Автономная некоммерческая организация высшего образования
«Университет Иннополис»

АННОТАЦИЯ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
(БАКАЛАВРСКУЮ РАБОТУ)
ПО НАПРАВЛЕНИЮ ПОДГОТОВКИ
09.03.01 – «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

НАПРАВЛЕННОСТЬ (ПРОФИЛЬ) ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ
«ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

Тема	**Проектирование и реализация языка программирования Elixireum: Elixir-подобный язык для виртуальной машины Ethereum**

Выполнил	**Поздняков Никита Михайлович** **Филонов Максим Андреевич**	подпись

Иннополис, Innopolis, 2024

Оглавление

Аннотация

В данной работе рассматривается необходимость создания нового языка программирования смарт-контрактов в экосистеме Ethereum, в которой в настоящее время доминируют контракты, написанные на Solidity. В нашей работе представлен Elixireum - функциональный язык с синтаксисом Elixir и динамической типизацией, предназначенный для повышения гибкости и креативности разработчиков децентрализованных приложений (dApp). Компилятор Elixireum был разработан и протестирован на совместимость с виртуальной машиной Ethereum (EVM) и способность поддерживать стандарты ERC-20 и ERC-721. Для сравнения потребления газа Elixireum и Solidity был использован комплексный набор тестов. Результаты показали, что, несмотря на функциональность и совместимость Elixireum с Ethereum, он потребляет больше газа, чем Solidity. Ключевые функции, такие как события и взаимодействие с хранилищем, были реализованы. Однако среди ограничений - высокий расход газа на развертывание, отсутствие внешних библиотек и взаимодействия с внешними смарт-контрактами. Будущая работа должна быть направлена на снижение затрат газа при развертывании и реализацию словарей, которые бы работали во время выполнения. Несмотря на высокое потребление газа, простота разработки и потенциал для более раннего развертывания Elixireum могут перевесить эти затраты, позиционируя его как жизнеспособную альтернативу в пространстве Web3.

Глава 1

Введение

1.1 Спрос на новый язык

Разработчики dApp в основном используют язык программирования Solidity. Моноязычность и монопарадигматизм ограничивают гибкость и креативность разработчиков. Кроме того, высокий порог вхождения, связанный с этим языком, может отпугнуть новичков. Для овладения им часто требуются значительные затраты времени и сил, что замедляет процесс разработки.

Учитывая эти ограничения, существует возможность для создания нового языка. Функциональная альтернатива, особенно та, которая поддерживает динамическую типизацию и опирается на основы общепризнанного языка программирования, могла бы значительно повысить доступность и эффективность разработки dApp. Такой язык не только облегчит освоение новым разработчикам, но и откроет новые возможности для инноваций в пространстве Web3. Такой потенциал делает разработку нового языка перспективным направлением для расширения возможностей и охвата среды программирования Ethereum.

1.2 Цель работы

1.2.1 Цели и гипотезы исследования

Нашей целью является создание языка смарт-контрактов с синтаксисом Elixir и динамическими типами - Elixireum. В результате язык должен быть выразительным, по крайней мере, чтобы иметь возможность реализовать токены ERC-20[1] и ERC-721[2]. Мы планировали добавить такие возможности, как неизменяемость, поддержка макросов и сопостовление с образцом, чтобы сделать этот язык функциональным. Также мы ожидали, что Elixireum будет потреблять столько же или меньше газа по сравнению с Solidity.

1.2.2 Ожидаемые результаты

- **Рабочий компилятор для Elixireum:** Разработка компилятора, способного транслировать язык Elixireum в байткод EVM.
- **Развернутый токен ERC-20:** Развернутый токен ERC-20, написанный на Elixireum, в качестве демонстрации того, что Elixireum обладает фундаментальной функциональностью, необходимой для разработки смарт-контрактов.
- **Сравнительное потребление газа:** Анализ потребления газа для контрактов Elixireum и сравнительная оценка с Solidity.

[1] https://eips.ethereum.org/EIPS/eip-20
[2] https://eips.ethereum.org/EIPS/eip-721

Глава 2

Методология

2.1 Тестирование и бенчмаркинг

Чтобы оптимизировать разработку Elixireum, мы решили создать набор тестов, охватывающий всю реализованную функциональность. После внедрения каждой функции или исправления ошибок мы запускали набор тестов, чтобы убедиться, что предыдущая функциональность не была нарушена. Тестовый набор включает в себя набор смарт-контрактов, которые в совокупности используют почти все реализованные функции, а также набор соответствующих тестовых Python-файлов.

Для сравнения мы также реализовали аналогичный скрипт бенчмарка на Python, который развертывает контракт Elixireum и его аналог в Solidity. Мы выполнили несколько запусков каждой функции, измеряя расход газа каждой из них и получая эту информацию из квитанций о транзакциях, предоставляемых узлом. Мы создали два модуля для сравнения пар контрактов ERC-20 и ERC-721.

Глава 3

Реализация

3.1 Обзор стека

В этом разделе описывается стек технологий, используемый в проекте: Elixir, Yul, компилятор Solidity.

3.1.1 Выбор Elixir

Здесь приводится обоснование выбора Elixir:

- Один из самых популярных обозревателей блокчейна с открытым исходным кодом, Blockscout[1], написан на языке Elixir. Исходя из этого, можно сказать, что Elixir хорошо известен в Web3-сообществе.
- Наша команда обладает значительным опытом в этом языке программирования.
- Elixir имеет открытый исходный код, поэтому мы можем легко использовать его механизмы для токенизации и парсинга Elixireum.

