ADAPT

MUFL: a Language for Reusable Decentralized Programming

Technical Brief, Aleksandr Bulkin, CoinFund, 2017-18

Summary

The **MU**table **F**unctions **L**anguage, MUFL (pronounced "muffle"), is the ADAPT framework's specialized programming language for developing reusable code in the context of decentralized systems. MUFL is a database programming language, with emphasis on granular permissioning and modularity. It answers the requirement present in decentralized programming that both data and code reside inside the same database, not at different layers of the technology stack.

MUFL is a functional language. It treats data as functions, the values of which can be explicitly set or changed (mutable functions). At the same time, functional constructs can be treated as data and stored inside the database. This uniformity between data and functional constructs makes possible a concise MUFL API, an interface that can be used to natively implement any data structure or functional primitive and expose it to the language.

Security of access and permissioning is achieved through name hiding and restricted data items akin to capabilities of capability-based addressing. This makes it possible to develop highly reusable code that represents economic or security models.

Background

Reusable programming in the context of blockchain software is difficult, not least because economically-enabled code possesses aspects that do not naturally fit into an abstract interface: value transfer, transaction economics, and complex security provisions. Ethereum's Solidity ecosystem so far has been unable to create a body of reusable code beyond some very simple components, primarily because gas fees require optimizations that cause deisolation, preventing clean reuse.

Smart contract implementations are only meaningful in the context of a consensus blockchain. Yet, decentralized code extends far beyond blockchain smart contracts, but includes other environments, such as clients, state channels, side chains, etc. Moreover, decentralized programming includes algorithms that don't natural fit the smart contract paradigm, such as miner rewards and transaction economics. With MUFL we look to address all these needs, while establishing a mechanism for effective code reuse in all of these contexts.

Blockchain programming is, at its core, database programming. At the same time, traditional database programming tools are not suitable for blockchain. In decentralized programming many of the constraints imposed on blockchain data and the way it is modified have to be verified inside the database, not in separate backend middleware components typical for traditional client-server architectures. Consequently, in order to adapt database programming to blockchain, an approach must be developed that brings data and code into the same layer of the software stack.

MUFL Overview

MUFL brings together concepts of functional programming, database programming, and capability-based security. As a language from the functional family, MUFL operates on functional constructs or lambdas. It also implements access to data through a functional interface, taking an original view that a data structure is also a function, albeit a special one – a function the values of which can be mutated.

As a language that is designed primarily to operate on databases, MUFL is optimized for access to highly structured data, abstracting from the programmer the details of its storage and specific implementation. All data structures accessible from MUFL are provided by native libraries that ensure a programming approach agnostic to specific storage or transmission medium.

In order to enable implementation of a broad range of security models, MUFL provides a convenient mechanism to manage access to data: a type of data called "a point". Points are similar to keys in capability-based addressing. Once created, a point can be passed from one module to another, but cannot be constructed ("forged") from another type of data. Using points to index records in a table ensures that the record can only be retrieved by the module that initially created it, or by code that received the point index from the creator. Alternatively, a point can be stored inside a record, and serve as a key to accessing it. Together with language-level name hiding, this mechanism gives rise to simple implementations of security constraints that are easily reusable.

Syntactically, MUFL uses the Curry notation for both function calls and data access (since data is also seen as functions). Like many functional languages it enforces one-time name binding. It then relies on mutable functions to represent mutating values. Additionally, it provides a flexible syntax for iterating over complex functional constructs. This syntax for iterative constructs serves as the query language for the framework. These and other concepts are detailed in subsequent sections.

Language Basics

We now present the basic language tutorial.

Basic Name Binding

```
a IS 1.                 // Assign value 1 to name 'a'
b IS a.                 // Assign value of 'a' to name 'b'
c IS "Alex".            // Assign string to name 'c'
e IS NIL.               // NIL is a special value
f = TRUE.               // TRUE is a special value
                        // '=' is synonymous to 'IS'
```

Next, we present basic mutable functions and the way data is mutated.

