Rell Documentation

ChromaWay AB

May 13, 2019

Contents

1 Introduction **1**
- 1.1 Rell SDK . . . 1
- 1.2 Chromia . . . 1
- 1.3 Rell language . . . 2

2 Quick start **3**
- 2.1 Install Docker . . . 3
- 2.2 Install the IDE . . . 3
- 2.3 Try the IDE . . . 4
- 2.4 Hello World! . . . 5
- 2.5 Videos . . . 7

3 Tutorial **9**
- 3.1 Language overview . . . 9
- 3.2 Blockchain programming . . . 9
- 3.3 Class definitions . . . 9
- 3.4 Operations . . . 11
  - 3.4.1 Relational operator basics . . . 11
  - 3.4.2 Simple operation . . . 12
  - 3.4.3 Function . . . 13
- 3.5 Query . . . 13
- 3.6 Relational expressions . . . 14
  - 3.6.1 Composite indices . . . 15
- 3.7 Examples and further exercises . . . 16

4 General Language Features **17**
- 4.1 Types . . . 17
  - 4.1.1 Simple types: . . . 17
  - 4.1.2 Complex types: . . . 17
  - 4.1.3 Nullable type . . . 18
  - 4.1.4 Tuple type . . . 18
  - 4.1.5 Collection types . . . 19
  - 4.1.6 GTXValue . . . 19
  - 4.1.7 Subtypes . . . 19
- 4.2 Module definitions . . . 20
  - 4.2.1 Class . . . 20
  - 4.2.2 Object . . . 21

4.2.3 Record . . . 22
4.2.4 Enum . . . 22
4.2.5 Query . . . 23
4.2.6 Operation . . . 23
4.2.7 Function . . . 23
4.3 Expressions . . . 24
4.3.1 Values . . . 24
4.3.2 Operators . . . 25
4.4 Statements . . . 26
4.4.1 Local variable declaration . . . 26
4.4.2 Basic statements . . . 26
4.4.3 If statement . . . 27
4.4.4 Loop statements . . . 27
4.5 Miscellaneous . . . 28
4.5.1 Comments . . . 28

5 Database Operations **29**
5.1 At-Operator . . . 29
5.1.1 Cardinality . . . 29
5.1.2 From-part . . . 29
5.1.3 Where-part . . . 30
5.1.4 What-part . . . 30
5.1.5 Tail part . . . 31
5.1.6 Result type . . . 31
5.1.7 Nested At-Operators . . . 31
5.2 Create Statement . . . 31
5.3 Update Statement . . . 32
5.4 Delete Statement . . . 32

6 Library **35**
6.1 System classes . . . 35
6.1.1 chain_context . . . 35
6.1.2 op_context . . . 36
6.2 Functions . . . 36
6.2.1 Global Functions . . . 36
6.2.2 Require functions . . . 37
6.2.3 integer . . . 37
6.2.4 text . . . 38
6.2.5 byte_array . . . 38
6.2.6 range . . . 39
6.2.7 list . . . 39
6.2.8 set . . . 40
6.2.9 map<K,V> . . . 40
6.2.10 enum . . . 41
6.2.11 GTXValue . . . 41
6.2.12 record . . . 42

7 Language specification **43**
7.1 Lexical Rules . . . 43
7.1.1 Whitespaces and comments . . . 43
7.1.2 Identifiers . . . 43
7.1.3 Keywords: . . . 43
7.1.4 Operators and delimiters . . . 44
7.1.5 Integer literals . . . 44

7.1.6 String literals . . . 44
7.1.7 Escape sequences . . . 44
7.1.8 Byte array literals . . . 44
7.2 Types . . . 45
7.2.1 General . . . 45
7.2.2 Built-in types . . . 45
7.2.3 Collection types . . . 46
7.2.4 Subtypes . . . 46
7.3 Classes . . . 47
7.3.1 Class syntax . . . 47
7.3.2 Attributes . . . 47
7.3.3 Keys, indices . . . 48
7.3.4 Handling of fields . . . 48
7.4 Operations, Queries, Functions . . . 49
7.4.1 Syntax . . . 49
7.4.2 Operations . . . 49
7.4.3 Examples . . . 50
7.4.4 Queries . . . 50
7.4.5 Return type . . . 50
7.4.6 Examples . . . 50
7.4.7 Functions . . . 51
7.4.8 Return type . . . 51
7.4.9 Common things for routines . . . 51

8 Client **53**
8.1 Try the example code . . . 53
8.2 Install the client . . . 54
8.3 Connect to the node . . . 54
8.4 Make a transaction (with operations inside) . . . 54
8.5 Query . . . 55

9 Examples **57**
9.1 Account-based token system . . . 57
9.2 Chroma Chat . . . 59
9.2.1 Requirements . . . 59
9.2.2 Class definition . . . 59
9.2.3 Operations . . . 60
9.2.4 Queries . . . 62
9.2.5 Run it . . . 62
9.2.6 Client side . . . 63
9.3 UTXO-based token system . . . 66

CHAPTER 1

Introduction

1.1 Rell SDK

Welcome to the first Rell SDK! This is a milestone in the development of Chromia, and one of the first opportunities to get your hands on the latest tools that have been developed for the platform. In line with our mission to make mainstream dapps a practical reality, we have decided to prioritise providing a rich and usable toolset for dapp developers right from the outset.

This early SDK consists of our Rell IDE which allows you to write and compile Rell code, as well as language documentation, code samples, and a simple tutorial. We will be expanding this material as Rell matures and develops. Right now it gives the dapp developer community an early look at what they will be able to achieve with Chromia and Rell.

We welcome any and all feedback, please send bug reports and suggestions to dev_feedback@chromia.com

1.2 Chromia

Rell is built for Chromia. Chromia is a new blockchain platform for decentralized applications, conceived in response to the shortcomings of existing platforms and designed to enable a new generation of dapps to scale beyond what is currently possible

While platforms such as Ethereum allow any kind of application to be implemented in theory, in practice they have many limitations: bad user experience, high fees, frustrating developer experience, poor security. This prevents decentralized apps (dapps) from going mainstream.

We believe that to address these problems properly we need to seriously rethink the blockchain architecture and programming model with the needs of decentralized applications in mind. Our priorities are to:

- Allow dapps to scale to millions of users.
- Improve the user experience of dapps to achieve parity with centralized applications.
- Allow developers to build secure applications with using familiar paradigms.

1.3 Rell language

Most dapp blockchain platforms use virtual machines of various kinds. But a traditional virtual machine architecture doesn't work very well with the Chromia relational data model, as we need a way to encode queries as well as operations. For this reason, we are taking a more language-centric approach: a new language called Rell (Relational language) will be used for dapp programming. This language allows programmers to describe the data model/schema, queries, and procedural application code.

Rell code is compiled to an intermediate binary format which can be understood as code for a specialized virtual machine. Chromia nodes will then translate queries contained in this code into SQL (while making sure this translation is safe) and execute code as needed using an interpreter or compiler.

Rell has the following features:

- Type safety / static type checks. It's very important to catch programming errors at the compilation stage to prevent financial losses. Rell is much more type-safe than SQL, and it makes sure that types returned by queries match types used in procedural code.
- Safety-optimized. Arithmetic operations are safe right out of the box, programmers do not need to worry about overflows. Authorization checks are explicitly required.
- Concise, expressive and convenient. Many developers dislike SQL because it is highly verbose. Rell doesn't bother developers with details which can be derived automatically. As a data definition language, Rell is up to 7x more compact than SQL.
- Allows meta-programming. We do not want application developers to implement the basics from scratch for every dapp. Rell will allow functionality to be bundled as templates.

Our research indicated that no existing language or environment has this feature set, and thus development of a new language was absolutely necessary.

We designed Rell in such a way that it is easy to learn for programmers:

- Programmers can use relational programming idioms they are already familiar with. However, they don't have to go out of their way to express everything through relational algebra: Rell can seamlessly merge relational constructs with procedural programming.
- The language is deliberately similar to modern programming languages like JavaScript and Kotlin. A familiar language is easier to adapt to, and our internal tests show that programmers can become proficient in Rell in matter of days. In contrast, the ALGOL-style syntax of PL/SQL generally feels unintuitive to modern developers.

CHAPTER 2

Quick start

This document should help you get up and running with the Rell IDE.

2.1 Install Docker

A software called Docker is required in order to run the IDE. It can be downloaded from docker.io, you will also have to create an account.

2.2 Install the IDE

This tutorial assumes that you are using Linux or a Mac OS, and that you have a certain familiarity with *nix operating systems and tools. It is possible to install the IDE on Windows, but we do not have specific instructions for doing so. We hope to add them in future.

1. Create a new directory called `relltest`
2. Create a file called `docker-compose.yml` and paste the following into it:

```
# Copyright (c) 2017 ChromaWay Inc. See README for license information.
version: '3.1'

services:

  postgres:
    image: chromaway/postgres:2.4.3-beta
    restart: always
    environment:
      POSTGRES_PASSWORD: postchain

  dev-preview:
    image: chromaway/dev-preview:0.7.0-b3
```

(continues on next page)

(continued from previous page)

```
    restart: always
    ports:
      - 127.0.0.1:7740:7740
      - 127.0.0.1:30000:30000
    depends_on:
      - postgres
```

3. Open a terminal in the `relltest` directory and type `docker-compose up`. When the terminal stops writing, you should be able to try the IDE, see next section.

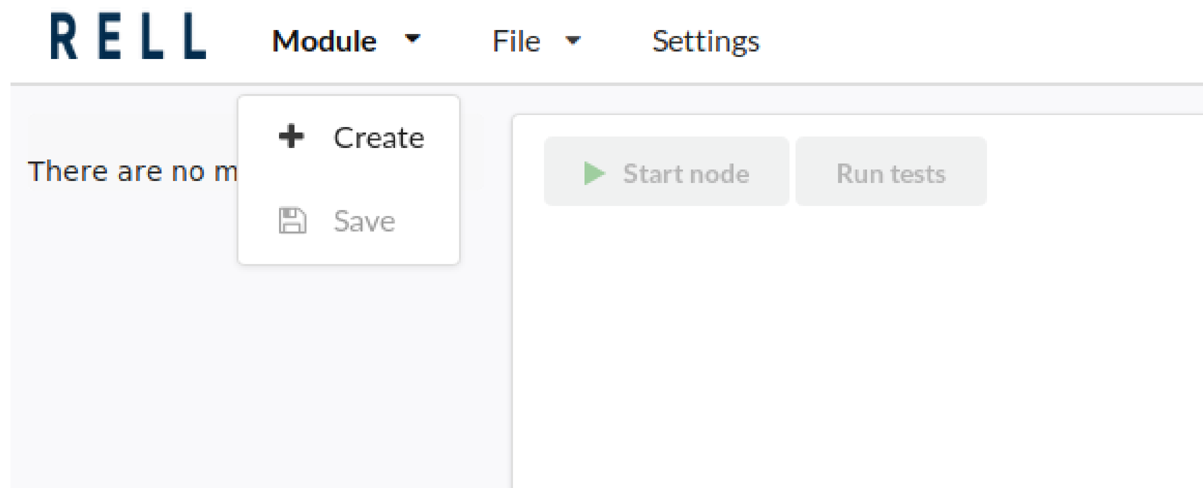
Note: You may need to use `sudo` to execute `docker-compose`.

