RGB smart contract advancements

Maxim Orlovsky

LNP/BP Standards Association &
Pandora Core AG

Implementation progress & updates

Core lib progress over last 2 weeks

- Secondary issuance & burn procedures for RGB20 assets
- NFT progress
- Improving and extending general API to work with state transitions etc
- Disclosures: completed
- Data containers API
- Virtualization advancements

RGB20 progress (fungible assets)

- Secondary issuance
- Burn & replacement epochs and procedures

RGB asset information is a special consignment containing only non-transfer types of state transitions (issue, burn, replacement, renomination)

- Subschema for simplified RGB20 asset without replacement procedure

Virtualization of RGB validation rules

Client-side validation

- Internal data consistency & completeness
- Changes in state

RGB client-side validation

Only restricts bitcoin script-based rules, not extends them.
i.e. you can spend bitcoin output, but get invalid RGB state (loose asset, for instance by trying to inflate its supply)

- **Bitcoin script** controls who owns (who can change the state)
- **RGB schema rules** controls _how_ state _may_ be changed to continue being valid state

RGB Schema

- Defines structure of client-side-validated data (state)
- Defines rules of how the state can change
  - Conservative: relations between state types and rights
    like “asset issuance right can issue assets, asset ownership right can only transfer them”
  - Dynamic: more complex equations which state values must fulfill
    like “sum of inputs must be equal to the sum of outputs”

RGB client-side-validation

Applies to *consignments* and (in restricted form) to *disclosures*

- **Conservative (rule-based)**: VM is not required, since it is
  “once written, never changed” and applies to all future schemata and forms of RGB contracts
  - Validation of internal node graph consistency
    (presence of all referenced contract graph nodes)
  - Validation against schema structure
  - Validation against single-use-seal commitment medium – prevention of double-spending
    (blockchain, state channel transaction graph structure)
- **Variative (script-based)**: was planned to be implemented with Simplicity VM,
  but currently written as embedded rust procedures (not in form of VM)
  - RGB20: validation of inflation, issuance volumes
  - RGB21,22 (NFT, identity): validation of 1-to-1 NFT transfer rules for operations combining multiple NFTs/identities

RGB schema is not a script,
but is a **data structure**

may (or may not) contain script extensions
(for dynamic part of the validation)

RGB Schema today

- **Serialized** in binary *strict-encoded format*
- **Interpreted**/verified by RGB Core Library code
- If the schema contains scripts, than a **VM** or **embedded procedures** can be used
  - Today only embedded procedures are available, hardcoded as a part of RGB Core Library
  - Tomorrow a scripting support will be added to RGB

RGB Schema: rust notation

- Just the data structure in form of dictionaries and arrays

```
Schema {
    rgb_features: none!(),
    root_id: none!(),
    genesis: GenesisSchema {
        metadata: type_map! {
            FieldType::Ticker => Once,
            FieldType::Name => Once,
            FieldType::RicardianContract => NoneOrOnce,
            FieldType::Precision => Once,
            FieldType::Timestamp => Once,
            // We need this field in order to be able to verify pedersen
            // commitments
            FieldType::IssuedSupply => Once
        },
        owned_rights: type_map! {
            OwnedRightsType::Inflation => NoneOrMore,
            OwnedRightsType::OpenEpoch => NoneOrOnce,
            OwnedRightsType::Assets => NoneOrMore,
            OwnedRightsType::Renomination => NoneOrOnce
        },
        public_rights: none!(),
        abi: none!(),
    },
    extensions: none!(),
    transitions: type_map! {
        TransitionType::Issue => TransitionSchema {
            metadata: type_map! {
                // We need this field in order to be able to verify pedersen
                // commitments
                FieldType::IssuedSupply => Once
            },
            closes: type_map! {
                OwnedRightsType::Inflation => OnceOrMore
            },
            owned_rights: type_map! {
                OwnedRightsType::Inflation => NoneOrMore,
                OwnedRightsType::OpenEpoch => NoneOrOnce,
                OwnedRightsType::Assets => NoneOrMore
            },
            public_rights: none!(),
            abi: bmap! {
                // sum(in(inflation)) >= sum(out(inflation), out(assets))
                TransitionAction::Validate => script::EmbeddedProcedure::FungibleIssue as script::EntryPoint
            }
        },
```