Basic Mutations

```
d IS NEW(dictionary integer any). // Bind name 'd'
                        // to a new dictionary function
                        // that maps integer numbers to any type
d 1 -> "hello".         // assign string to element '1' of 'd'
print (d 1) "\n".       // prints ''hello''

d 2 -> NEW(dictionary integer string).
                        // element (d 2) is now
                        // also a function
d 2 1 -> "bye".
print (d 2 1) "\n".     // prints ''bye''
d 2 1 -> NIL.           // deletes the element
```

ADAPT

MUFL: a Language for Reusable Decentralized Programming

Technical Brief, Aleksandr Bulkin, CoinFund, 2017-18

Modules and Functions

```
MODULE test {    // defines a new module
  HIDDEN {       // defines a block of private names
    // line 6 defines a table and binds it to a hidden name
    // NEW(A,B) constructs type A with constructor argument B
    // '()' is an alternative syntax for 'NIL'
    tbl IS NEW(table (integer 4) (fixed_string 25), ()).
  }

  insert IS FUNC TAKES // defines a function
    num TYPE integer,  // two parameters
    val TYPE string
  DOES {
    tbl num -> val.    // function body mutates 'tbl'
  }

  lookup IS FUNC TAKES
    num TYPE integer
  RETURNS string
  DOES {
    RETURN (tbl num).
  }
}

test::insert 1 "hello".
```

Arrays, Sets, Dictionaries, and Patterns

```
arr IS ["alex", 2, 3].   // creates an array
print (arr 0) "\n".      // prints 'alex'

// creates a dictionary:
dict IS ("a"->"hello", "b"->"world").
print (dict "b") "\n".   // prints 'world'

// creates a set of values:
company IS ("Alex", "John", "James").
print (company "Alex") "\n". // prints 'TRUE'
print (company "Bob") "\n". // prints 'NIL'

// dictionaries and sets can mix:
mixed IS ("A", "B"->1).
empty IS (,).              // syntax for empty set

// binding multiple names in the same statement:
BIND [a,b] TO [1,2].       // same as 'a IS 1. b IS 2.'
BIND ("a"->c) TO ("a"->1). // same as 'c IS 1.'
```

Iterating Over Data

```
a IS [["a", "b", "c"],["d", "e", "f"]].

// print out every element of the array
SCAN a BIND i THEN BIND j HAS value DO {
  print "a[" i "," j "]=" value "\n".
}

// print out every element [*,2]
SCAN a BIND i THEN WITH 2 HAS value DO {
  print "a[" i ",2]=" value "\n".
}

// print out every element on the diagonal
SCAN a BIND i THEN BIND j HAS value WHERE i==j DO {
  print "a[" i "," i "]=" value "\n".
}
```

Using Integer Sequences

```
seq1 IS (1..10).
seq2 IS (1..10 ++2).
seq3 IS (10..1 --1).

SCAN (10..20 ++2) BIND i HAS j DO {
  print i "*2 + 10 == " j "\n".
}

a IS [["a", "b", "c"],["d", "e", "f"]].

// nested scan binds the value for the parent scan in sequence
SCAN a BIND i THEN SCAN (0..1) END HAS value DO {
  // for the first two elements of every
  // sub-array of 'a'.
}
```

Finally, we give a brief tutorial-style overview of MUFL metaprogramming features.

Metaprogramming and Types

```
MODULE x {
  META {
    // define types
    short_string IS fixed_string 25. // 'fixed_string' is a
    long_string IS fixed_string 256. // type-function
    db_rec IS record ( "name"->short_string,
                       "address"->long_string).
    id IS integer 8.
    data IS table id db_rec.
  }

  tbl_data IS NEW(data, ()).

  lookup_func IS FUNC TAKES
    index TYPE id
  RETURNS db_rec
  DOES {
    RETURN (tbl_data id).
  }

  META {
    // query the return type of a function
    t_ret IS meta::get_return_type lookup_func.
  }
}

// generic modules take type-arguments
GENERIC MODULE y TAKES
  t TYPE meta::type
IS
{
  f IS FUNC TAKES
    param TYPE t
  DOES { ... }
}

MODULE z {
  // generic modules get instantiated inside other modules
  MODULE y_of_int INSTANTIATES y integer.

  // and can be accessed
  y_of_int::f 22.
}
```

Tokens and Cells

We now demonstrate how to use MUFL to implement reusable decentralized code. We provide an implementation of a supply of tokens wrapped into cryptographically-protected wallets. The resulting database exposes a limited API which consists of transactions to create a wallet and to transfer tokens from one wallet to another. It is easy to imagine additional improvements to allow multiple kinds of tokens, and to remove the need to create a wallet explicitly. However, for purposes of brevity we omit these and many other possible features.