2.3 Try the IDE

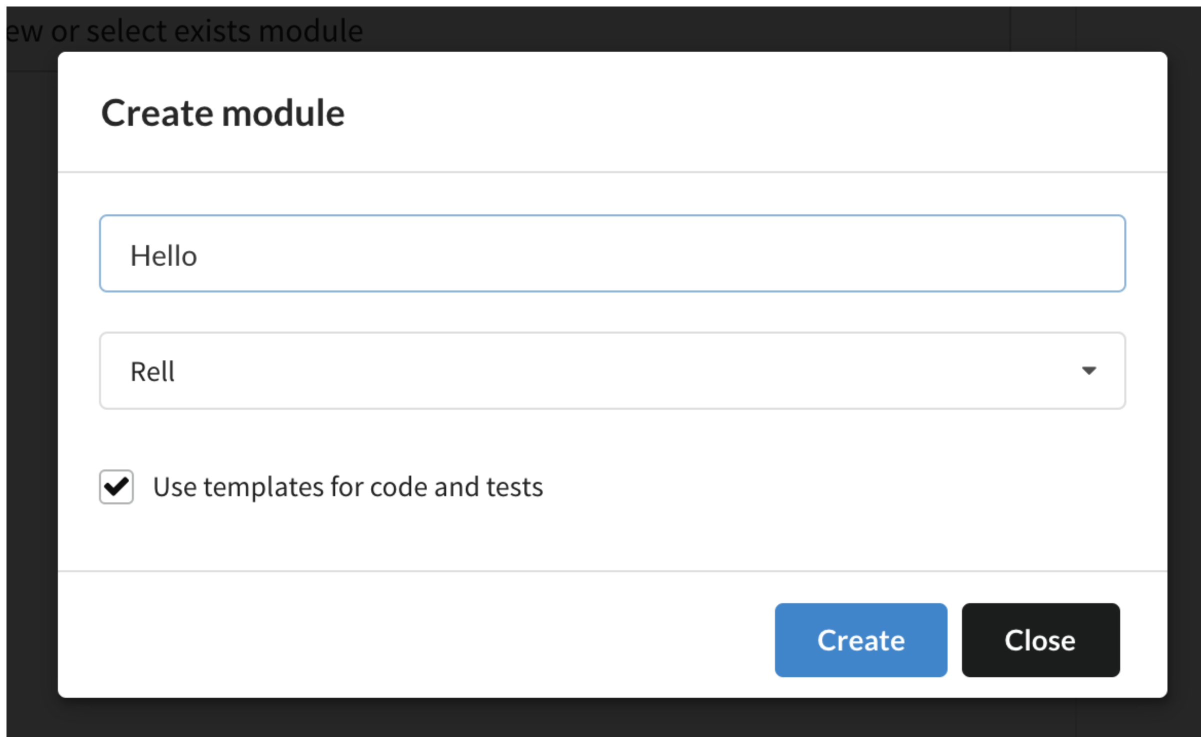
Point your browser to http://localhost:30000. You should see the Rell IDE.

Click Select Module (first element in the top bar) and then Create. A new window will appear. Write any name for your module, such as Hello.

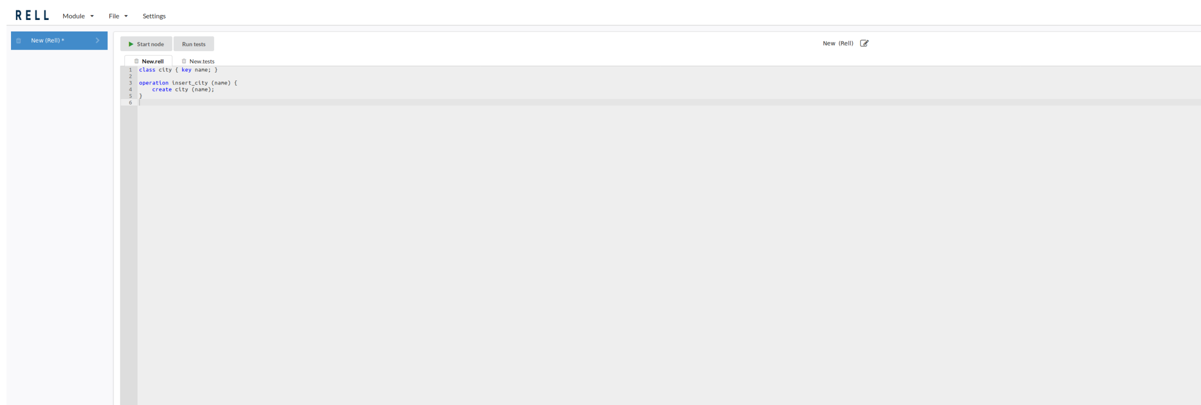
In the dropdown, select "Rell", and select the "Use template for code and tests" option.



This will open a modal element where you can specify the name of the Module and the Language used (in this example: Rell). For convenience you can include template and test code.



Click the blue "Create" button. The screen will now be filled with code.



Browser To the left bar is the Browser. You can use it to work with several examples.

Editor Inside the central element on the left the editor filled with a template of source code.

Test window Inside the central element on the right there is a window with XML. This is a way to run your code, simulating a real application.

Buttons On top of the Editor there is a button "Start Node", don't press that one yet. There is also a button "Run tests". Click it to see that the test passes.

2.4 Hello World!

As a minimal first application, you can make a Hello World example with a focus on Ukraine.

If you checked the *use template* box and look at editor section on the top, you will see this code as a template:

```
class city { key name; }

operation insert_city (name) {
    create city (name);
}
```

This is a small registry of cities. In order to run the code we need a test in XML.

```
<test>
<block>
    <transaction>
    <signers><param type="bytea" key="Alice"/></signers>
    <operations>
        <operation name="insert_city">
        <string>Kiev</string>
        </operation>
    </operations>
    </transaction>
</block>
</test>
```

Click the ‘Run tests’ button, and a green message will appear.

After all this work, we suggest that you put “Relational Blockchain” on your CV.

Next step is to learn more about Rell in the *Tutorial*.

However if you feel eager to click the *Start Node* button and create a complete running application, you can learn how to use the javascript client library in the *Client* section.

2.5 Videos

If you are stuck, we have some videos showing how to get started, on a Mac.

With_docker_installed_installing_developer_preview

CHAPTER 3

Tutorial

3.1 Language overview

Rell is a language for relational blockchain programming. It combines the following features:

1. Relational data modeling and queries similar to SQL. People familiar with SQL should feel at home once they learn the new syntax.
2. Normal programming constructs: variables, loops, functions, collections, etc.
3. Constructs which specifically target application backends and, in particular, blockchain-style programming including request routing, authorization, etc.

Rell aims to make programming as ergonomic as possible. It minimizes boilerplate and repetition. At the same time, as a static type system it can detect and prevent many kinds of defects.

3.2 Blockchain programming

There are many different styles of blockchain programming. In the context of Rell, we see blockchain as a method for secure synchronization of databases on nodes of the system. Thus Rell is very database-centric.

Programming in Rell is pretty much identical to programming application backends: you need to handle requests to modify the data in the database and other requests which retrieve data from a database. Handling these two types of requests is basically all that a backend does.

But, of course, before you implement request handlers, you need to describe your data model first.

3.3 Class definitions

In SQL, usually you define your data model using `CREATE TABLE` syntax. In Java, you can define data objects using `class` definition. (See the section about *Classes* in the language specification for more information).

Rell uses persistent objects, thus a class definition automatically creates the storage (e.g. a table) necessary to persist objects of a class. As you might expect, Rell's class definition includes a list of attributes:

```
class user {
    pubkey: pubkey;
    name: text;
    age: integer;
}
```

It is very common that the name of the attribute is the same as its type. For example, it makes sense to call user's pubkey "pubkey." Rell allows you to shorten `pubkey: pubkey;` to just `pubkey;`. Rell also has a number of convenient semantic types (see here: *Types*), so there is a type called `name` as well. Thus you can rewrite the definition above as just:

```
class user { pubkey; name; }
```

Typically a system should not allow different users to have the same name. That is, names should be unique. If name is unique, it can be used to identify a user. In Rell, this can be done by defining a key, i.e. `key name;`. Note that it's not necessary to define both key and attribute. Rell is smart enough to figure out that if you use an attribute in a key, that attribute should exist in a class.

It also might be useful to find a user by his pubkey. Should it also be unique? Not necessarily. A user might have several different identities. When you want to enable fast retrieval, but do not need uniqueness, you can use `index` definition:

```
class user {
   key name;
   index pubkey;
}
```

However, if you want pubkey to be unique for an user, you can add a second key:

```
class user {
   key name;
   key pubkey;
}
```

(See *Keys, indices* for more information.)

Typically, when you define a class in a programming language, it creates a type which can be used to refer to instances of that class. This is exactly how it works in Rell. The definition of class `user` creates a type `user` which is a type of references to objects stored in a database. References can themselves be used as attributes. For example, you might want to define something owned by a user, say, a channel. You can describe it like this:

```
class channel {
   index owner: user;
   key name;
}
```

`index` makes it possible to efficiently find all channels owned by a user. `key` makes sure that channel names are unique within the system.

Let's analyze `channel` class definition from a point of view of a traditional relational database terminology. A single user can be associated with multiple `channel` objects, but a single `channel` is always related to a single `user`. Thus this represents one-to-many relationship. `owner` attribute of a channel refers to `user` object and thus constitutes a foreign key.

If channel names should be unique only in context of a single user (e.g. `alice/news` and `bob/news` are different channels), then a composite key can be used:

```
class channel {
   key owner: user, name;
}
```

This basically means that a pair of `(owner, name)` should be unique.

Finally, one might ask: what changes if we change `index owner: user` to `key owner: user`? This makes a user reference unique per `channel` table, thus there can be at most one channel per user in that case. (I.e. if `owner` is declared as a key, relationship between users and channels becomes a one-to-one relationship.)

3.4 Operations

Now that we defined the data model, we can finally get to handling requests. As previously mentioned, Rell works with two types of requests:

1. Data-modifying requests. We call them `operations` which are applied to the database state.
2. Data-retrieving requests. We call them `queries`.

But for both types of requests we are going to need to refer to things in the database, so let's consider relational operators first.

3.4.1 Relational operator basics

First, let's look how we create objects:

```
create user (pubkey=x
↪"0373599a61cc6b3bc02a78c34313e1737ae9cfd56b9bb24360b437d469efdf3b15",
             name="Alice");
```

This is essentially the same as `INSERT` operation in SQL, but the syntax is a bit different. Rell is smart enough to identify the connection between arguments and attributes based on their type. `x"..."` notation is a hexadecimal `byte_array` literal which is compatible with `pubkey` type. On the other hand, name is provided via `text` literal. Thus we can write:

```
create user("Alice", x
↪"0373599a61cc6b3bc02a78c34313e1737ae9cfd56b9bb24360b437d469efdf3b15");
```

The order of arguments does not matter here, they are matched with attributes based on types.

How do we find that object now?

```
val alice = user @ {.name=="Alice"};
```

`@` operator retrieves a single record (or an object in this case) satisfying the search criteria you provided. If there is no such record, or more than one exists, it raises an error. It's recommended to use this construct when an operation needs a single record to operate on. If this requirement is violated the operation will be aborted and all its effects will be rolled back. Thus it is a succinct and effective way to deal with requirements.

(`val` defines a read-only variable which can later be used in an expression. A variable defined using `var` can be reassigned later.)

If you want to retrieve a list of users, you can use the `@*` operator. For example:

```
val all_users = user @* {};
```

This returns a list of all users (since no filter expression was provided, all users match it). Value declarations can include a type, for example, we can specify that `all_users` is of type `list<user>` like this:

```
val all_users: list<user> = user @* {};
```

Since the Rell compiler knows a type of every expression it does not really need a type declaration, however, if one is provided, it will check against it. Type declarations are mostly useful as documentation for programmers reading the code and should be omitted in cases where there is no ambiguity.

Both `@` and `@*` correspond to `SELECT` in SQL. A complete list of operators can be found in *Operators and delimiters*.

