Aiken 59 Commits this week  
  
  
Link: <https://cardanoupdates.com/aiken-lang/aiken>

### Aiken-lang New Functions:

Aiken is a functional smart contract language for Cardano, designed to improve simplicity and usability compared to Plutus. Below, I'll provide a breakdown of potential new functions in Aiken, explaining their code structure, how they could be implemented, and their purpose.

### 1. \*\*`try...catch` for Enhanced Error Handling\*\*

A `try...catch` construct allows developers to handle errors more gracefully.

#### Code Example:

```aiken

fn safe\_divide(a: Int, b: Int) -> Result<Int, String> {

if b == 0 {

Err("Division by zero error")

} else {

Ok(a / b)

}

}

fn calculate() -> Int {

match safe\_divide(10, 0) {

Ok(result) => result,

Err(error) => {

trace("Error occurred: ", error);

0

}

}

}

```

#### Explanation:

- \*\*`Result` type\*\*: A standard in functional programming that represents either a success (`Ok`) or an error (`Err`).

- \*\*Error Handling\*\*: The `match` expression handles both success and error cases, ensuring safe execution without crashes.

### 2. \*\*Performance Profiling Function: `profile\_execution`\*\*

A function that measures and returns the execution time and gas usage of a smart contract.

#### Code Example:

```aiken

fn profile\_execution<T>(f: fn() -> T) -> (T, Time, Gas) {

let start\_time = current\_time();

let start\_gas = gas\_used();

let result = f();

let end\_time = current\_time();

let end\_gas = gas\_used();

(result, end\_time - start\_time, end\_gas - start\_gas)

}

fn expensive\_computation() -> Int {

-- some expensive operations --

100

}

fn test\_profile() -> (Int, Time, Gas) {

profile\_execution(expensive\_computation)

}

```

#### Explanation:

- \*\*`profile\_execution`\*\*: This function profiles another function by tracking its execution time and gas consumption.

- \*\*Utility\*\*: Helps developers understand the performance impact of their contract, useful for optimization.

### 3. \*\*Multi-Signature Verification with `verify\_multisig`\*\*

A function to verify multi-signature schemes, allowing contracts to require multiple signatures for validation.

#### Code Example:

```aiken

fn verify\_multisig(pub\_keys: List<PubKeyHash>, signatures: List<Signature>, required\_sigs: Int) -> Bool {

let valid\_sigs = filter(is\_valid\_signature, pub\_keys, signatures);

length(valid\_sigs) >= required\_sigs

}

fn is\_valid\_signature(pub\_key: PubKeyHash, signature: Signature) -> Bool {

verify\_signature(pub\_key, signature)

}

```

#### Explanation:

- \*\*`verify\_multisig`\*\*: Verifies if enough valid signatures are present to satisfy the multi-signature requirement.

- \*\*Use Case\*\*: Commonly used in DAO voting, wallets with shared control, or other multi-party applications.

### 4. \*\*Oracle Integration with `fetch\_data\_from\_oracle`\*\*

A function that fetches and verifies external data from an oracle.

#### Code Example:

```aiken

fn fetch\_data\_from\_oracle(oracle\_id: PubKeyHash, data\_hash: Hash, signature: Signature) -> Option<Data> {

if verify\_oracle(oracle\_id, data\_hash, signature) {

Some(get\_oracle\_data(data\_hash))

} else {

None

}

}

fn verify\_oracle(oracle\_id: PubKeyHash, data\_hash: Hash, signature: Signature) -> Bool {

verify\_signature(oracle\_id, signature, data\_hash)

}

```

#### Explanation:

- \*\*`fetch\_data\_from\_oracle`\*\*: Retrieves and verifies data from an external oracle, ensuring the authenticity of the information.

- \*\*Use Case\*\*: Enables smart contracts to interact with off-chain data (e.g., real-world events, price feeds).

### 5. \*\*State Management Function: `initialize\_state`\*\*

A state initialization function for setting up the contract’s initial state.

#### Code Example:

```aiken

fn initialize\_state() -> Datum {

Datum {

balance: 0,

owner: PubKeyHash("0x12345"),

is\_active: True

}

}

```

#### Explanation:

- \*\*`initialize\_state`\*\*: Sets up a contract’s initial state. This is important for contracts that maintain complex state across transactions.

- \*\*Use Case\*\*: Used for initializing stateful contracts, such as games or DeFi protocols.

### 6. \*\*Formal Verification Support Function: `prove\_contract\_correctness`\*\*

This function provides a proof of contract correctness, ensuring the contract satisfies a given specification.