The reusability of code is achieved by separating the implementation into three modules: `cells`, which implements the token supply broken up into small containers that enable token transfer, but have no security provisions; `gates`, which implements generalized cryptographic protection of some feature; and `main`, which combines the first two modules and defines the external API of the database.

If we wanted to implement multi-signature wallets, we would create a new module, `multisig_gates`, which would implement the functionality necessary to trigger a target transaction by multiple signatories. Module `cells` would stay completely intact, as cells don't care whether they are owned by a single-signature gate or a multi-signature one.

Here is the implementation of `cells`:

MODULE cells

```
1  MODULE cells {
2      META {
3          // define internal types of the module
4          t_amount IS decimal.
5          t_cell_id IS unique_id,
6          t_cell IS records::type (
7              "holdings" -> t_amount,
8              "dkey"     -> t_cell_id,
9              "wkey"     -> point
10         ).
11
12         t_coll_cells IS table t_cell_id t_cell.
13         t_wkey IS point.
14         t_coll_cells_by_wkey IS table t_wkey t_cell.
15     }
16
17     HIDDEN {
18         // Hidden information inside the module
19         tbl_cells IS NEW ( t_coll_cells, () ).
20         tbl_cells_by_wkey IS NEW ( t_coll_cells_by_wkey, () ).
21
22         create_cell_internal IS FUNC TAKES
23             cell_id TYPE t_cell_id
24             amount TYPE t_amount,
25         RETURNS t_wkey
26         DOES {
27             IF (tbl_cells cell_id ~= NIL) ABORT.
28
29             wkey IS NEW( t_wkey, () ).
30
31             cell IS NEW t_cell (
32                 "holdings"      -> amount,
33                 "dkey"          -> cell_id,
34                 "wkey"          -> wkey
35             ).
36
37             tbl_cells cell_id -> cell.
38             tbl_cells_by_wkey wkey -> cell.
39             RETURN wkey.
40         }
41     }
```

MODULE cells (cont.)

```
36     transfer IS FUNC TAKES
37         from    TYPE t_wkey,
38         to      TYPE t_cell_id,
39         amount  TYPE t_amount
40     DOES {
41         IF amount < 0 THEN ABORT.
42
43         from_cell IS tbl_cells_by_wkey from.
44         IF from_cell == NIL THEN ABORT.
45
46         to_cell IS tbl_cells to.
47         IF to_cell == NIL THEN ABORT.
48
49         IF from_cell "holdings" - amount < 0 THEN ABORT.
50         IF to_cell "holdings" + amount < to_cell "holdings"
51             THEN ABORT.
52
53         from_cell "holdings" -> from_cell "holdings" - amount.
54         to_cell "holdings" -> to_cell "holdings" + amount.
55     }
56
57     get_dkey IS FUNC TAKES
58         wkey TYPE t_wkey
59     RETURNS t_cell_id
60     RETURN tbl_cells_by_wkey t_wkey "dkey".
61
62     create IS FUNC TAKES
63         cell_id TYPE t_cell_id
64     RETURN create_cell_internal cell_id 0.
65
66     first_supply IS NEW( array t_wkey ).
67     first_supply 0 -> create_cell_internal (NEW t_cell_id) 1000000.
68 }
```

This module constructs a table of containers (cells) that hold tokens. Each cell is represented by a database record. The module implements a function to transfer tokens from one cell to another called `transfer`.

Cells are stored in a table `tbl_cells` and have an additional index `tbl_cells_by_wkey` (lines 17-18). Each cell can be accessed for deposit or for withdrawal. Deposit access requires the cell's address, a unique number called `dkey`, which stands for "deposit key", sometimes referred to as `cell_id`.

The `wkey` ("withdrawal key") of a cell is a protected reference that permits the code that possesses it to withdraw funds from the cell. `wkey` is of type `point` which is an unforgeable value. The creator of the cell gets its `wkey` and may pass it to other code that provides additional features, such as the module `gates` discussed next.

We notice that all data is hidden inside the model via private (`HIDDEN`) names. The language provides no possible mechanism to circumvent this restriction. Consequently, as long as the module does not expose the values of hidden names to callers or clients, its data will be protected from attacks.

Lines 59-60 are a hacky way to construct the supply of the token. We do it this way for brevity. Nevertheless, this is a valid and secure way to achieve the desired effect: module `main` presented below will take over the `wkey` of `first_supply`, preventing anyone else from accessing it illegally. In a realistic production environment the database would expose a way for clients to compile and run code, either openly, or by a governing vote. The solution presented here (as seen in module `main`) will prevent such code from containing a hidden attack that would take over the token supply.