3.4.2 Simple operation

Let's make an operation which allows a user to create a new channel:

```
operation register_channel (user_pubkey: pubkey, channel_name: name) {
    require( is_signer(user_pubkey) );
    create channel (
        owner = user@{.pubkey == user_pubkey},
        name = channel_name
    );
}
```

Let's go through this line by line. First we declare the operation name and a list of parameters:

```
operation register_channel (user_pubkey: pubkey, channel_name: name) {
```

This is very similar to a function definitions in other languages. In fact, an operation is a function of a special kind: it can be invoked using a blockchain transaction by its name. When invoking `register_channel`, the caller must provide two arguments of specified types, otherwise it will fail.

```
require( is_signer(user_pubkey) );
```

We don't want Alice to be able to pull a prank on Bob by registering a channel with a silly name on his behalf. Thus we need to make sure that the transaction was signed with a key corresponding to the public key specified in the first parameter. (In other words, if Bob's public key is passed as `user_pubkey`, the transaction must also be signed by Bob, that is, Bob is a signer of this transaction.) This is a common pattern in Rell – typically you specify an actor in a parameter of an operation and in the body of the operation you verify that the actor was actually the signer. `require` fails the operation if the specified condition is not met.

Note: What happens with a failed operation? A transaction which contains this operation will be rejected, that is, it won't be included into a blockchain and will be eventually forgotten. When developer is debugging an application he might be able to retrieve a message which describes a reason why a transaction was rejected. However, in a real production use this cannot be relied upon, as blockchain nodes can be bombed with large quantities of invalid transactions, thus nodes normally won't store a list of rejected transactions.

`create channel`, obviously, creates a persistent object `channel`. You don't need to explicitly store it, as all created objects are persisted if operation succeeds.

`user@{.pubkey=user_pubkey}` – now we retrieve a user object by its pubkey, which should be unique. If no such user exists operation will fail. We do not need to test for that explicitly as `@` operator will do this job.

Rell can automatically find attribute names corresponding to arguments using types. As `user` and `name` are different types, `create channel` can be written like this:

```
create channel (user@{.pubkey=user_pubkey}, channel_name);
```

3.4.3 Function

Sometimes multiple operations (or queries) need a same piece functionality, e.g. some kind of a validation code, or code which retrieves objects in a particular way. In order to not repeat yourself you can use `function`. Functions work similarly to operations: they get some input and can perform validations and work with data. Additionally, they also have a return type which can be specified after the list of parameters. For example, if you want to allow the user of a channel to change the name of the channel itself:

```
// We added mutable specifier to channel's attribute "name" to make name editable.
// Note that in case both an attribute and a key need to be declared.

class channel {
    mutable name;
    key name;
    index owner: user;
}

function get_channel_owned_by_user(user_pub: pubkey, channel_name: name): channel {
    val user = user@{.pubkey == user_pub};
    return channel@{channel_name, .owner == user};
}

operation change_channel_name(signer: pubkey, old_channel_name: name, new_channel_
↪name: name) {
    require(is_signer(signer));
    val channel_to_change = get_channel_owned_by_user(signer, old_channel_name);
    update channel@{channel == channel_to_change}(.name = new_channel_name);
}
```

In the function `get_channel_owned_by_user` the code first retrieves a user with given public key and returns a channel owned by the the retrieved user with the given channel name. Operator `@` expects exactly one object to be found (see *Cardinality* for more information.), thus you can be sure that in case there is no user or channel with such a pubkey or a name the function will fail and so will the operation that is calling it. Finally, the function returns the channel instance that was validated, saving the developer the hassle to check owner every time a channel is retrieved.

Please note that you must mark the attribute `name` with the keyword `mutable`. This is because only the fields which are declared mutable can be changed using the update statement.

3.5 Query

Storing data without the ability to access it again would be useless. Let's consider a simple example - retrieving channel names for a user with a certain name:

::

query get_channel_names (user_name: name) {

return channel @* { .owner == user@{.name==user_name}

} (.name);

}

Here you see a selection operator you're already familiar with – `@*`. We select all the channels with a given owner (which we first find by name).

Then we extract name attribute from retrieved objects using the `(.name)` construct.

Note that since we only need `name` from channel, is also possible to write

::

query get_channel_names (user_name: name) {

return channel @* { .owner == user@{.name==user_name}

}.name;

}

3.6 Relational expressions

In general, a relational expression consists of five parts, some of which can be omitted:

```
FROM OPERATOR { WHERE } (WHAT) LIMIT
```

1. *FROM* describes where data is taken from. It can be a single class, such as just `user`. Or, it can be combination of multiple classes, e.g. `(user, channel)`. In the later case, conceptually we are dealing with a Cartesian product, which is a set of all possible combinations. But, in typically *WHERE* part will then provide a condition which defines a correspondence between objects of difference classes. E.g. one can select such `(user, channel)` combinations where `user` is an owner of the `channel`. This works same way as `JOIN` in SQL, in fact, the optimizer will typically translate it to JOINs.
2. *OPERATOR* – there are different operators depending on required cardinality. They are:
   - `@` – exactly one, returns a value
   - `@*` – any number, returns a list of values
   - `@+` – at least one, returns a list of values
   - `@?` – one or zero, returns a nullable value
3. *WHERE* describes how to filter the *FROM* set. So, you would use your search criteria as well as JOINs.
4. *WHAT* describes how to process the set, for doing a projection, aggregation or sorting. If it is ommitted then members of the set are returned as they are.
5. *LIMIT* for operators which return a list, limits the number of elements returned.

In SQL, the logical processing order does not match the order in which clauses are written, for example, `FROM` is logically processed before `SELECT` even though `SELECT` comes first. (SQL logical processing order can be found e.g. in SQL Server documentation).

The order of components of a relational expression in Rell matches the logical processing order. So, first a set is defined, then it is filtered, and then it is post-processed. Of course, the query planner is allowed to perform operations in a different order, but that shouldn't affect the results. Thus a relational expression can be understood as a kind of a pipeline.

Let's see some examples of relational expressions. Suppose in addition to `user` and `channel` classes we provided before, we also have:

```
class message {
    index channel;
    index timestamp;
    text;
}
```

We can retrieve all messages of a given user:

```
(channel, message) @* {
    channel.owner == given_user, message.channel == channel
}(message.text);
```

So, basically, we join `channel` with `message`. We can shorten the expression using class aliases:

```
(c: channel, m: message) @* { c.owner == given_user, m.channel == c } (m.text, m.
↪timestamp)
```

We can easily read this expression left to right:

- consider all pairs `(c, m)` where `c` is `channel` and `m` is `message`
- find those where `c.owner` equals `given_user` and `m.channel` equals `c`
- extract `text` and `timestamp` from `m`

The result of this expression is a list of tuples with `text` and `timestamp` attributes.

The above expression can be easily modified to retrieve the latest 25 messages:

```
(c: channel, m: message) @* {
    c.owner == given_user, m.channel == c
} (m.text, -sort m.timestamp) limit 25
```

Here we sorted results by timestamp in a descending order using `-sort` (minus prefix means descending) and limited the number of returned rows.

3.6.1 Composite indices

We can also only select recent messages by adding, for example, `m.timestamp >= given_timestamp` condition to *WHERE* part. But a database cannot filter messages efficiently (that is, without considering every message) using two criteria at once unless we create a *composite index*, changing the `message` class definition in the following way:

```
class message {
    index channel, timestamp;
    text;
}
```

Instead two separate indexes we got one composite index. The idea here is that we want to retrieve not the latest messages overall, but the latest messages *for a given channel*. Thus, we need to order messages by channels first. Paged retieval can be done using the following query:

```
query get_next_messages (user_name: name, upto_timestamp: timestamp) {
   val given_user = user@{user_name};
   return (c: channel, m: message) @* {
       c.owner == given_user, m.channel == c, m.timestamp < upto_timestamp
   } (m.text, -sort m.timestamp) limit 25;
}
```

This can be used in an app like Twitter. A visitor might first retrieve the latest 25 messages, then go further – in which case the client will send a query with a timestamp of the oldest message retrieved.

To understand why this can work efficiently, consider that the index stores an ordered collection of pairs. For example:

```
1. (channel_1, 1000000) -> m1
2. (channel_1, 1000050) -> m3
3. (channel_1, 1000100) -> m5
4. (channel_2, 1000025) -> m2
5. (channel_2, 1000075) -> m4
```

A database can efficiently find a place which corresponds to a given timestamp in a given channel and traverse the index through it.

Note: It's worth noting that all SQL databases work this way, this feature is not unique to Rell. But in a decentralized system resources are typically precious, thus it is important for Rell programmers to understand the query behavior and use indices efficiently.

3.7 Examples and further exercises

We have prepared some examples of how to implement other functionality in Rell and Chromia.

- *Chroma Chat*
- *Account-based token system*
- *UTXO-based token system*

CHAPTER 4

General Language Features

4.1 Types

4.1.1 Simple types:

- `boolean`
- `integer`
- `text`
- `byte_array`
- `json`
- `unit` (no value; cannot be used explicitly)
- `null` (type of `null` expression; cannot be used explicitly)

Simple type aliases:

- `pubkey` = `byte_array`
- `name` = `text`
- `timestamp` = `integer`
- `tuid` = `text`

4.1.2 Complex types:

- class
- `T?` - nullable type
- `record`

- tuple: `(T1, ..., Tn)`
- `list<T>`
- `set<T>`
- `map<K,V>`
- `range` (can be used in `for` statement)
- `GTXValue` - used to encode parameters and results of operations and queries

4.1.3 Nullable type

- Class attributes cannot be nullable.
- Can be used with almost any type (except nullable, `unit`, `null`).
- Nullable nullable (`T??` is not allowed).
- Normal operations of the underlying type cannot be applied directly.
- Supports `?:`, `?.` and `!!` operators (like in Kotlin).

Compatibility with other types:

- Can assign a value of type `T` to a variable of type `T?`, but not the other way round.
- Can assign `null` to a variable of type `T?`, but not to a variable of type `T`.
- Can assign a value of type `(T)` (tuple) to a variable of type `(T?)`.
- Cannot assign a value of type `list<T>` to a variable of type `list<T?>`.

Allowed operations:

- Null comparison: `x == null`, `x != null`.
- `?:` - Elvis operator: `x ?: y` means `x` if `x` is not `null`, otherwise `y`.
- `?.` - safe access: `x?.y` results in `x.y` if `x` is not `null` and `null` otherwise.
- Operator `?.` can be used with function calls, e. g. `x?.upperCase()`.
- `!!` - null check operator: `x!!` returns value of `x` if `x` is not `null`, otherwise throws an exception.
- `require(x)`, `requireNotEmpty(x)`: throws an exception if `x` is `null`, otherwise returns value of `x`.

Examples:

```
val x: integer? = 123;
val y = x;            // type of "y" is "integer?"
val z = y!!;          // type of "z" is "integer"
val p = require(y);   // type of "p" is "integer"
```

4.1.4 Tuple type

Examples:

- `(integer)` - one value
- `(integer, text)` - two values
- `(integer, (text, boolean))` - nested tuple

- (x: integer, y: integer) - named fields (can be accessed as A.x, A.y)

Tuple types are compatible only if names and types of fields are the same:

- (x:integer, y:integer) and (a:integer,b:integer) are not compatible.
- (x:integer, y:integer) and (integer,integer) are not compatible.

4.1.5 Collection types

Collection types are:

- list<T> - an ordered list
- set<T> - an unordered set, contains no duplicates
- map<K,V> - a key-value map

Collection types are mutable, elements can be added or removed dynamically.

Only a non-mutable type can be used as a map key or a set element.

Following types are mutable:

- Collection types (list, set, map) - always.
- Nullable type - only if the underlying type is mutable.
- Record type - if the record has a mutable field, or a field of a mutable type.
- Tuple - if a type of an element is mutable.

4.1.6 GTXValue

GTXValue is a type used to repsesent encoded arguments and results of remote operation and query calls. It may be a simple value (integer, string, byte array), an array of values or a string-keyed map.

Some Rell types are not GTX-compatible. Values of such types cannot be converted to/from GTXValue, and the types cannot be used as types of operation/query parameters or result.