RGB Schema:
YAML representation

- For debugging purposes
- Can be produced with command-line
  `$ rgb schema convert`
  after installing with
  `$ cargo install rgb-core –all-features`
- Not used anywhere

```yaml
genesis:
  metadata:
    0: once
    1: once
    2: noneOrOnce
    3: once
    4: once
    160: once
  owned_rights:
    1: noneOrOnce
    160: noneOrMore
    161: noneOrMore
    170: noneOrOnce
  public_rights: []
  abi: {}
extensions: {}
transitions:
  0:
    metadata: {}
    closes:
      161: onceOrMore
    owned_rights:
      161: noneOrMore
    public_rights: []
    abi: {}
  16:
    metadata:
      0: noneOrOnce
      1: noneOrOnce
      2: noneOrOnce
      3: noneOrOnce
    closes:
      1: once
    owned_rights:
      1: noneOrOnce
    public_rights: []
    abi: {}
```

RGB Schema: Bech32 encoding

- For data exchange purposes

- Bech32-encoded
  zip-compressed
  strict-encoded
  schema data


```
schema1qxz4pkcdsvcqc0zzq22tu5p8vzz8uaue3ef82ymg
llsg03cstkn3hfywr53zf2rsjjrhmwdmcnvge89xecgyzg6
j6rtvdg8ygfvhd7eualhg49tc23qrd5tue4mzhdg7u5qx8m
vghqsrz9ttjwjdjtklr7820vepwk9q56jfq34yy9l9prxxj
h9dz55gr26dgdc3du6e7ddg2vecwd3rttcdg3xj9ef4vf0k
wfrhqs4csrl0swvuhhrecgj24cpvhd47dx3hfhzcfx5ncel
59crkymu2yfm53nmzcdasen5zmk4sqlvey6t0rrlcutdj7g
l47jrudxdtlmttlx7gtv7uxh605pcw79337
```

RGB Schema: Contractum Language

github.com/rgb-org/contractum-lang

- Just a convenient way of defining new schema requiring no rust coding knowledge
- Impossible to define invalid/internally inconsistent schema
- Not a script language!
  This is data definition language, like HTML, XML, YAML etc
- May contain scripts for dynamic validation
  (once VM for RGB will be completed)

```
final contract RGB20 implements FungibleAsset {
    field ticker: str
    field name: str
    field contractText: str?
    field issuedSupply: value
    field precision: uint8
    field created: timestamp

    assigns inflationRight: value*
    assigns assetsOwnership: value+ {
        validate() {
            diff = parent[..].sum() - self[..].sum()
            if diff != 0 {
                error(failures.FungibleAsset.Inflation(diff))
            }
        }
    }
    assigns renominationRight: decl?
    assigns burnEpochOpening: decl?

    fn validateSupply() {
        if self.issuedSupply != self.assetsOwnership[..].sum() {
            error(failures.FungibleAsset.IncorrectIssue())
        }
    }

    validate() {
        validateSupply()
    }

    transition AssetTransfer fulfills assetsOwnership+ {
        assigns assetsOwnership: value+
    }
```