MUFL: a Language for Reusable Decentralized Programming

Technical Brief, Aleksandr Bulkin, CoinFund, 2017-18

Gates

Module `gates` provides cryptographic protection for restricted functionality. In the implementation of `cells` above, we saw that a cell can be accessed by any code holding its `wkey`. Obviously, it is not possible to pass the value of a cell's `wkey` to a user or agent external to the database where the code is running. Doing so would require converting the `wkey` to a binary value and receiving this value from outside of the database would require constructing the `wkey` from binary. But this would mean that anyone in possession of the binary value of `wkey` would then be able to withdraw tokens from a cell. `wkey` prevents this possibility as it cannot be constructed from a binary value.

Instead, in order to provide cryptographic security, we implement module `gates`. This is a generic module that takes any lambda and hides it inside a data structure called a `gate`. The `gate` also contains a public key of the user that is allowed to invoke the lambda. Given this functionality, the creator of a `cell` will be able to wrap the `transfer` function into a lambda (binding the `cell`'s `wkey` parameter) and create a gate that hides this lambda, ensuring that cryptographic signature is required to invoke it.

In this module we see the metaprogramming features of MUFL. The module is `GENERIC`, meaning that it takes a type-argument `t`. This argument defines the type of the parameter accepted by the lambda hidden inside the gate. It is possible for generic modules to take any compile-time data structure, including an array of types.

MODULE gates

```
GENERIC MODULE gates TAKES
    t_argument TYPE meta::type, // generic module arguments
IS {
    META {
        t_gate_id IS unique_id.
        t_gate IS records::type (
            // signature of the lambda
            "lambda"    -> function [t_argument] NIL,
            "signatory" -> cryptography::t_public_key,
        ).

        t_coll_gates IS table t_gate_id t_gate.
    }

    HIDDEN {
        tbl_gates IS NEW ( t_coll_gates). // table of gates
    }

    create IS FUNC TAKES
        lambda TYPE function [t_argument] NIL,
        signatory TYPE cryptography::public_key
    DOES {
        // constructs a new gate given the lambda
        // and the public_key
        gate IS NEW (t_gate, (
            "lambda"      -> lambda,
            "signatory"   -> signatory,
        )).

        gate_id IS NEW (t_gate_id, ()).
        tbl_gates gate_id -> gate.

        RETURN gate_id.
    }
```

MODULE gates (cont.)

```
    invoke IS FUNC TAKES
        gate_id TYPE t_gate_id,
        arguments TYPE t_argument,
        signature TYPE t_signature
    DOES {
        // invokes the protected functionality
        // given the cryptographic signature
        gate IS tbl_gates gate_id.
        signature_valid IS
            ::cryptography::verify_signature
                (gate "signatory") arguments signature.

        IF NOT signature_valid THEN ABORT.
        // finally, invoke the lambda
        gate "lambda" arguments.
    }
}
```

The Main Module

So far we have only shown library modules. What follows is a module specific to this database which combines the functionality above into a fully functional database that has wallets and exposes to clients the ability to transfer funds between them.

MODULE main

```
MODULE main {
    // First, define parameters to the transfer transaction
    META {
        t_wspec IS records::type (
            "operation" -> string 20, // Value "::cells::transfer"
                                      // will be required here
            "from"      -> t_cell_id,
            "to"        -> t_cell_id,
            "amount"    -> t_amount
        )

        // Types of parameters to the transaction API
        // of this database

        // Type accepted by the create_wallet transaction
        t_API_create_wallet IS records::type (
            "signatory" -> cryptography::public_key
        ).

        // Type accepted by the transfer transaction
        t_API_transfer IS pair t_wspec cryptography::signature.

       // now instantiate module ::gate with the right type
       MODULE gates INSTANTIATES ::gates t_wspec.

       t_coll_wallet_gates IS
            table t_cell_id gate::t_gate_id.
    }

    HIDDEN {
        // Construct a table of wallets
        tbl_wallet_gates IS NEW (t_coll_wallet_gates, ()).
    }

    // this defines the API surface of the current database
    // First, an API point for the create_wallet function
    create_wallet_API IS NEW
        (API_entry_point t_API_create_wallet, ()).
    // Next, an API point for the transfer function
    transfer_API IS NEW (API_entry_point t_API_transfer, ()).
```