Rules of GTX-compatibility:

- a map<K,V> is GTX-compatible only if K is text.
- range is not GTX-compatible
- a complex type is not GTX-compatible if a type of its component is not GTX-compatible

For queries, a type must be pretty-GTX-compatible. Rules are:

- a type must be GTX-compatible
- for tuples: either all fields have names, or no field has a name
- component types must be pretty-GTX-compatible as well

4.1.7 Subtypes

If type B is a subtype of type A, a value of type B can be assigned to a variable of type A (or passed as a parameter of type A).

- T is a subtype of T?.

- `null` is a subtype of `T?`.
- `(T,P)` is a subtype of `(T?,P?)`, `(T?,P)` and `(T,P?)`.

4.2 Module definitions

4.2.1 Class

```
class company {
    name: text;
    address: text;
}

class user {
    firstName: text;
    lastName: text;
    yearOfBirth: integer;
    mutable salary: integer;
}
```

If attribute type is not specified, it will be the same as attribute name:

```
class user {
    name;        // built-in type "name"
    company;     // user-defined type "company" (error if no such type)
}
```

Attributes may have default values:

```
class user {
    homeCity: text = 'New York';
}
```

Keys and Indices

Classes can have `key` and `index` clauses:

```
class user {
    name: text;
    address: text;
    key name;
    index address;
}
```

Keys and indices may have multiple attributes:

```
class user {
    firstName: text;
    lastName: text;
    key firstName, lastName;
}
```

Attribute definitions can be combined with `key` or `index` clauses, but such definition has restrictions (e. g. cannot specify `mutable`):

```
class user {
    key firstName: text, lastName: text;
    index address: text;
}
```

Class annotations

```
class user (log) {
    name: text;
}
```

The `log` annotation has following effects:

- Special attribute `transaction` of type `transaction` is added to the class.
- When an object is created, `transaction` is set to the result of `op_context.transaction` (current transaction).
- Class cannot have mutable attributes.
- Objects cannot be deleted.

4.2.2 Object

Object is similar to class, but there can be only one instance of an object:

```
object event_stats {
    mutable event_count: integer = 0;
    mutable last_event: text = 'n/a';
}
```

Reading object attributes:

```
query get_event_count() = event_stats.event_count;
```

Modifying an object:

```
operation process_event(event: text) {
    update event_stats ( event_count += 1, last_event = event );
}
```

Features of objects:

- Like classes, objects are stored in a database.
- Objects are initialized automatically during blockchain initialization.
- Cannot create or delete an object from code.
- Attributes of an object must have default values.

4.2.3 Record

Record declaration:

```
record user {
    name: text;
    address: text;
    mutable balance: integer = 0;
}
```

- Attributes are immutable by default, and only mutable when declared with `mutable` keyword.
- An attribute may have a default value, which is used if the attribute is not specified during construction.

Creating record values:

```
val u = user(name = 'Bob', address = 'New York');
```

Same rules as for the `create` expression apply: no need to specify attribute name if it can be resolved implicitly by name or type:

```
val name = 'Bob';
val address = 'New York';
val u = user(name, address);
val u2 = user(address, name); // Order does not matter - same record object is created.
```

Record attributes can be accessed using operator `.`:

```
print(u.name, u.address);
```

Safe-access operator `?.` can be used to read or modify attributes of a nullable record:

```
val u: user? = findUser('Bob');
u?.balance += 100;        // no-op if 'u' is null
```

4.2.4 Enum

Enum declaration:

```
enum currency {
    USD,
    EUR,
    GBP
}
```

Values are stored in a database as integers. Each constant has a numeric value equal to its position in the enum (the first value is 0).

Usage:

```
var c: currency;
c = currency.USD;
```

Enum-specific functions and properties:

```
val cs: list<currency> = currency.values() // Returns all values (in the order in
↪which they are declared)

val eur = currency.value('EUR') // Finds enum value by name
val gbp = currency.value(2) // Finds enum value by index

val usdStr: text = currency.USD.name // Returns 'USD'
val usdValue: integer = currency.USD.value // Return 0.
```

4.2.5 Query

- Cannot modify the data in the database (compile-time check).
- Must return a value.
- If return type is not explicitly specified, it is implicitly deducted.
- Parameter types and return type must be pretty-GTX-compatible.

Short form:

```
query q(x: integer): integer = x * x;
```

Full form:

```
query q(x: integer): integer {
    return x * x;
}
```

4.2.6 Operation

- Can modify the data in the database.
- Does not return a value.
- Parameter types must be GTX-compatible.

```
operation createUser(name: text) {
    create user(name = name);
}
```

4.2.7 Function

- Can return nothing or a value.
- Can modify the data in the database when called from an operation (run-time check).
- Can be called from queries, operations or functions.
- If return type is not specified explicitly, it is `unit` (no return value).

Short form:

```
function f(x: integer): integer = x * x;
```

Full form:

```
function f(x: integer): integer {
    return x * x;
}
```

When return type is not specified, it is considered `unit`:

```
function f(x: integer) {
    print(x);
}
```

4.3 Expressions

4.3.1 Values

Simple values:

- Null: `null` (type is `null`)
- Boolean: `true`, `false`
- Integer: `123`, `0`, `-456`
- Text: `'Hello'`, `"World"`
- Byte array: `x'1234'`, `x"ABCD"`

Text literals may have escape-sequences:

- Standard: `\r`, `\n`, `\t`, `\b`.
- Special characters: `\"`, `\'`, `\\`.
- Unicode: `\u003A`.

Tuple:

- `(1, 2, 3)` - three values
- `(123, 'Hello')` - two values
- `(456,)` - one value (because of the comma)
- `(789)` - not a tuple (no comma)
- `(a = 123, b = 'Hello')` - tuple with named fields

List:

```
[ 1, 2, 3, 4, 5 ]
```

Map:

```
[ 'Bob' : 123, 'Alice' : 456 ]
```

4.3.2 Operators

Special:

- `.` - member access: `user.name`, `s.sub(5, 10)`
- `()` - function call: `print('Hello')`, `value.str()`
- `[]` - element access: `values[i]`

Null handling:

- `?:` - Elvis operator: `x ?: y` returns `x` if `x` is not `null`, otherwise returns `y`.
- `?.` - safe access operator: `x?.y` returns `x.y` if `x` is not `null`, otherwise returns `null`; similarly, `x?.y()` returns either `x.y()` or `null`.
- `!!` - null check: `x!!` returns `x` if `x` is not `null`, otherwise throws an exception.

Examples:

```
val x: integer? = 123;
val y = x;              // type of "y" is "integer?"

val a = y ?: 456;       // type of "a" is "integer"
val b = y ?: null;      // type of "b" is "integer?"

val p = y!!;            // type of "p" is "integer"
val q = y?.hex();       // type of "q" is "text?"
```

Comparison:

- ==
- !=
- ===
- !==
- <
- >
- <=
- >=

Operators == and != compare values. For complex types (collections, tuples, records) they compare member values, recursively. For `class` object values only object IDs are compared.

Operators === and !== compare references, not values. They can be used only on types: tuple, `record`, `list`, `set`, `map`, `GTXValue`, `range`.

Example:

```
val x = [1, 2, 3];
val y = list(x);
print(x == y);      // true - values are equal
print(x === y);     // false - two different objects
```

If:

Operator `if` is used for conditional evaluation:

```
val max = if (a >= b) a else b;
return max;
```

Arithmetical:

- `+`
- `-`
- `*`
- `/`
- `%`

Logical:

- `and`
- `or`
- `not`

Other:

- `in` - check if an element is in a range/set/map

4.4 Statements

4.4.1 Local variable declaration

Constants:

```
val x = 123;
val y: text = 'Hello';
```

Variables:

```
var x: integer;
var y = 123;
var z: text = 'Hello';
```

4.4.2 Basic statements

Assignment:

```
x = 123;
values[i] = z;
y += 15;
```

Function call:

```
print('Hello');
```

Return:

```
return;
return 123;
```

Block:

```
{
    val x = calc();
    print(x);
}
```

4.4.3 If statement

```
if (x == 5) print('Hello');

if (y == 10) {
    print('Hello');
} else {
    print('Bye');
}

if (x == 0) {
    return 'Zero';
} else if (x == 1) {
    return 'One';
} else {
    return 'Many';
}
```

4.4.4 Loop statements

For:

```
for (x in range(10)) {
    print(x);
}

for (u in user @* {}) {
    print(u.name);
}
```

The expression after `in` may return a `range` or a collection (`list`, `set`, `map`).

While:

```
while (x < 10) {
    print(x);
    x = x + 1;
}
```

Break:

```
for (u in user @* {}) {
    if (u.company == 'Facebook') {
        print(u.name);
        break;
    }
}

while (x < 5) {
    if (values[x] == 3) break;
    x = x + 1;
}
```

4.5 Miscellaneous

4.5.1 Comments

Single-line comment:

```
print("Hello"); // Some comment
```

Multiline comment:

```
print("Hello"/*, "World"*/);
/*
print("Bye");
*/
```

CHAPTER 5

Database Operations

5.1 At-Operator

Simplest form:

```
user @ { .name == 'Bob' }
```

General syntax:

```
<from> <cardinality> { <where> } [<what>] [limit N]
```

5.1.1 Cardinality

Specifies whether the expression must return one or many objects:

- `T @? {}` - returns `T`, zero or one, fails if more than one found.
- `T @ {}` - returns `T`, exactly one, fails if zero or more than one found.
- `T @* {}` - returns `list<T>`, zero or more.
- `T @+ {}` - returns `list<T>`, one or more, fails if none found.

5.1.2 From-part

Simple (one class):

```
user @* { .name == 'Bob' }
```

Complex (one or more classes):

```
(user, company) @* { user.name == 'Bob' and company.name == 'Microsoft' and
user.xyz == company.xyz }
```

Specifying class aliases:

`(u: user) @* { u.name == 'Bob' }`

`(u: user, c: company) @* { u.name == 'Bob' and c.name == 'Microsoft' and u.xyz == c.xyz }`

5.1.3 Where-part

Zero or more comma-separated expressions using class attributes, local variables or system functions:

`user @* {}` - returns all users

`user @ { .name == 'Bill', .company == 'Microsoft' }` - returns a specific user (all conditions must match)

Attributes of a class can be accessed with a dot, e. g. `.name` or with a class name or alias, `user.name`.

Class attributes can also be matched implicitly by name or type:

```
val ms = company @ { .name == 'Microsoft' };
val name = 'Bill';
return user @ { name, ms };
```

Explanation: the first where-expression is the local variable `name`, there is an attribute called `name` in the class `user`. The second expression is `ms`, there is no such attribute, but the type of the local variable `ms` is `company`, and there is an attribute of type `company` in `user`.

5.1.4 What-part

Simple example:

`user @ { .name == 'Bob' } ( .company.name )` - returns a single value (name of the user's company)

`user @ { .name == 'Bob' } ( .company.name, .company.address )` - returns a tuple of two values

Specifying names of result tuple fields:

`user @* {} ( x = .company.name, y = .company.address, z = .yearOfBirth )` - returns a tuple with named fields (`x`, `y`, `z`)

Sorting:

`user @* {} ( sort .lastName, sort .firstName )` - sort by `lastName` first, then by `firstName`.

`user @* {} ( -sort .yearOfBirth, sort .lastName )` - sort by `yearOfBirth` desdending, then by `lastName` ascending.

Field names can be combined with sorting:

`user @* {} ( sort x = .lastName, -sort y = .yearOfBirth )`

When field names are not specified explicitly, they can be deducted implicitly by attribute name:

```
val u = user @ { ... } ( .firstName, .lastName, age = 2018 - .yearOfBirth );
print(u.firstName, u.lastName, u.age);
```

5.1.5 Tail part

Limiting records:

`user @* { .company == 'Microsoft' } limit 10`

Returns at most 10 objects. The limit is applied before the cardinality check, so the following code can't fail with "more than one object" error:

`val u: user = user @ { .company == 'Microsoft' } limit 1;`

5.1.6 Result type

Depends on the cardinality, from- and what-parts.

- From- and what-parts define the type of a single record, `T`.
- Cardinality defines the type of the @-operator result: `T?`, `T` or `list<T>`.