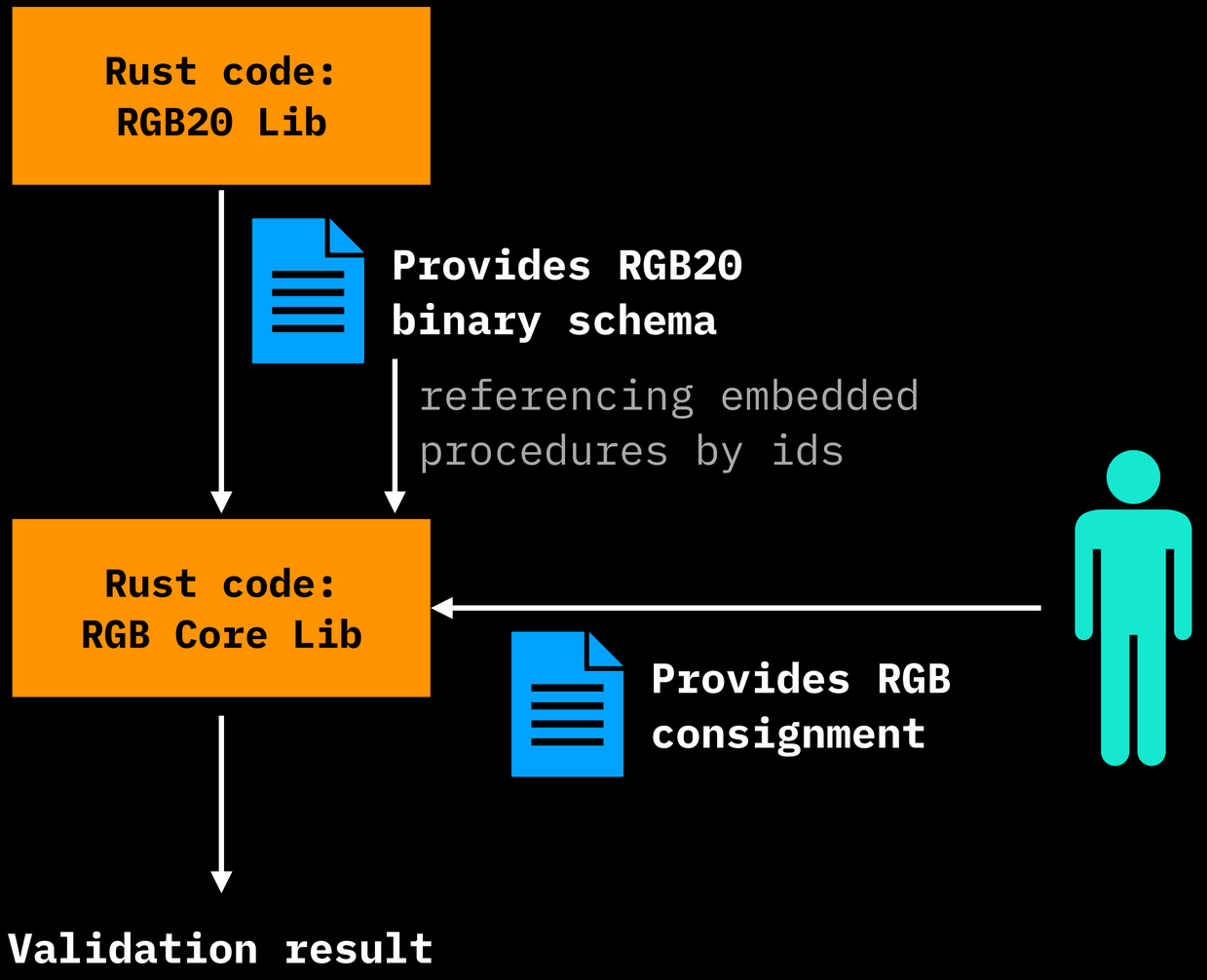
Virtualization of dynamic validations

Virtualization helps:

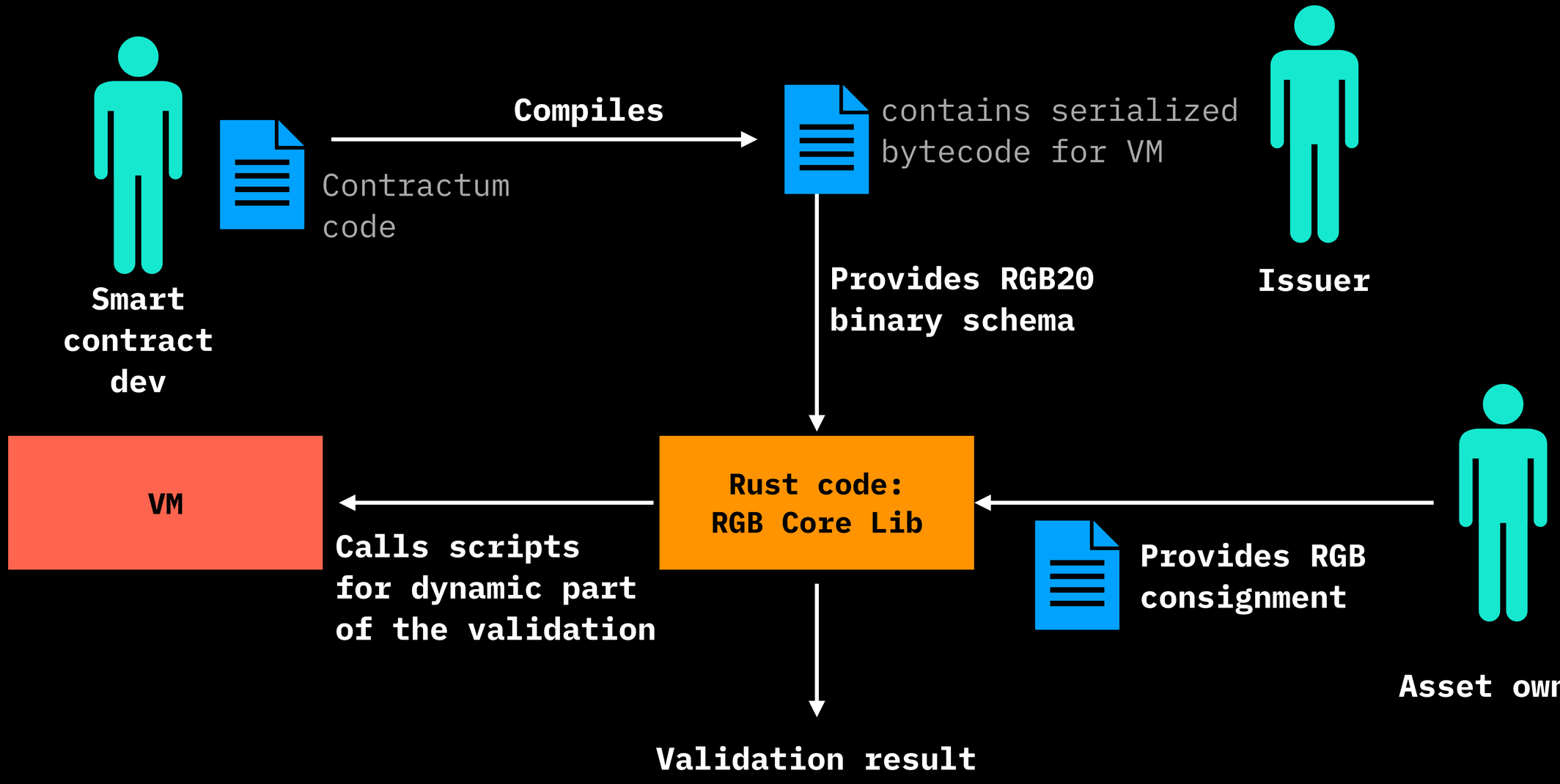
- distinguish dynamic and conservative parts of validation
- isolate state of the contract from other logic of RGB Core library
- may allow creation of custom schemata in a permissionless way by external devs