#### Code Example:

```aiken

fn prove\_contract\_correctness(input: Int) -> Bool {

input > 0

}

fn validate\_input(input: Int) -> Bool {

prove\_contract\_correctness(input) && input < 100

}

```

#### Explanation:

- \*\*`prove\_contract\_correctness`\*\*: Ensures the contract follows a provable rule, enhancing its security.

- \*\*Use Case\*\*: Critical for high-stakes applications where contracts need to be mathematically proven to work as intended.

### 7. \*\*Cross-Chain Interaction Function: `call\_crosschain\_contract`\*\*

A function that allows smart contracts to interact with other blockchains.

#### Code Example:

```aiken

fn call\_crosschain\_contract(chain\_id: Int, contract\_address: Address, data: Data) -> Bool {

if verify\_chain(chain\_id) {

send\_to\_contract(chain\_id, contract\_address, data)

} else {

False

}

}

```

#### Explanation:

- \*\*`call\_crosschain\_contract`\*\*: Enables interaction with smart contracts on other chains, facilitating cross-chain functionality.

- \*\*Use Case\*\*: Useful for multi-chain DeFi applications or interoperable blockchain solutions.

### 8. \*\*Dynamic Input Validation with `validate\_input\_format`\*\*

A function that checks the format and correctness of user inputs before executing the contract logic.

#### Code Example:

```aiken

fn validate\_input\_format(input: String) -> Bool {

let valid\_length = length(input) >= 5;

let valid\_chars = all(is\_alphanumeric, input);

valid\_length && valid\_chars

}

```

#### Explanation:

- \*\*`validate\_input\_format`\*\*: Ensures that user inputs conform to expected formats, increasing security and preventing malformed data from entering the contract.

- \*\*Use Case\*\*: Helps in validating user-submitted data in DApps or user-driven contracts.

### 9. \*\*Composable Smart Contracts with `compose\_contracts`\*\*

A function that allows combining the logic of multiple contracts into a single execution flow.

#### Code Example:

```aiken

fn compose\_contracts(contract1: fn() -> Bool, contract2: fn() -> Bool) -> Bool {

contract1() && contract2()

}

fn contract\_a() -> Bool {

-- logic for contract A --

True

}

fn contract\_b() -> Bool {

-- logic for contract B --

True

}

fn execute\_composed\_contracts() -> Bool {

compose\_contracts(contract\_a, contract\_b)

}

```

#### Explanation:

- \*\*`compose\_contracts`\*\*: Chains two or more contracts into a unified execution flow, where all must pass for the overall transaction to succeed.

- \*\*Use Case\*\*: Useful in scenarios where multiple contracts interact, such as DeFi protocols combining lending and staking.

### 10. \*\*`trace\_execution` for Contract Execution Tracing\*\*

A function for tracing execution paths during debugging.

#### Code Example:

```aiken

fn trace\_execution<T>(message: String, f: fn() -> T) -> T {

trace("Execution start: ", message);

let result = f();

trace("Execution end: ", message);

result

}

fn complex\_calculation() -> Int {

-- complex logic here --

42

}

fn debug\_contract() -> Int {

trace\_execution("Complex Calculation", complex\_calculation)

}

```

#### Explanation:

- \*\*`trace\_execution`\*\*: Logs key points during contract execution to help developers debug and monitor contract behavior.

- \*\*Use Case\*\*: Useful for debugging complex smart contracts during development.

---

These new functions and code examples provide enhanced functionality, including improved error handling, performance profiling, state management, and cross-chain interactions, making Aiken smarter and more versatile for Cardano's smart contract ecosystem.

### Aiken lang new development multi-signature

Here's a smart contract example written in Aiken, incorporating some of the new functions and features we've discussed. This contract is designed to manage a multi-signature wallet, where funds can only be spent if a threshold of signatures from the wallet owners is met. It includes features like enhanced error handling, multi-signature verification, and execution profiling.