Examples:

- `user @ { ... }` - returns `user`
- `user @? { ... }` - returns `user?`
- `user @* { ... }` - returns `list<user>`
- `user @+ { ... }` - returns `list<user>`
- `(user, company) @ { ... }` - returns a tuple `(user,company)`
- `(user, company) @* { ... }` - returns ‘list<(user,company)>‘‘
- `user @ { ... } ( .name )` - returns `text`
- `user @ { ... } ( .firstName, .lastName )` - returns `(text,text)`
- `(user, company) @ { ... } ( user.firstName, user.lastName, company )`
- returns `(text,text,company)`

5.1.7 Nested At-Operators

A nested at-operator can be used in any expression inside of another at-operator:

`user @* { .company == company @ { .name == 'Microsoft' } } ( ... )`

This is equivalent to:

```
val c = company @ { .name == 'Microsoft' };
user @* { .company == c } ( ... )
```

5.2 Create Statement

Must specify all attributes that don't have default values.

```
create user(name = 'Bob', company = company @ { .name == 'Amazon' });
```

No need to specify attribute name if it can be matched by name or type:

```
val name = 'Bob';
create user(name, company @ { company.name == 'Amazon' });
```

Can use the created object:

```
val newCompany = create company(name = 'Amazon');
val newUser = create user(name = 'Bob', newCompany);
print('Created new user:', newUser);
```

5.3 Update Statement

Operators `@`, `@?`, `@*`, `@+` are used to specify cardinality, like for the at-operator. If the number of updated records does not match the cardinality, a run-time error occurs.

```
update user @ { .name == 'Bob' } ( company = 'Microsoft' );             // exactly one
update user @? { .name == 'Bob' } ( deleted = true );                   // zero or one
update user @* { .company.name == 'Bad Company' } ( salary -= 1000 );   // any number
```

Can change only `mutable` attributes.

Class attributes can be matched implicitly by name or type:

```
val company = 'Microsoft';
update user @ { .name == 'Bob' } ( company );
```

Using multiple classes with aliases. The first class is the one being updated. Other classes can be used in the where-part:

```
update (u: user, c: company) @ { u.xyz == c.xyz, u.name == 'Bob', c.name == 'Google' }
↪ ( city = 'Seattle' );
```

Can specify an arbitrary expression returning a class, a nullable class or a collection of a class:

```
val u = user @? { .name == 'Bob' };
update u ( salary += 5000 );
```

A single attribute of can be modified using a regular assignment syntax:

```
val u = user @ { .name == 'Bob' };
u.salary += 5000;
```

5.4 Delete Statement

Operators `@`, `@?`, `@*`, `@+` are used to specify cardinality, like for the at-operator. If the number of deleted records does not match the cardinality, a run-time error occurs.

```
delete user @ { .name == 'Bob' };                    // exactly one
delete user @? { .name == 'Bob' };                   // zero or one
delete user @* { .company.name == 'Bad Company' };   // any number
```

Using multiple classes. Similar to `update`, only the object(s) of the first class will be deleted:

```
delete (u: user, c: company) @ { u.xyz == c.xyz, u.name == 'Bob', c.name == 'Google' }
↪;
```

Can specify an arbitrary expression returning a class, a nullable class or a collection of a class:

```
val u = user @? { .name == 'Bob' };
delete u;
```

CHAPTER 6

Library

6.1 System classes

```
class block {
    block_height: integer;
    block_rid: byte_array;
    timestamp;
}

class transaction {
    tx_rid: byte_array;
    tx_hash: byte_array;
    tx_data: byte_array;
    block;
}
```

It is not possible to create, modify or delete objects of those classes in code.

6.1.1 chain_context

`chain_context.raw_config: GTXValue` - blockchain configuration object, e. g. `{"gtx":{"rellSrcModule":"foo.rell"}}`

`chain_context.args: module_args?` - module arguments specified in `raw_config` under path `gtx.rellModuleArgs`. The type is `module_args`, which must be a user-defined record. If no `module_args` record is defined in the module, the `args` field cannot be accessed. The value is `null` if arguments are not specified in the module configuration.

Example of `module_args`:

```
record module_args {
    s: text;
    n: integer;
}
```

Corresponding module configuration:

```
{
    "gtx": {
        "rellSrcModule": "foo.rell",
        "rellModuleArgs": {
            "s": "Hello",
            "n": 123
        }
    }
}
```

Code that reads `module_args`:

```
function f() {
    print(chain_context.args?.s);
    print(chain_context.args?.n);
}
```

6.1.2 op_context

`op_context.last_block_time: integer` - the timestamp of the last block, in milliseconds (like `System.currentTimeMillis()` in Java). Returns `-1` if there is no last block (the block currently being built is the first block). Can be used only in an operation or a function called from an operation, but not in a query.

`op_context.transaction: transaction` - the transaction currently being built. Can be used only in an operation or a function called from an operation, but not in a query.

6.2 Functions

6.2.1 Global Functions

`abs(integer): integer` - absolute value

`is_signer(byte_array): boolean` - returns `true` if a byte array is in the list of signers of current operation

`json(text): json` - parse a JSON

`log(...)` - print a message to the log (same usage as `print`)

`max(integer, integer): integer` - maximum of two values

`min(integer, integer): integer` - minimum of two values

`print(...)` - print a message to STDOUT:

- `print()` - prints an empty line
- `print('Hello', 123)` - prints `"Hello 123"`

6.2.2 Require functions

For checking a boolean condition:

`require(boolean[, text])` - throws an exception if the argument is `false`

For checking for `null`:

`require(T?[, text]): T` - throws an exception if the argument is `null`, otherwise returns the argument

`requireNotEmpty(T?[, text]): T` - same as the previous one

For checking for an empty collection:

`requireNotEmpty(list<T>[, text]): list<T>` - throws an exception if the argument is an empty collection, otherwise returns the argument

`requireNotEmpty(set<T>[, text]): set<T>` - same as the previous

`requireNotEmpty(map<K,V>[, text]): map<K,V>` - same as the previous

When passing a nullable collection to `requireNotEmpty`, it throws an exception if the argument is either `null` or an empty collection.

Examples:

```
val x: integer? = calculate();
val y = require(x, "x is null"); // type of "y" is "integer", not "integer?"

val p: list<integer> = getList();
requireNotEmpty(p, "List is empty");

val q: list<integer>? = tryToGetList();
require(q);          // fails if q is null
requireNotEmpty(q); // fails if q is null or an empty list
```

6.2.3 integer

`integer.MIN_VALUE` = minimum value (`-2^63`)

`integer.MAX_VALUE` = maximum value (`2^63-1`)

`integer(s: text, radix: integer = 10)` - parse a signed representation, fail if invalid

`integer.parseHex(text): integer` - parse an unsigned HEX representation

`.hex(): text` - convert to an unsigned HEX representation

`.str(radix: integer = 10)` - convert to a signed string representation

`.signum(): integer` - returns `-1`, `0` or `1` depending on the sign

6.2.4 text

`.empty(): boolean`

`.size(): integer`

`.compareTo(text): integer` - as in Java

`.startsWith(text): boolean`

`.endsWith(text): boolean`

`.contains(text): boolean` - `true` if contains the given substring

`.indexOf(text, start: integer = 0): integer` - returns `-1` if substring is not found (as in Java)

`.lastIndexOf(text[, start: integer]): integer` - returns `-1` if substring is not found (as in Java)

`.sub(start: integer[, end: integer]): text` - get a substring (start-inclusive, end-exclusive)

`.replace(old: text, new: text)`

`.upperCase(): text`

`.lowerCase(): text`

`.split(text): list<text>` - strictly split by a separator (not a regular expression)

`.trim(): text` - remove leading and trailing whitespace

`.matches(text): boolean` - `true` if matches a regular expression

`.encode(): byte_array` - convert to a UTF-8 encoded byte array

`.charAt(integer): integer` - get a 16-bit code of a character

`.format(...)` - formats a string (as in Java):

- `'My name is <%s>'.format('Bob')` - returns `'My name is <Bob>'`

Special operators:

- `+` : concatenation
- `[]` : character access (returns single-character `text`)

6.2.5 byte_array

`byte_array(text)` - create a `byte_array` from a HEX string, e.g.`'1234abcd'`

`byte_array(list<integer>)` - create a `byte_array` from a list; values must be 0 - 255

`.empty(): boolean`

`.size(): integer`

`.decode(): text` - decode a UTF-8 encoded text

`.sub(start: integer[, end: integer]): byte_array` - sub-array (start-inclusive, end-exclusive)

`.toList(): list<integer>` - list of values 0 - 255

Special operators:

- + : concatenation
- [] : element access

6.2.6 range

range(start: integer = 0, end: integer, step: integer = 1) - start-inclusive, end-exclusive (as in Python):

- range(10) - a range from 0 (inclusive) to 10 (exclusive)
- range(5, 10) - from 5 to 10
- range(5, 15, 4) - from 5 to 15 with step 4, i. e. [5, 9, 13]
- range(10, 5, -1) - produces [10, 9, 8, 7, 6]
- range(10, 5, -3) - produces [10, 7]

Special operators:

- in - returns true if the value is in the range (taking step into account)

6.2.7 list

list<T>() - a new empty list

list<T>(list<T>) - a copy of the given list (list of subtype is accepted as well)

list<T>(set<T>) - a copy of the given set (set of subtype is accepted)

.empty(): boolean

.size(): integer

.contains(T): boolean

.containsAll(list<T>): boolean

.containsAll(set<T>): boolean

.indexOf(T): integer - returns -1 if element is not found

.sub(start: integer[, end: integer]): list<T> - returns a sub-list (start-inclusive, end-exclusive)

.str(): text - returns e. g. '[1, 2, 3, 4, 5]'

.add(T): boolean - adds an element to the end, always returns true

.add(pos: integer, T): boolean - inserts an element at a position, always returns true

.addAll(list<T>): boolean

.addAll(set<T>): boolean

.addAll(pos: integer, list<T>): boolean

.addAll(pos: integer, set<T>): boolean

.remove(T): boolean - removes the first occurrence of the value, return true if found

`.removeAll(list<T>): boolean`

`.removeAll(set<T>): boolean`

`.removeAt(pos: integer): T` - removes an element at a given position

`.clear()`

Special operators:

- `[]` - element access (read/modify)
- `in` - returns `true` if the value is in the list

6.2.8 set

`set<T>()` - a new empty set

`set<T>(set<T>)` - a copy of the given set (set of subtype is accepted as well)

`set<T>(list<T>)` - a copy of the given list (with duplicates removed)

`.empty(): boolean`

`.size(): integer`

`.contains(T): boolean`

`.containsAll(list<T>): boolean`

`.containsAll(set<T>): boolean`

`.str(): text` - returns e. g. `'[1, 2, 3, 4, 5]'`

`.add(T): boolean` - if the element is not in the set, adds it and returns `true`

`.addAll(list<T>): boolean` - adds all elements, returns `true` if at least one added

`.addAll(set<T>): boolean` - adds all elements, returns `true` if at least one added

`.remove(T): boolean` - removes the element, returns `true` if found

`.removeAll(list<T>): boolean` - returns `true` if at least one removed

`.removeAll(set<T>): boolean` - returns `true` if at least one removed

`.clear()`

Special operators:

- `in` - returns `true` if the value is in the set

6.2.9 map<K,V>

`map<K,V>()` - a new empty map

`map<K,V>(map<K,V>)` - a copy of the given map (map of subtypes is accepted as well)

`.empty(): boolean`

`.size(): integer`

`.contains(K): boolean`

.get(K): V - get value by key (same as [])

.str(): text - returns e. g. '{x=123, y=456}'

.clear()

.put(K, V) - adds/replaces a key-value pair

.putAll(map<K, V>) - adds/replaces all key-value pairs from the given map

.remove(K): V - removes a key-value pair (fails if the key is not in the map)

.keys(): set<K> - returns a copy of keys

.values(): list<V> - returns a copy of values

Special operators:

- [] - get/set value by key
- in - returns true if a key is in the map

6.2.10 enum

Assuming T is an enum type.

