serialization formats, (b) RPC networking stacks and (c) target languages

- **Extensible**

  New types can be added as needed

User workflow

1. Define data and services types (aka, the user's schema)
2. Pick a backend
   - programming language (Go)
   - serialization (IPLD)
   - networking stack (DAGJSON-over-HTTP)
3. Generate code:
   - data types + encoders/decoders
   - services client/server

Significance of types

Universal type properties

Semantics of data and services (agnostic to programming language)

- What values can be held
- Which types can be used in place of others

Backend-specific type properties

- **Wire representation** of data (e.g. in the IPLD Data Model)
- **Programmatic representation** of data in the target programming language

Types

Non-parametric

- **Builtin:** Bool, Float, Int, Byte, Char, String, Bytes
- **Special:** Any, Nothing

Parametric

- **Composite:** Link, List, Map, Structure, Tuple, Inductive
- **Functional:** *Function*, *Service*, *Method*
- ***Combinatorial: Singleton, Union***

Future candidates

- Int128, UInt256, Float64, …

User defines types using AST

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: Structure{
            Fields: Fields{
                Field{ Name: "Foo", Type: List{Element: Char{}} }, // list of char
                Field{ Name: "_bar", GoName: "Bar", Type: Ref{Name: "MyLink"} },
            }
        },
    },
}
```

Documentation of types

- **AST rules for defining types**
  https://github.com/ipld/edelweiss/blob/main/doc/manual-milestone1-slides/manual-milestone1-slides.md
- **Type representations in the IPLD data model:**
  https://github.com/ipld/edelweiss/blob/main/doc/representations.md
- **All documentation**
  https://github.com/ipld/edelweiss/blob/main/doc

Structure

Semantically:

- A list of named and typed fields, written as

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

Representationally:

- Encodes as IPLD map

Programmatically:

- Code-generated Go `struct`. Field values are embedded (non-pointers).

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`

Union

Semantically:

- One of a list of possible types, written as

```
Union{
      Cases: Cases{
            Case{GoName: "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

Protocols evolve gracefully with Unions

```
Structure{
   Fields: Fields{
      Field{ Name: "ContentKey", Type: Bytes{} },
      Field{ Name: "Provider", Type: Ref{Name: "Peer"} },
   }
},
```

```
Structure{
   Fields: Fields{
      Field{ Name: "ContentKey", Type: Bytes{} },
      Field{ Name: "Provider", Type: Union{
            Cases: Cases{
               Case{Name: "Peer", Type: Ref{Name: "Peer"} }
               Case{Name: "Miner", Type: Ref{Name: "Miner"} }
            }
         }
      },
   }
},
```

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}}
            ...
      }
}
```

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,
				},
			},
			...
		},
	},
}
```

Demonstration

An end-to-end service example

https://github.com/ipld/edelweiss/tree/main/examples/greeting-service

Reframe API: A real-life production service

- Reframe API spec: https://github.com/ipfs/specs/blob/master/REFRAME.md
- Reframe implementation: https://github.com/ipfs/go-delegated-routing

Writing custom backends

The compiler pipeline:

1. Parse user's type definitions in AST form
2. Resolve all type references (check for dangling refs, name clashes, etc.)
3. Prepare Go names for types and methods (map Edelweiss namespaces to Go)
4. **Invoke code-generation templates**

Issues with naive code templates

The naive approach, using string-based templates:

```
package {{.MyPkgName}} // <-- problem
import "fmt" // <-- problem

func {{.MyFuncName}} (arg {{.MyArgTypeName}}) error {
    ...
    {{.IncludeAnotherCodeTemplate}} // <-- problem
    ...
    return fmt.Errorf("error msg") // <-- problem
}
```

Problems:

- Packaging is hard-wired in templates
- Templates are not composable (outer template does not know inner's needed imports)
- Hard to keep track of imported packages and their aliases and name clashes

Language-aware code templates

Go language-aware template:

```
func {{.MyFuncName}} (arg {{.MyArgType}}) error {
    ...
    {{.IncludeAnotherCodeTemplate}}
    ...
    return {{.FmtErrorf}}("error msg")
}
```

Template data:

```
{
    MyFuncName: "MyFunc",
    MyArgType: GoTypeRef{Pkg: "github.com/ipld/ipld", Name: "Node"},
    FmtErrorf: GoRef{Pkg: "fmt", Name: "Errorf"},
    IncludeAnotherCodeTemplate: ...,
}
```

The way forward

There's a lot we didn't cover:

- Performance, no-reflection, zero-copy
- Automated protocol interoperability checks (aka, type safety)
- Type transformations
- Best-effort parsing
- Areas of application (middleboxes, smart contracts, command-line clients, protocol bridges, etc.)