### Multi-Signature Wallet Contract Example in Aiken

```aiken

-- Define a multi-signature wallet smart contract

type Datum = {

owners: List<PubKeyHash>,

threshold: Int,

balance: Int

}

type Redeemer = {

signatures: List<Signature>

}

-- Validate multi-signature wallet transaction

fn validate(ctx: ScriptContext, datum: Datum, redeemer: Redeemer) -> Bool {

trace\_execution("Multisig Validation", || {

let tx\_info = ctx.tx\_info;

-- Ensure transaction has valid signatures

let valid\_signatures = verify\_multisig(datum.owners, redeemer.signatures, datum.threshold);

if !valid\_signatures {

trace("Validation failed: Not enough valid signatures");

return False;

}

-- Check that transaction withdraws the correct balance

let tx\_outputs = tx\_info.outputs;

let expected\_balance = datum.balance;

if check\_tx\_output(tx\_outputs, expected\_balance) {

trace("Validation passed: Transaction is valid");

True

} else {

trace("Validation failed: Balance mismatch");

False

}

})

}

-- Verify multi-signature validity

fn verify\_multisig(pub\_keys: List<PubKeyHash>, signatures: List<Signature>, required\_sigs: Int) -> Bool {

let valid\_sigs = filter(is\_valid\_signature, pub\_keys, signatures);

length(valid\_sigs) >= required\_sigs

}

-- Verify if a given signature matches a public key

fn is\_valid\_signature(pub\_key: PubKeyHash, signature: Signature) -> Bool {

verify\_signature(pub\_key, signature)

}

-- Check if the transaction outputs match the expected balance

fn check\_tx\_output(outputs: List<TxOut>, expected\_balance: Int) -> Bool {

let output\_balance = foldl(sum\_tx\_output, 0, outputs);

output\_balance == expected\_balance

}

fn sum\_tx\_output(acc: Int, tx\_out: TxOut) -> Int {

acc + tx\_out.value

}

-- Trace the execution of the smart contract for debugging purposes

fn trace\_execution<T>(message: String, f: fn() -> T) -> T {

trace("Execution start: ", message);

let result = f();

trace("Execution end: ", message);

result

}

-- Initialize the multi-signature wallet with owners and threshold

fn initialize\_wallet(owners: List<PubKeyHash>, threshold: Int) -> Datum {

Datum {

owners: owners,

threshold: threshold,

balance: 0

}

}

-- Example function to initialize the wallet with dummy data

fn create\_multisig\_wallet() -> Datum {

initialize\_wallet(

[

PubKeyHash("0x12345"),

PubKeyHash("0x67890"),

PubKeyHash("0xabcdef")

],

2 -- Two signatures required

)

}

-- Performance profiling for the contract's execution

fn profile\_contract\_execution(ctx: ScriptContext, datum: Datum, redeemer: Redeemer) -> (Bool, Time, Gas) {

profile\_execution(|| validate(ctx, datum, redeemer))

}

-- Profile the execution time and gas usage of the function

fn profile\_execution<T>(f: fn() -> T) -> (T, Time, Gas) {

let start\_time = current\_time();

let start\_gas = gas\_used();

let result = f();

let end\_time = current\_time();

let end\_gas = gas\_used();

(result, end\_time - start\_time, end\_gas - start\_gas)

}

```

### Contract Breakdown:

1. \*\*Multi-Signature Validation (`validate`)\*\*:

- This is the core validation function of the contract.

- It checks if the number of valid signatures is greater than or equal to the threshold required for the multi-signature wallet.

2. \*\*Error Handling and Debugging with `trace\_execution`\*\*:

- The `trace\_execution` function helps developers trace the flow of the contract’s execution by logging key points during contract operations, allowing better debugging.

3. \*\*Multi-Signature Verification (`verify\_multisig`)\*\*:

- This function verifies that enough valid signatures are provided to authorize a transaction.

- It filters out invalid signatures and compares the count of valid signatures to the required threshold.

4. \*\*Transaction Output Check (`check\_tx\_output`)\*\*:

- Ensures that the transaction is spending the correct amount of funds from the wallet by checking if the output matches the wallet balance.

5. \*\*Contract Initialization (`initialize\_wallet`)\*\*:

- This function sets up the wallet with a list of owners and a signature threshold.

- It initializes the contract’s state with a balance of 0 and stores the public keys of the owners.

6. \*\*Performance Profiling (`profile\_execution`)\*\*:

- The `profile\_execution` function measures the execution time and gas usage of the `validate` function.

- This can be useful for optimizing contract performance, especially in resource-sensitive blockchain environments.

### Usage Scenario:

1. \*\*Contract Creation\*\*:

- The contract is initialized with a set of owners and a signature threshold using `initialize\_wallet`. For example, a wallet can be created where three owners are registered, and two signatures are required for any transaction.

2. \*\*Contract Execution\*\*:

- When a transaction is made from the wallet, the `validate` function is called to check if the transaction has enough valid signatures. If the number of valid signatures is greater than or equal to the required threshold, the transaction proceeds.

3. \*\*Debugging and Profiling\*\*:

- The `trace\_execution` and `profile\_execution` functions provide insights into how the contract is executed, helping developers optimize their contracts and debug issues.

### Improvements:

This contract example showcases how we can integrate new Aiken features like error handling, multi-signature verification, and performance profiling, enhancing contract usability, security, and performance tracking.