T.values(): list<T> - returns all values of the enum, in the order of declaration

T.value(text): T - finds a value by name, throws en exception if not found

T.value(integer): T - finds a value by index, throws an exception if not found

Enum value properties:

.name: text - the name of the enum value

.value: integer - the numeric value (index) associated with the enum value

---

6.2.11 GTXValue

GTXValue.fromJSON(text): GTXValue - decode a GTXValue from a JSON string

GTXValue.fromJSON(json): GTXValue - decode a GTXValue from a json value

GTXValue.fromBytes(byte_array): GTXValue - decode a GTXValue from a binary-encoded form

.toJSON(): json - encode in JSON format

.toBytes(): byte_array - encode in binary format

6.2.12 record

Functions available for all `record` types:

`T.fromBytes(byte_array): T` - decode from a binary-encoded `GTXValue`

`T.fromGTXValue(GTXValue): T` - decode from a `GTXValue`

`T.fromPrettyGTXValue(GTXValue): T` - decode from a pretty-encoded `GTXValue`

`.toBytes(): byte_array` - encode in binary format

`.toGTXValue(): GTXValue` - encode to a `GTXValue`

`.toPrettyGTXValue(): GTXValue` - encode to a pretty-encoded `GTXValue`

CHAPTER 7

Language specification

7.1 Lexical Rules

7.1.1 Whitespaces and comments

Whitespaces are like in Java: characters for which `java.lang.Character.isWhitespace(c) == true` (e.g. `space`, `tab`, `end-of-line`, etc.).

Comments are like in Java.

- Single-line comment: starts with `//`, ends with end-of-line or end-of-file.
- Multiline comment: starts with `/*`, ends with `*/`.
- Error if there is no `*/` after `/*`.

7.1.2 Identifiers

Identifiers are like in Java: first character has `java.lang.Character.isJavaIdentifierStart(c) == true`, other characters have `java.lang.Character.isJavaIdentifierPart(c) == true`. Simple definition: sequence of letters, digits and underscores (_), first character is not a digit. Can contain non-English letters. Identifiers are case-sensitive.

7.1.3 Keywords:

Keyword is one of the following reserved identifiers:

```
and break class create delete else false for function if in index key limit
list map mutable not null operation or query return set sort true update val
var while
```

- A keyword cannot be used as a general identifier, i.e. as a name of a class, function, variable, etc.

- Longest possible keyword/identifier is taken, i.e. string "format" is an identifier "format", not keyword "for" and identifier "mat".

7.1.4 Operators and delimiters

List of operators and delimiters:

```
! !! != % %= ( ) * *= + += , - -= . / /= : ; < <= = == > >= ? ?. ?: @ [ ] { }
```

- Longest possible operator/delimiter is taken, i. e. string `<=` is a single operator `<=`, not two operators `<` and `=`.

7.1.5 Integer literals

- Decimal: `regex /[0-9]+/`.

Maximum decimal value: `9223372036854775807` (`2^63 - 1`). Error if the value is greater.

- Hex: `regex /0x[0-9A-Fa-f]+/`, e.g. `0x0`, `0xABCD`, etc.

Maximum hex value: `0x7FFFFFFFFFFFFFFF` (`2^63 - 1`). Error if the value is greater.

- Cannot have a letter directly after an integer literal, e. g. `1234X` is an error, not two tokens `1234`, `X`.

7.1.6 String literals

- Enclosed in single (') or double (") quotes.
- There is no difference between single-quoted and double-quoted strings, i. e. 'Hello' and "Hello" are equal string literals.
- Cannot contain an end-of-line character (0x0A), i. e. closing quote must be on the same line as the open quote.
- Error if there is no closing quote on the same line.

7.1.7 Escape sequences

Simple escape sequences: `\b` `\t` `\r` `\n` `\"` `\'` `\\`

Unicode escape sequence: `\u1234` `\uABCD` `\uAbCd` etc. - must have exactly 4 hex digits.

Error if wrong escape sequence is specified (`\` character, but not one of valid escape sequences).

7.1.8 Byte array literals

- Syntax: `x"..."` or `x'...'`, only hex digits (upper or lower case) can be used within quotes.

Examples: `x''` `x"123456"` `x"DeadBeef"`

- Must start with lower-case `x`, not upper-case `X`.
- Must contain an even `(2*N)` number of hex digits (because 1 byte = 2 hex digits).
- Cannot contain escape sequences or end-of-lines.

7.2 Types

7.2.1 General

A type of an attribute, parameter, variable, etc. can be:

- name of a built-in or user-defined type (Identifier)
- nullable type
- tuple type
- collection type

7.2.2 Built-in types

Basic built-in types are:

`boolean` `byte_array` `integer` `json` `range` `text`

Built-in type aliases are:

`name = text` `pubkey = byte_array` `timestamp = integer` `tuid = text`

Type alias `A = T` means that entities (attributes, variables, etc.) of type `A` will effectively have type `T` during compilation.

Special types

Special types cannot be used in code explicitly (in attribute declarations, etc.), but they are used by the compiler internally as types of some expressions.

- Special types are: `unit`, `null`.
- Names of special types cannot be used in code as types. Trying to use "unit" as a type causes an error e. g. "Unknown type name". "null" is a keyword, so using it as a type is a syntax error.

Nullable type

The idea was taken from Kotlin.

Syntax:

NullableType: `Type "?"`

Examples:

- `integer?`
- `list<text>?`

Error if the underlying type is nullable, e. g. `integer??`.

Tuple type

Consists of one or more fields. Each field must have a type and may have a name.

Syntax:

TupleType: `"(" TupleTypeField ( "," TupleTypeField )* ")"`

TupleTypeField: `( Identifier ":" )? Type`

Examples:

- `(integer)`
- `(integer, text)`
- `(x: integer, y: integer)`
- `(p: text, q: byte_array, list<integer>)`

Error if same field name is used more than once.

7.2.3 Collection types

Collection types are: list, set, map.

Syntax:

- `"list" "<" Type ">"`
- `"set" "<" Type ">"`
- `"map" "<" Type "," Type ">"`

Examples:

- `list<integer>`
- `set<text>`
- `map<text, byte_array>`

7.2.4 Subtypes

Purpose: if type `B` is a subtype of type `A`, a value of type `B` can be assigned to a variable of type `A`.

1. `T` is subtype of `T`.
2. `T` is subtype of `T?`.
3. `null` is subtype of `T?`.
4. Tuple type `T1` is subtype of tuple type `T2` if:
   - the number of fields is the same
   - names of corresponding fields are the same (if a field has no name, the other field must have no name)
   - type of each field of T1 is a subtype of the type of the corresponding field of T2

Examples:

- `(integer, text)` is subtype of `(integer, text?)`
- `(integer, text?)` is subtype of `(integer?, text?)`

- `(integer, text?)` is not subtype of `(integer, text)`, because `text?` is not subtype of `text`
- `(x: integer, y: integer)` is subtype of `(x: integer?, y: integer?)`
- `(x: integer, y: integer)` is not subtype of `(p: integer, q: integer)`, because field names differ
- `(integer, text)` is not subtype of `(x: integer, y: integer)`
- `(x: integer, y: integer)` is not subtype of `(integer, text)`

7.3 Classes

Class has a name and zero or more member definitions.

- When a class with name A is defined, A can be used as a type name in the code after the class definition.
- Error if there already is a built-in or user-defined type with same name.
- Class members are: attribute, key, index.

7.3.1 Class syntax

ClassDefinition: `"class" Identifier "{" ClassMemberDefinition* "}"`

```
ClassMemberDefinition :
    AttributeDefinition
    KeyDefinition
    IndexDefinition
```

Example:

```
class user {
    name: text;
    address: text;
    key name;
    index address;
}
```

7.3.2 Attributes

Attribute definition may contain a name, type, default value expression and modifiers (e. g. mutable).

Syntax:

AttributeDefinition: `"mutable"? FieldDefinition ("=" Expression) ";"`

FieldDefinition: `Identifier (":" Type)?`

- If type is not specified, same type as the attribute name is taken (built-in or user-defined). Error if there is no such type.
- Error if there already is another attribute with same name in the same class.
- If default value expression is specified, the type of the expression must be a subtype of the attribute's type.
- Expressions specification will be written later. We can use simplest expressions now for testing: integer literal, string literal, true, false, null, etc.

Examples:

```
name;            // same as "name: name;", there is a built-in type "name"
address: text;
mutable age: integer;
mutable status: text = 'Unknown';
```

7.3.3 Keys, indices

Keys and indices consist of one or more fields.

Syntax:

KeyDefinition: `"key" FieldDefinition ("," FieldDefinition)* ";"`

IndexDefinition: `"index" FieldDefinition ("," FieldDefinition)* ";"`

7.3.4 Handling of fields

- Error if same field name is used more than once within one key/index.
- If there is no attribute with such name, an attribute is added to the class implicitly. The added attribute is not mutable, has no default value.
- If there is an attribute with such name, the key/index field cannot have a type specified.

No error:

`key foo:  integer;`

Error:

```
foo: integer;
key foo: integer;
```

Error if there already is a key/index with same set of fields.

Not an error:

```
index a;
index a, b;
```

Error:

```
index a, b;
index b, a;
```

It does not matter if a key/index is defined before or after an attribute used in it

Code:

```
x: integer;
key x;
```

is equivalent to:

```
key x;
x: integer;
```

Same for field type restrictions: does not matter whether it is before or after the attribute definition

No error:

`key x:  integer;`

No error:

```
x: integer;
key x;
```

Error:

```
x: integer;
key x: integer; // ERROR
```

Error:

```
key x: integer; // ERROR
x: integer;
```

7.4 Operations, Queries, Functions

Let's say that operations, queries and functions are routines. Some rules are common for all routines, while other rules are specific for operations, queries or functions.

7.4.1 Syntax

Module : Definition*

Definition : ClassDefinition | RecordDefinition | RoutineDefinition

RoutineDefinition : Operation | Query | Function

ClassDefinition syntax is covered above.

- Each routine has a name.
- Error when defining a routine, and another routine with the same name already exists.

Built-in functions are also taken into account when checking this rule. (The list of built-in functions will be given in a future chapter.)

7.4.2 Operations

Operation: "operation" Identifier "(" FormalParams? ")" BlockStatement

FormalParams: FieldDefinition ("," FieldDefinition)*

BlockStatement: "{" Statement* "}"

- FieldDefinition syntax is given in chapter 3 (it's the same as for class fields).
- Statement syntax will be given in a future chapter about statements.
- Return type of an operation is "unit". Thus, an operation cannot return a value. Return statement cannot have an expression, even if the expression returns unit:

```
return; // OK

return print('Hello'); // Error, even though print() returns unit.
```

7.4.3 Examples

```
operation foo(user; value: integer) {
    if (value == 0) return;
    update account @ { user } ( score += value );
}
```

7.4.4 Queries

Query: `query` Identifier `(` FormalParams? `)` (`:` Type)? QueryBody

QueryBody: `SimpleBody | ComplexBody`

SimpleBody: = Expression `;`

ComplexBody: BlockStatement

7.4.5 Return type

- A query has a specific return type and always returns a value.
- If return type is not specified explicitly, it is implicitly deducted from return expressions.
- For simple body: return type is the type of the expression.
- Error if the type of the expression is "unit".
- For complex body: return type is the common type of types of all expressions used in return statements.
- Error if there is no common type for return expressions types.
- If explicit return type is specified.
- For simple body: error if the type of the expression is not a subtype of the explicit return type.
- For complex body: error if the type of the expression in a return statement is not a subtype of the explicit return type.
- For complex body: error if there is no return statement.