Module Main (cont.)

```
    // function that will be bound to the create_wallet API point
    create_wallet IS FUNC TAKES
        signatory TYPE cryptography::t_public_key
    DOES {
        // First it creates a cell
        wkey IS cells::create 0 NEW( cells::t_cell_id, ()).
        // Then, wrap the cell's wkey into a lambda
        transfer_func IS FUNC TAKES
            args TYPE t_wspec
        DOES {
            IF t_wspec "operation" ~= "::cells::transfer"
                THEN ABORT.
            // wkey is bound from the context above
            IF t_wspec "from" ~= cells::get_dkey wkey THEN ABORT.
            transfer wkey (args "to") (args "amount").
        }

        // creates a gate, giving it the transfer_func
        gate IS gate::create transfer_func signatory.
        tbl_wallet_gates signatory -> gate.
    }

    // Bind the API point for creating a wallet
    create_wallet_API "invoke" ->
        FUNC TAKES
            arg TYPE external_packet
        DOES {
            // convert packet received from the client
            // into an internal and safe representation
            transaction IS t_API_create_wallet "convert" arg.
            IF transaction == NIL THEN ABORT.
            signatory IS transaction "signatory".
            create_wallet signatory.
        }

    // Bind the API point for transferring tokens
    transfer_API "invoke" -> FUNC TAKES
        arg TYPE external_packet
    DOES {
        transaction IS t_API_transfer "convert" arg.
        IF transaction == NIL THEN ABORT.

        BIND wspec, signature TO transaction.
        from IS wspec "from".

        gate IS tbl_wallet_gates from.
        IF gate == NIL THEN ABORT.

        gates::invoke gate wspec signature.
    }

    // Finally, create the first supply of the token and bind
    // the cell to the owner's address
    HIDDEN {
        create_first_supply IS FUNC TAKES
            signatory TYPE cryptography::t_public_key
        DOES {
            wkey IS cells::first_supply 0.
            cells::first_supply 0 -> NIL.
            transfer_func IS FUNC args TYPE t_wspec DOES {
                IF t_wspec "from" ~= cells::get_dkey wkey
                    THEN ABORT.
                transfer wkey (args "to") (args "amount").
            }

            gate IS gate::create transfer_func signatory.
            tbl_wallet_gates signatory -> gate.
        }

        // This is the public address of the owner
        create_first_supply 0x02134abf80de234....
    }
}
```

Discussion

We have shown an implementation of a decentralized database that supports a native token and allows payments from one participant to another. Obviously, the implementation is lacking in some important respects:

1. Only one type of tokens is supported
2. There are no provisions for paying transaction fees
3. The initial supply of tokens is constructed at database creation time
4. There are no provisions for running additional code or extending the database with new functionality
5. It requires an additional step to construct a wallet, as the address is not the same as the owner's public key.

However, the current implementation is about 200 lines of code (including comments), of which only a few dozen lines are non-reusable, as most of the `main` module could also be meaningfully made into a library. Provided that there are no bugs, this implementation is fully secure from attacks, even if the database were to allow transactions that install new modules.

A basic MUFL implementation of a generic atomic swap for ADAPT databases similarly clocks in at about 500 lines of code and is both reusable and extensible. In order to use atomic swaps, any ADAPT database can easily install the required modules at creation time, and will require very little additional work on part of the database creator.

The `gates` module demonstrates a generic approach to implementing security around clients' actions. In a similar vein, one can easily imagine a generic module that implements proposals and voting. Such simple governance functionality can then be applied to any critical activity inside a decentralized database, for example it could control the process of installing additional modules.

Modules expose various features as API access points to the database. This creates a flexible mechanism for ADAPT databases to define or restrict the functionality they offer to clients, allowing developers to tailor decentralized databases to their specific use cases much better and with more security than existing generic networks.

Consider now the process of developing and testing such a database. No complicated infrastructure is required to test this code, as it will work equally well in both centralized and decentralized environments. In order to build and test an ADAPT database, a single local node is needed, the console of which will provide features for building and testing a database. When the testing is completed, the developer can take the image of the resulting database and launch it as a root state of a decentralized network.

Conclusion

ADAPT is a platform designed to increase the speed of innovation in decentralized programming. MUFL is the programming language that enables reuse and flexibility of decentralized software. It will work in a broader range of contexts than existing smart contract languages and will drive the accumulation of a rich codebase of decentralized and economically-enabled open source software components.