Functional Specification for The Fafel Programming Language

Introduction

Fafel is a functional smart contract language for the Ethereum Virtual Machine (EVM). It stands for “Finally, a Functional Ethereum Language”. It is designed to provide a simple and intuitive way to write smart contracts that are easy to understand and verify.

Design Goals

The design of Fafel is guided by the following goals:

- Simplicity: Fafel is designed to be a simple and intuitive language that is easy to understand and reason about.
- Predictability: Fafel uses a pure functional programming model, which means that functions have no side effects and always return the same output for a given input. This makes it easier to reason about the behaviour of Fafel contracts and helps to prevent unintended consequences.
- Safety: Fafel has a strong static type system, which ensures that the type of every expression is known at compile time and that type errors are caught before the code is executed. This helps to prevent unintended consequences and makes it easier to reason about the behaviour of Fafel contracts.

Language Overview

Fafel is a compiled language, written in Haskell, and it compiles to the Yul intermediate language used by the Ethereum platform. This allows Fafel contracts to be executed on the EVM and interact with other contracts and applications on the Ethereum network.

Fafel uses a pure functional programming model, which means that functions have no side effects and always return the same output for a given input. This makes it easier to reason about the behaviour of Fafel contracts and helps to prevent unintended consequences.

In Fafel, the state of each contract is described using a type that incorporates all of the state variables of the contract. This allows the compiler to enforce rules about how the state can be modified, ensuring that the contract behaves predictably and providing additional safety guarantees for developers.

Fafel uses nested expressions instead of statement sequencing to update the state within contract functions. This makes the code easier to understand and reason about, as well as enforcing rules about how the state can be modified.

Syntax

Fafel has the following grammar:

```
<program> ::= {<statement> | <comment>}
<statement> ::= <contract> | <type> | <function> | <view-function>
<contract> ::= "contract" <identifier> "{" <state-type> {<function>} {<view-function>}
"}"
<type> ::= "type" <identifier> "=" <data-type>
<state-type> ::= "state" "{" <state-variable> {"," <state-variable>} "}"
<state-variable> ::= <identifier> ":" <data-type>
<function> ::= "def" <identifier> "(" <args> ")" "->" <state-type> "=" <expr>
<view-function> ::= "view" <identifier> "(" <args> ")" "->" <state-variable> "=" <expr>
<args> ::= <identifier> ":" <data-type> {"," <identifier> ":" <data-type>}
<data-type> ::= <atomic-type> | <list-type> | <map-type>
<atomic-type> ::= "int" | "uint" | "bool" | "bytes" | "address"
<list-type> ::= "[" <atomic-type> "]"
<map-type> ::= "{" <atomic-type> ":" <atomic-type> "}"
<expr> ::= <literal> | <variable> | <function-call> | <contract-creation> | <if-expr> |
<in-expr> | <map-expr> | <binary-expr> | <unary-expr> | <compare-expr> | <assign-expr>
<literal> ::= <integer> | <bool> | <bytes> | <address>
<variable> ::= <identifier>
<function-call> ::= <identifier> "(" <args> ")"
<contract-creation> ::= <identifier> "new" "(" <args> ")"
<if-expr> ::= "if" <expr> "then" <expr> "else" <expr>
<in-expr> ::= <expr> "in" <expr>
<map-expr> ::= "map" "(" <identifier> "," <expr> ")" "to" <expr>
<binary-expr> ::= <expr> <binary-operator> <expr>
<binary-operator> ::= "+" | "-" | "*" | "/" | "and" | "or"
<unary-expr> ::= <unary-operator> <expr>
<unary-operator> ::= "not"
<compare-expr> ::= <expr> <compare-operator> <expr>
<compare-operator> ::= "<" | ">" | "<=" | ">=" | "in"
<assign-expr> ::= <variable> "=" <expr>
<comment> ::= "--" {<any character>}
<identifier> ::= <letter> {<letter> | <digit>}
<letter> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
| "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<address> ::= "0x" {<hexadecimal-digit>}
<hexadecimal> ::= "0x" {<hexadecimal-digit>}
<hexadecimal-digit> ::= <digit> | "a" | "b" | "c" | "d" | "e" | "f"
<bool> ::= "true" | "false"
<bytes> ::= <hexadecimal>
```

Built-in Types

Fafel has a number of built-in types, including:

- Int: An integer type for representing whole numbers
- Uint: An unsigned integer type for representing non negative numbers
- Bool: A boolean type for representing true/false values
- Bytes: A byte array type for representing binary data
- Address: An address type for representing Ethereum addresses
- Map: A map type for representing key-value pairs
- List: A list type for representing ordered collections of values

Reserved Keywords

- contract: Used to define a contract
- def: Used to define a function
- type: Used to define a type
- if: Used to define conditional statements
- in: Used to check if a value exists in a collection
- map: Used to apply a function to each element of a collection
- return: Used to return a value from a function
- view: Used to mark a function as non state-modifying

Mathematical Operations

Fafel also has support for mathematical operations, such as arithmetic and logical operations. These operations can be used within contract functions to perform calculations and make decisions based on the input provided by users.

State Type & Functions

Fafel has a state type which incorporates all of the state variables in the contract. Functions that modify state take an input state and return an updated output state.

View Functions

Fafel allows developers to mark functions as view functions if they do not modify the state of the contract. These functions return state variables, allowing getter functions to return only single state variables. This allows the compiler to enforce rules about how the state can be modified, ensuring that the contract behaves predictably and providing additional safety guarantees for developers.

Examples

Simple contract to store a value and add to it:

```
contract MyContract {
  state {
```

```
    value: int
  }

  def add(x: int) -> state {
    return state {
      value = value + x
    }
  }
}
```

Simple contract to check if a value is equal to a state value:

```
contract MyContract {
  state {
    value: int
  }

  view isEqualTo(x: int) -> bool {
    if value == x then return true else return false
  }
}
```

Conclusion

Fafel is a simple and intuitive language for writing smart contracts on the Ethereum platform. Its strong static type system, pure functional programming model, and support for mathematical operations make it easy to write contracts that are predictable, safe, and easy to verify.
Fafel is designed to be a base upon which to build more complex and sophisticated smart contract languages in the future. It should be noted that the implementation of Fafel is subject to change. By providing a simple and intuitive language that is easy to understand and reason about, Fafel aims to make it easier for developers to write and verify smart contracts on the Ethereum platform.