7.4.6 Examples

```
query getUserCount(company) = (user @* { company }).size(); // Returns integer.

query getUserCount(companyName) {
    if (companyName == "") return 0;
    return (user @* { company.name == companyName }).size();
}

Error: no common return type
query q(x: integer) {
```

(continues on next page)

(continued from previous page)

```
    if (x < 0) return 'Hello';
    return 123;     // Error on this line.
}

Error: actual return type differs from the declared one
query q(): integer = 'Hello';
query q(): integer { return 'Hello'; }
```

7.4.7 Functions

Function: "function" Identifier "(" FormalParams? ")" (":" Type)? FunctionBody

FunctionBody: SimpleBody | ComplexBody

7.4.8 Return type

- If return type is not specified, the return type of the function is "unit".

*Simple body

- The type of the expression must be a subtype of the return type of the function.
- The type of the expression cannot be "unit".

Complex body

- If return type is not specified (thus, it is "unit"), return statements must have no expression (i. e. must use "return;", not "return X;").
- If return type is specified, type of expressions in return statements must be a subtype of the return type.
- Order of function definitions does not matter, all functions defined in a module are visible everywhere in the module.

This allows recursive and mutually-recursive functions:

```
function a(x: integer) {
    if (x > 0) b(x - 1); // b() is visible here, but it is defined below.
}

function b(x: integer) {
    if (x > 0) a(x - 1);
}
```

7.4.9 Common things for routines

- Queries and non-unit functions must always return a value.
- Error if there is no return statement on one of code paths:

```
function f(x: integer): integer {
    print(x);
} // Error: no return statement at all.

function f(x: integer): integer {
```

(continues on next page)

(continued from previous page)

```
    if (x > 0) return x * x;
} // Error: no return statement for one of code paths.

function f(x: integer): integer {
    if (x > 0) {
        return x * x;
    } else {
        print('invalid argument');
    } // Error: no return statement for this branch.
}
```

More formal rules how to check if there is a return value will be given in the chapter on statements (future).

CHAPTER 8

Client

This client tutorial is a continuation on the quickstart "city" example. In this section we illustrate how to send transactions to and retrieve information from a blockchain node running Rell.

8.1 Try the example code

First of all, we need to add a query to Rell source file:

```
query is_city_registered(city_name: text): boolean {
      return (city @? { city_name }) != null;
}
```

Clicking 'Start node' will start a Postchain node in a single-node mode which is convenient for testing. The node builds blocks when there are transactions, or at least once every 30 seconds. It also has REST API we can interact with to submit transactions and retrieve information.

The client code is written in JavaScript, this example uses the NodeJS environment. `postchain-client-example_` can be downloaded using git:

```
git clone https://bitbucket.org/chromawallet/postchain-client-example.git
```

To run it, execute:

```
npm install
node index.js
```

This will create a transaction, sign it, submit to a node. And once transaction is added to a block, client will perform a query.

Now let's see how this client code can be implemented:

8.2 Install the client

We assume you have `nodejs` installed. The client library is called `postchain-client_` and can be installed from npm.

Create an new directory for your test. Open a terminal in the new directory, initialize npm and install the client.

```
npm init -y
npm install postchain-client --save
```

8.3 Connect to the node

To connect to a Postchain node we need to know its REST API URL and blockchain identifier. DevPreview bundle comes with following defaults:

```
const pcl = require('postchain-client');

const node_api_url = "http://localhost:7740"; // using default postchain node REST␣
↪API port

// default blockchain identifier used for testing
const blockchainRID =
↪"78967baa4768cbcef11c508326ffb13a956689fcb6dc3ba17f4b895cbb1577a3";

const rest = pcl.restClient.createRestClient((node_api_url, blockchainRID, 5);
```

Once we set up the information about the the REST Client connection, we can create the gtxClient connection. This in particular, needs to receive the previous REST connection, the blockchainRID in Buffer format and an array the names of the operations that you want call (at the moment this can be left empty):

```
const gtx = pcl.gtxClient.createClient(
    rest,
    Buffer.from(blockchainRID, 'hex'),
    []
);
```

Now that the connection is set, you can start to create transactions and queries.

8.4 Make a transaction (with operations inside)

You need to create the transaction client side, sign it with one or more keypairs, send it to the node and wait for it to be included into a block.

First, let's create the transaction and specify the public key of the person(s) that will sign it. To create a random user keypair on the go you can use `makeKeyPair()` function.

```
const user = pcl.util.makeKeyPair();
const tx = gtx.newTransaction([user.pubKey]);
```

Once it is created is possible call as many operations as you want.

```
tx.addOperation('insert_city', "Tel Aviv");
tx.addOperation('insert_city', "Stockholm");
/* etc */
```

Now, all is left is to sign and post the transaction

```
tx.sign(user.privKey, user.pubKey);
tx.postAndWaitConfirmation();
```

Note: `tx.postAndWaitConfirmation()` returns a promise, and thus can be `await`-ed.

8.5 Query

Queries also make use of `gtx` client.

`gtx.query` accepts as first parameter the name of the query as specified in the module and then an object with as parameter name the variable name as specified in the query module.

E.g:

```
function is_city_registered(city_name) {
  return gtx.query("is_city_registered", {city_name: city_name});
}
```

will work with query specified in the Rell file:

```
query is_city_registered(city_name: text): boolean {
      return (city @? { city_name }) != null;
}
```

Note: `gtx.query(queryName, queryObject)` also returns a promise.

CHAPTER 9

Examples

9.1 Account-based token system

Tokens are the bread & butter of blockchains, thus it is useful to demonstrate how a token system can be implemented in Rell. There are roughly two different implementation strategies:

- Account-based tokens which maintain an updateable balance for each account (which can be associated with a key or an address)
- UTXO-based ones (Bitcoin-style) deal with virtual "coins" which are minted and destroyed in transactions

This section details the account-based implementation. For an example of a UTXO based system see *UTXO-based token system*.

A minimal implementation can look like this:

```
class balance {
      key pubkey;
      mutable amount: integer;
}

operation transfer(from_pubkey: pubkey, to_pubkey: pubkey, xfer_amount: integer) {
          require( is_signer(from_pubkey) );
          require( xfer_amount > 0 );
          require( balance@{from_pubkey}.amount >= xfer_amount );
          update balance@{from_pubkey} (amount -= xfer_amount);
          update balance@{to_pubkey} (amount += xfer_amount);
}
```

There are a few items which should be highlighted in this code. First, let's note that `balance@{from_pubkey}.amount` is simply a shorthand notation for `balance@{from_pubkey} (amount)`.

`update` relational operator combines a relational expression specifying objects to update with a form which specifies how to update their attributes. Attributes are updateable only if they are market as `mutable`.

Note: We don't need to worry about concurrency issues (i.e. that the balance can change after we checked it) because Rell applies operations within a single blockchain sequentially.

But this minimal implementation is not very useful, as there's no mechanism for a wallet to identify payments it receives (without somehow scanning the blockchain, or asking the payer to share the transaction with the recipient). Other blockchains systems might resort to third-party tools and complex protocols to handle this (for example, the Electrum Bitcoin wallet connects to Electrum Servers which perform blockchain indexing). Rell-based blockchains can just use built-in indexing to keep track of payment history. For example, by using the additional `payment` class. To make things more efficient, we also wrap pubkey into `user` class, thus getting:

```
class user { key pubkey; }

class balance {
    key user;
    mutable amount: integer;
}

class payment {
      index from_user: user;
      index to_user: user;
      amount: integer;
      timestamp;
}

operation transfer(from_pubkey: pubkey, to_pubkey: pubkey, xfer_amount: integer) {
          require( is_signer(from_pubkey) );
          require( xfer_amount > 0 );
          val from_user = user@{from_pubkey};
          val to_user = user@{to_pubkey};
          require( balance@{from_user}.amount >= xfer_amount );
          update balance@{from_user} (amount -= xfer_amount);
          update balance@{to_user} (amount += xfer_amount);
          create payment (
                 from_user,
                 to_user,
                 amount=xfer_amount,
                 timestamp=op_context.last_block_time);
}
```

Note: In `create payment (from_user, to_user, ...)` Rell can figure out matching attributes from names of local variables as they match exactly. It is often the case that you can use the same name for the same concept.)

Note: In a future version of Rell it will be possible to timestamp objects automatically using the `log` annotation, with the added benefit that they are then linked to the corresponding transaction and block.

The example above can be easily extended to support multiple types of tokens. For example:

```
class asset { key asset_code; }

class balance {
      key user, asset;
```

(continues on next page)

(continued from previous page)

```
    mutable amount: integer;
}
```

Here we use a composite key to keep track of the balance for each `(user, asset)` pair.

9.2 Chroma Chat

In this section we will write the code for a public chat.

9.2.1 Requirements

The requirements we set are the following:

- There is one admin with an amount of tokens automatically assigned (say 1000000)
- The admin is the first person that registers themselves on the dapp
- Any registered user can register a new user and transfer some tokens to her, after having burned 100 tokens as a fee
- Users are identified by their public key
- Channels are streams of messages belonging to the same topic (which is specified in the name of the channel, e.g. "showerthoughts", where you can send messages with the thoughts you had under the shower).
- Registered users can create channels
- When a new channel is created, only the creator is within the group. She can add any *existing* users. This operation costs 1 token.

9.2.2 Class definition

The structure of it will be:

```
class user { key pubkey; }

class channel {
  key name;
  admin: user;
}

class channel_member { key channel, member: user; }

class message {
  key channel, timestamp;
  index posted_by: user;
  text;
}

class balance {
  key user;
  mutable amount: integer;
}
```

Let's analyse it:

User As said, user is solely identified by her public key

Channel Channels are identified by the name (which ideally reflects the topic of the channel itself) and the user who created it. Note that two channels cannot have the same name (`key`) and that an user can be admin of multiple channels.

Message One message has the text and reference of the user who sent it. Additionally, the channel and timestamp of publication is recorded. Note that `key channel, timestamp` means that only one message can be sent within a channel at given timestamp (but of course several messages on different channels can be recorded at single timestamp).

Balance This is kind of self explanatory: one user has an amount of tokens. Tokens can be spent (or more in general transfered), for this reason the field is marked as `mutable`.

9.2.3 Operations

Init

To initialize the module, we need to have at least one registered user. We don't want the user to call this function once the admin is set (i.e. we don't want users to change the admin). To prevent such event, we create an operation called `init` which verified that no users are registered and, in case of positive response, creates a new admin.

```
operation init (founder_pubkey: pubkey) {
  require( (user@*{} limit 1).len() == 0 );
  val founder = create user (founder_pubkey);
  create balance (founder, 1000000);
}
```

The operation receives a public key as input (note that it does not verify that signer of the transaction is the same specified in input field `founder_pubkey`, meaning you can specify a different public key).

The interesting point is `require( (user@*{} limit 1).len() == 0 );`. Here we retrieve a lists of users with a limit of 1: we get the first user in the table. If there is no user, it will return an empty list. Indeed we check its length and if it's 0 we can proceed in running the operation since there are no users registered.

In the third and fourth line the founder user is created and 1000000 tokens are given to her.

Decrease balance (Function)

For convenience we create a function to decrease a user's balance. We write it because we don't want to duplicate our checks and potentially create bugs.

```
function decrease_balance (user, deduct_amount: integer) {
  require( balance@{user}.amount >= deduct_amount);
  update balance@{user} (amount -= deduct_amount);
}
```

Register a new user

As, said, registered users should be allowed to add new users, with a fee of 100 tokens as specified in `val registration_cost = 100`. We then verify that the signer exists, decrease their balance, create the new user and transfer to him a certain positive amount of tokens.

```
operation register_user (
   existing_user_pubkey: pubkey,
   new_user_pubkey: pubkey,
   transfer_amount: integer
) {
  require( is_signer(existing_user_pubkey) );
  val registration_cost = 100;
  val existing_user = user@{existing_user_pubkey};
  require( transfer_amount > 0 );
  decrease_balance(existing_user, transfer_amount + registration_cost);
  val new_user = create user (new_user_pubkey);
  create balance (new_user, transfer_amount);
}
```

Create a new channel

Registered users can create new channels. Given the public key and the name of the channel, we simply have to verify that she is actual registered user, receive the fee, create the channel (if it already exists, the create command will fail since the name is a `key`) and add that user as chat member.

```
operation create_channel ( admin_pubkey: pubkey, name) {
  require( is_signer(admin_pubkey) );
  val admin = user@{admin_pubkey};
  decrease_balance(admin, 100);
  val channel = create channel (admin, name);
  create channel_member (channel, admin);
}
```

Add user to channel

The admin of a channel (the one who created the channel) can add another user after having paid a fee of 1 token.

