Simple smart contract developed byt aiken-lang new features and function

Example: Token Contract

This example demonstrates a basic token contract using Aiken-lang's enhanced features such as pattern matching, immutable data structures, concurrency primitives, and improved error handling.

```
    // Function to check balance
    public func balanceOf(address: String) -> Int {
        let state = getState()
        Map.lookup(address, state.balances) |> default 0
    }

    // Function to get total supply
    public func getTotalSupply() -> Int {
        let state = getState()
        state.totalSupply
    }
}


// Define the Token contract
contract Token {
```

```
    // State of the contract
    type State = {
        balances: Map<String, Int>, // Mapping from user addresses to token balances
        totalSupply: Int            // Total supply of tokens
    }

    // Initial state
    let initialState = {
        balances: Map.empty(),
        totalSupply: 0
    }

    // Function to create new tokens
    public func mint(recipient: String, amount: Int) -> Result<String, String> {
        let state = getState()

        // Validate the amount
        if amount <= 0 {
            return Err("Amount must be positive")
        }

        // Update state
        let newBalances = Map.insert(recipient, amount, state.balances)
        let newState = { state with
            balances = newBalances,
            totalSupply = state.totalSupply + amount
        }

        setState(newState)
        Ok("Tokens minted successfully")
    }

    // Function to transfer tokens
    public func transfer(from: String, to: String, amount: Int) -> Result<String, String> {
        let state = getState()

        // Validate the transfer
        match Map.lookup(from, state.balances) with
        | None => Err("Sender address not found")
        | Some(balance) =>
            if balance < amount {
                return Err("Insufficient balance")
            }
```

```
            // Update balances
            let updatedBalances = Map.insert(from, balance - amount, state.balances)
            let finalBalances = Map.insert(to, (Map.lookup(to, updatedBalances) |> default 0) +
amount, updatedBalances)
            let newState = { state with
                balances = finalBalances
            }

            setState(newState)
            Ok("Transfer successful")
    }
```

Explanation of the Smart Contract

1. **State Management**: The `State` type represents the contract's state, including token balances and total supply. This state is immutable and updated through function calls.
2. **Pattern Matching**: Used in the `transfer` function to handle different cases for address lookup. It gracefully handles situations where the sender address may not be found.
3. **Immutable Data Structures**: The contract uses immutable data structures (like `Map`) to ensure that state updates do not alter the original state but rather create a new state.
4. **Error Handling**: Improved error handling is used to validate inputs and handle various error conditions, such as insufficient balance or invalid amounts.
5. **Concurrency Primitives**: Although not explicitly shown in this simple example, Aiken's support for concurrency could be leveraged in more complex contracts involving asynchronous operations or multi-threaded environments.
6. **Meta-Programming**: The contract doesn't use meta-programming explicitly in this example, but Aiken's meta-programming features could be utilized for more dynamic contract generation or code optimizations.

Deploying and Interacting with the Contract

To deploy and interact with this contract, you would typically use a blockchain platform that supports Aiken-lang. This process involves compiling the smart contract to bytecode, deploying it to the blockchain, and then calling its functions to interact with it. Make sure to follow the specific instructions provided by the Aiken-lang and blockchain platform documentation.