How RGB is validated today

- Third parties can do custom schemata
- But these schemata may use only existing embedded procedures from the RGB Core Lib, so their functionality is limited



How RGBv1 will be validating



RGBv1 VM & independent devs roadmap

- Current version of RGB Core Lib and **RGBv1** in its first release
  - will not use VM
  - will utilize embedded procedures
- Later in 2021/early 2021, RGB Core Lib will be updated with **VM**
  - this will not break client-side-validation since the lib already recognizes VM codes and has place to store bytecode
- Independently **Contractum** language will be completed, simplifying creation of custom/complex smart contracts by independent devs
- This does not affect wallet devs and those integrating existing RGB contracts into their software

Which VM to use for RGB?

- **Simplicity:**
  - not VM :(
  - not ready :(
  - no toolchain for devs :(
  - way too complex to understand how to program
- **WASM:** (and actually the same applies to **LLVM**, **JVM**, **CLR** + they have even more negative sides)
  - requires a lot of customization for blockchain/client-side-validation applications
  - the only version is from Parity Labs, doing Polkadot, infamous with creating contracts multiple times hacked
- **IELE** or **kEVM**:
  - EVM on drugs :(
  - a lot of custom unneeded functionality for non-bitcoin type of cryptocurrency
- do something custom?

AluVM - from “arithmetic logic unit”

github.com/internet2-org/aluvm-spec

- Purely functional & arithmetical: each operation is an arithmetic function
- No external state; converts set of inputs into false/true validation result
- Extremely robust & deterministic: no exceptions are possible
  - no stack (register-based VM)
  - no random memory access
  - no I/O, memory allocations
- If it compiles, it will always run successfully
- Easy to be implemented in hardware, like in FPGAs

Virtualization selection criteria

	Embedded procedures (*status quo*)	AluVM	WASM	Simplicity
Code audibility	Poor	Average	Good	Excellent (formally provable)
Safety	Endianess is tricky	Good	Good	Excellent
Cryptographic primitives	Grin-based, hard migration	Grin-based, smooth migration	Not known, but smooth migration	Not ready
VM Implementation effort	None	Small	High	Extreme high
Schema validation implementation effort	Medium	Low	High	Extreme high
Can be used by other schema devs	No	Yes	Easily	Not really

Scoring

	Embedded procedures (*status quo*)	AluVM	WASM	Simplicity
Security	0	6	4	8
Implementation complexity	3	2	1	0
Extensibility	7	6	4	0
Overall	**10**	**14**	**9**	**8**