So we check once again that the signer is the `admin_pubkey` specified, we decrease the admin balance of 1 token, and we add a new user to the channel via `channel_member`.

```
operation add_channel_member (admin_pubkey: pubkey, channel_name: name, member_
↪pubkey: pubkey) {
  require( is_signer(admin_pubkey) );
  val admin_usr = user@{admin_pubkey};
  decrease_balance(admin_usr, 1);
  val channel = channel@{channel_name, .admin==user@{admin_pubkey}};
  create channel_member (channel, member=user@{member_pubkey});
}
```

Post a new message

People in a channel will love to share their opinions. They can do so with the `post_message` operation where a signer `is_signer(pubkey)` can post a message in the channel `val channel = channel@{channel_name};` he is registered into `require( channel_member@?{channel, member} );` after the payment of a 1 token fee. Note the 3 input parameter nop is not used. We will see why later in this section.

```
operation post_message (channel_name: name, pubkey, message: text, nop: byte_array) {
  require( is_signer(pubkey) );
  val channel = channel@{channel_name};
  val member = user@{pubkey};
  require( channel_member@?{channel, member} );
  decrease_balance(member, 1);
  create message (channel, member, text=message, op_context.last_block_time);
}
```

9.2.4 Queries

It is useful to write data into a database in a distributed fashion, although writing would be meaningless without the ability to read.

Query all channels where a user is registered

Getting the channels one user is registered into is simple, selecting from `channel_member` with the given user's public key.

```
query get_channels(user_pubkey: text) {
  return channel_member@*{.member==user@{byte_array(user_pubkey)}}.channel.name;
}
```

Other simple queries

Likewise we can get the balance from one user.

```
query get_balance(user_pubkey: text) {
  return balance@{user@{byte_array(user_pubkey)}}.amount;
}
```

Retrieve the last message written in a chat, for a channel preview for example. Please note the use of `limit` in order to optimize the query.

```
query get_last_message(channel_name: name) {
  return message@?{channel@{channel_name} } (text = .text, posted_by = .posted_by.
↪pubkey, -sort timestamp = .timestamp) limit 1;
}
```

And the messages sent in one channel sorted from the newest to the oldest.

```
query get_last_messages(channel_name: name) {
  return message@*{ channel@{channel_name} }
      ( text = .text, posted_by = .posted_by.pubkey, -sort timestamp = .timestamp );
}
```

9.2.5 Run it

Assuming we have the `docker-compose.yml` file and we brought it up, we can simply:

- Browse to `localhost:30000`

- Create a new module
- Paste the above code in the `code` section (You can find the full code here).
- Remove all the tests

```
<test>
  <block>
  </block>
</test>
```

- Click `Run tests`
- When the tests are passed, click on `Run Node`

Congratulations! You should now have a running node.

9.2.6 Client side

At this stage we should have a running node with your *freshly made* module.

What about interface it with a classy JS based application?

Well to do it we need the client package, called `postchain-client`

```
const pcl = require('postchain-client);
const crypto = require('crypto');
```

Then we need to declare the address of the REST server (which is ran by the node, default is `7740`) and the blockchainRID of the blockchain (in the dev-preview, this is already set to `78967baa4768cbcef11c508326ffb13a956689fcb6dc3ba17f4b895cbb1577a3` and the number of sockets (5).

We then get an istance of GTX Client, via `gtxClient.createClient` and giving the rest object and blockchain-RID in input. Last parameters is an empty list of operation (this is needed if you don't use Rell language, in fact, you can also code a module with standard SQL or as a proper kotlin/java module).

```
const rest = pcl.restClient.createRestClient("http://localhost:7740/",
↪'78967baa4768cbcef11c508326ffb13a956689fcb6dc3ba17f4b895cbb1577a3', 5)
const gtx = pcl.gtxClient.createClient(
    rest,
    Buffer.from(
        '78967baa4768cbcef11c508326ffb13a956689fcb6dc3ba17f4b895cbb1577a3',
        'hex'
    ),
    []
);
```

Create and send a transaction with the init operation

First thing we probably want is to register and create the admin, we do so calling the `init` function.

```
function init(adminPUB, adminPRIV) {
  const rq = gtx.newTransaction([adminPUB]);
  rq.addOperation('init', adminPUB);
  rq.sign(adminPRIV, adminPUB);
  return rq.postAndWaitConfirmation();
}
```

The first thing we do is to declare a new transaction and that it will be signed by admin private key (we provide the public key, so the node can verify the veracity of transaction.

We add the operation called `init` and we pass as input argument the admin public key. We then sign the transaction with the private key (we specify the public key in order to correlate which private key refers to which public key in case of multiple signatures).

Finally we send the transaction to the node via the method `postAndWaitconfirmation` which returns a promise and resolves once it is confirmed.

Given the following keypair, we can create the admin.

```
const adminPUB = Buffer.from(
    '031b84c5567b126440995d3ed5aaba0565d71e1834604819ff9c17f5e9d5dd078f',
    'hex'
);
const adminPRIV = Buffer.from(
    '0101010101010101010101010101010101010101010101010101010101010101',
    'hex'
);

init(adminPUB, adminPRIV);
```

Create other operations

We can also create a new channel, post a message, invite a user to dapp, invite a user in a channel

```
function createChannel(admin, channelName) {
    const pubKey = pcl.util.toBuffer(admin.pubKey);
    const privKey = pcl.util.toBuffer(admin.privKey);
    const rq = gtx.newTransaction([pubKey]);
    rq.addOperation("create_channel", pubKey, channelName);
    rq.sign(privKey, pubKey);
    return rq.postAndWaitConfirmation();
}

function postMessage(user, channelName, message) {
    const pubKey = pcl.util.toBuffer(user.pubKey);
    const privKey = pcl.util.toBuffer(user.privKey);
    const rq = gtx.newTransaction([pubKey]);
    rq.addOperation("post_message", channelName, pubKey, message, crypto.
↪randomBytes(32));
    rq.sign(privKey, pubKey);
    return rq.postAndWaitConfirmation();
}


function inviteUser(existingUser, newUserPubKey, startAmount) {
    const pubKey = pcl.util.toBuffer(existingUser.pubKey);
    const privKey = pcl.util.toBuffer(existingUser.privKey);
    const rq = gtx.newTransaction([pubKey]);
    rq.addOperation("register_user", pubKey, pcl.util.toBuffer(newUserPubKey),␣
↪parseInt(startAmount));
    rq.sign(privKey, pubKey);
    return rq.postAndWaitConfirmation();
}
```

(continues on next page)

(continued from previous page)

```
function inviteUserToChat(existingUser, channel, newUserPubKey) {
    const pubKey = pcl.util.toBuffer(existingUser.pubKey);
    const privKey = pcl.util.toBuffer(existingUser.privKey);
    const rq = gtx.newTransaction([pubKey]);
    rq.addOperation("add_channel_member", pubKey, channel, pcl.util.
↪toBuffer(newUserPubKey));
    rq.sign(privKey, pubKey);
    return rq.postAndWaitConfirmation();
}
```

Although there is really nothing critical in these functions, there are few things worth noting:

- We expect public and private keys in `hex` format, and we convert them to Buffer with `pcl.util.toBuffer(admin.pubKey);`
- In order to protect the system from replay attacks, the blockchain does not accept transactions which hash is equal to an already existing transaction. This means that an user is not allowed to write the same message twice in a channel since if at day one he writes "hello" the transaction will be something like `rq.addOperation("post_message", the_channel, user_pub, "hello");`, when he will write 'hello' a second time the transaction will be the same and therefore rejected. To solve this problem we add some random bytes via `crypto.randomBytes(32)`, and create a different transaction hash.

Querying the blockchain from the client side

Previously we wrote the queries on blockchain side. Now we need to query from the dapp. To do so we use the previously mentioned `postchain-client` package.

```
// Rell query, reported here for easy look up
// query get_balance(user_pubkey: text) {
// return balance@{user@{byte_array(user_pubkey)}}.amount;
//}

function getBalance(user) {
  return gtx.query("get_balance", {
          user_pubkey: user.pubKey
      });
}
```

As you can see everything is contained into `gtx.query`: the first argument is the query name in the rell module, and the second argument is the name of the expected attribute in the query itself wrapped in an object. The name of the object is the one specified in module and the value, of course, the value we want to send. Please note that `buffer` values must before be converted into hexadecimal strings.

Other queries:

```
function getChannels(user) {
  return gtx.query("get_channels", {
          user_pubkey: user.pubKey
      });
}

function getMessages(channel) {
    return gtx.query("get_last_messages", {channel_name: channel});
}

function getLastMessage(channelName) {
```

(continues on next page)

(continued from previous page)

```
    return gtx.query("get_last_message", {
        channel_name: channelName
    });
}
```

9.3 UTXO-based token system

As an exercise, we can also implement a Bitcoin-style token system.

We first define an unspent transaction output structure:

```
class utxo {
    pubkey;
    amount: integer;
}
```

Then define the transfer operation that roughly follows Bitcoin transaction structure – it has a list of inputs and outputs:

```
operation transfer (inputs: list<utxo>, output_pubkeys: list<pubkey>, output_amounts:␣
↪list<integer>) {
      var input_sum = 0;
      for (an_utxo in inputs) {
          require(is_signer(an_utxo.pubkey));
          input_sum += an_utxo.amount;
          delete utxo@{utxo == an_utxo};
      }
      var output_sum = 0;
      require(output_pubkeys.size() == output_amounts.size());
      for (out_index in range(output_pubkeys.size())) {
          output_sum += output_amounts[out_index];
          create utxo (output_pubkeys[out_index],
                       output_amounts[out_index]);
      }
      require(output_sum <= input_sum);
}
```

There are quite a lot of new constructs used in this example:

- `list<...>` is, obviously, a collection. Besides lists, Rell also supports `set` and `map`, see *Collection types* for syntax.
- in `list<utxo>` utxo object references are physically implemented using integer identifiers which are used interally
- `an_utxo.pubkey` accesses an attribute of an object, which is a database query identical to `utxo@{utxo==an_utxo} (pubkey)`
- variable type is automatically inferred from expression used for initialization. One can also write it like `var output_sum : integer = 0;`
- `delete` operation accepts a relational expression which identifies object(s)
- `.size()` method can be used get the size of a collection
- `for (... in ...)` works both for collections and for ranges of integer values
- `[]` is used to refer to an element of a collection

Note that we perform checks as we go. This is OK because Rell is transactional: if a requirement fails or an error is generated, the whole operation (in fact, the whole transaction) is rolled back. Rell is typically used with a GTX transaction format which supports multiple signers and multiple operations per transaction. Thus it can easily support Bitcoin-style multi-input transactions, atomic token swaps, multi-sig etc.

Now a bit about `delete` operator. Isn't it strange to enable deletion of data from a blockchain?!

Here we aren't deleting data "from a blockchain", we are removing entries from *the current blockchain state*. This is exactly how it works in a Bitcoin node – once entries in an unspent transaction output set are spent, they are deleted. A typical Bitcoin node doesn't keep track of spent transaction outputs.

A system based on Rell (e.g. Postchain or Chromia) works in exactly the same way: raw information about transactions and operations is preserved in a blockchain. The database contains both raw blockchain transactions and processed current state. The current state is what a Rell programmer can work with: he is allowed to do destructive updates and delete entries. These operations do not affect the raw blockchain.