Edelweiss

A decentralized protocol compiler

github.com/ipld/edelweiss

Petar Maymounkov

LabWeek 2022

History blitz

Decentralized data structures require a new data model.

IPLD = decentralized JSON = content-addressed links + JSON

Data model is **agnostic** to application abstraction (above), and serialization (below).

Protocol Stack	**Application level**	File formats (UnixFS, MFS, Filecoin storage, etc) Service Interfaces (Reframe, StoreTheIndex, Filecoin actors, etc)
	Data model	IPLD
	Serialization and RPC	JSON, CBOR, Protocol Buffers, XML, Apache Avro, etc HTTP, libp2p, etc

Application developers need

- Define higher-level **data abstractions**
  - E.g. file system, content provider, signed peer record

- Define **service interfaces**
  - E.g. Reframe API, StoreTheIndex API, Filecoin multisig actor API
  - E.g. IPFS command-line interface

- Get **language bindings**
  - Go, JavaScript, Rust at minimum
  - Python, Lean, Julia as well

System developers need

- To use context-specific **serialization, networking and RPC backends** e.g.
  - Synchronous JSON-over-HTTP (Reframe)
  - Asynchronous JSON-over-HTTP (FVM)
  - CBOR-over-HTTP (Reframe)
  - CBOR-over-libp2p (IPFS?)

Prior work

Application **data abstractions** modelled using schema definition languages
(Protobuf, XML Schema, JSON Schema)

- **Drawbacks:**
  - Awkward for uncoordinated decentralized work (e.g. protobuf field indices)
  - Serialization-specific features (don't apply to IPLD)

Service interfaces modelled using interface definition languages
(Protobuf, Apache Thrift)

- **Drawbacks:**
  - Lack of decentralized abstractions like **links** and **lambdas** (e.g. FVM actors and EVM smart contracts pass **callbacks**)
  - Hard to change the backend (network stack and call semantics)

Our approach

Data and service definition language

- Higher data and service abstractions (e.g. file system state and API) modelled as types
- Types are composed from a standard set of generic building blocks (maps, lists, structures, unions, lambdas, links, singletons, etc) from programming
- Types have IPLD representations (encode to and decode from IPLD)

Bindings to programming languages

- Compiler generates code
- Simple, language-aware framework for defining code generation templates (novel)

Developer tools

- Compiler can check if two user types (e.g. file format, service API) are interoperable
- Compiler can generate type descriptors (not implemented yet)
- Support for generic, dependent types (e.g. “link to a file system of CAR files”)

Basic types

Primitive: Bool, Float, Int, Byte, Char, String, Bytes, Int128, UInt256, Float64, ...

Special: Any, Nothing

Composite: Link, List, Map, Structure, Tuple, Inductive/Enum *(Lean, OCaml, Rust)*

Functional: Lambda, Service, Method

Predicate: Singleton, Union *(Lean, OCaml, Julia, Rust)*

Decentralization and interoperability features

SemVer is all-or-nothing. Inappropriate for Web3.

Interoperability type checking

- Developers can check whether two different protocols (e.g. an old client and a new server) can work together from their type descriptors
  - Fine- and coarse-grain (e.g. method, data, and service)
  - At compile-time and run-time

Schema definition language can express all possible IPLD tree patterns (novel)

- Singleton type decodes by exact match (e.g. used for magic numbers in protocols)
- Union type decodes one of a few of alternatives

Schemas for parsing IPLD

Web3 developers need to write schemas for existing IPLD protocols.

Use types as schemas for parsing desired IPLD tree patterns.

Can user-defined types describe all possible IPLD tree patterns?

This depends on the set of basic types. Existing schema languages lack:

- Type for exact matching (singleton), eg
  "this field must have the value 3"
- Type for sequential alternatives (union), eg
  "this field must either parse as a PeerRecord or else as an IPNSRecord"

Same needs when extending current protocols in backwards compatible ways.

State of affairs

Current users

- Reframe API, IPFS, DHT Hydra Booster, StoreTheIndex, SomeGuy

What's behind us

- Generation of high-performance data type serialization in Go
- Generation of asynchronous JSON/HTTP and CBOR/HTTP backends in Go

What's ahead of us

- Source syntax parsing
- Data and network backends for TypeScript/JavaScript
- Prototype a backend for FVM actors

We are hiring a compiler engineer!

Reach out to petar@protocol.ai