[1] https://www.blockscout.com – Официальный сайт Blockscout

3.1.2 Выбор Yul

- Изначально наша стратегия заключалась в компиляции Elixireum непосредственно в байткод EVM. Впоследствии мы переключились на промежуточное представление (ПП), выбрав Yul как ПП, соответствующее нашим требованиям. Преимущества этого решения заключаются в следующем:
  - Совместимость со всеми версиями EVM, что устраняет необходимость в разработке версий.
  - Избежание реализации таких примитивов, как вызовы функций, управление переменными и стеком, что позволяет высвободить ресурсы для расширения функциональных возможностей языка.
  - Снижение потребления газа благодаря оптимизатору компилятора Solidity.

  Несмотря на эти соображения, данный выбор имеет некоторые ограничения, а именно:
  - Зависимость от компилятора Solidity для Elixireum.
  - Нерешенная ошибка[2] в компиляторе Solidity, в нашем случае она проявляется для больших смарт-контрактов Elixireum при компиляции с отключенным оптимизатором. Ошибка приводит к невозможности работать с большим количеством переменных и большой глубиной стека.

[2]https://github.com/ethereum/solidity/issues?q=is:issue+is:open+StackTooDeepError

Никаких существенных недостатков по сравнению с прямой компиляцией в байткод EVM.

3.2 Технические детали

3.2.1 Организация памяти

В нашей языковой реализации мы решили хранить все переменные времени выполнения в энергозависимой памяти EVM. Это решение оправдано низкими газовыми затратами, связанными с операциями с памятью, и простотой реализации. Мы рассматривали две альтернативы: временное хранилище, которое дороже памяти, и стек, который предполагает более сложное управление.

В сгенерированном коде Yul переменная определяется как указатель на область памяти, где хранится ее значение. В этом указателе первый байт зарезервирован для номера типа, а последующие байты содержат само значение для простых случаев. Для более сложных случаев, таких как списки и кортежи, первый байт по-прежнему указывает на тип, затем следует слово[3], определяющее количество элементов, а затем сами элементы, каждый из которых хранится как встроенная переменная (тип + значение). Для строк или байтовых массивов первый байт указывает тип (1 для строк, 102 для байтовых массивов), следующее слово определяет количество байт, а затем собственно байты.

Поскольку мы разработали функциональный язык, мы должны следовать одному из его фундаментальных принципов - неизменяемости. Это

[3] в данном контексте слово - это 32 байта

означает, что каждая операция присваивания создает новую переменную. На практике мы «забываем» предыдущий указатель, присваиваем новый указатель и помещаем переменную в новое место. Каждая новая переменная должна храниться последовательно после предыдущей. Для этого в сгенерированном коде Yul мы определяем переменную offset$. Эта переменная содержит адрес памяти, где будет храниться следующая переменная. В качестве решения для пограничных случаев мы обновляем эту переменную с помощью Yul функции msize, присваивая переменной offset$ текущий размер памяти, округленный до одного слова.

В синтаксисе Elixireum нет спецификаторов типов для переменных. Для работы с динамическими типами мы установили несколько принципов, на которых построена наша система типов:

- Возвращаемые значения публичных функций должны строго соответствовать указанным типам; в противном случае, будет вызвана операция revert[4].
- Для литералов тип определяется во время компиляции.
- Тип хранится в первом байте переменной.

Пример работы со строками в Elixireum выглядит следующим образом:

```
1  a = "abc" # 0x616263
```

Листинг 3.1: Код Elixireum для регистра строк

Соответствующий Yul код, сгенерированный для этого случая строки, таков:

[4]opcode, который останавливает текущее выполнение и выбрасывает ошибку. Все изменения состояния будут отменены

```
let offset$ := msize()
let a$ := offset$
mstore8(offset$, 1)
offset$ := add(offset$, 1)
mstore(offset$, 3)
offset$ := add(offset$, 32)
mstore(offset$, 0x61626300...00)
offset$ := add(offset$, 3)
```

Листинг 3.2: Сгенерированный Yul-код для случая строки

В этом коде Yul переменная a$ определена как указатель памяти. Тип переменной сохраняется с помощью mstore8, затем с помощью mstore сохраняется размер байта строки. После хранения самой строки offset$ увеличивается на размер строки (3 байта).

Пример работы со списками в Elixireum выглядит следующим образом:

```
a = [1, true, "abc"]
```

Листинг 3.3: Код Elixireum для случая списка

```
let offset$ := msize()

let var0$ := offset$
...
let bool_var1$ := offset$
...
let str2$ := offset$
...

let a$ := offset$
mstore8(offset$, 103)
offset$ := add(offset$, 1)
mstore(offset$, 3)
offset$ := add(offset$, 32)
```

```
15    let ignored_
16    ignored_, offset$ := copy_from_pointer_to$(var0$, offset$)
17    ignored_, offset$ := copy_from_pointer_to$(bool_var1$, offset$)
18    ignored_, offset$ := copy_from_pointer_to$(str2$, offset$)
```

Листинг 3.4: Сгенерированный yul-код для случая строки

В этом коде Yul переменные var0$, bool_var1$ и str2$ определены для целочисленных, булевых и строковых литералов, соответственно. Затем определяется список a$. Длина списка хранится с помощью mstore, а фактические значения элементов списка копируются в память с помощью функции copy_from_pointer_to$ из внутренней стандартной библиотеки Elixireum, которая соответствующим образом обновляет offset$.

Глава 4

Выводы

4.1 Результаты и сравнение

Мы провели серию тестов, применив тестовый пакет к набору подготовленных смарт-контрактов. Все тесты завершились успешно, что свидетельствует о достижении главной цели проекта: разработан компилятор, способный транслировать язык Elixireum в байткод EVM. Кроме того, мы развернули токен ERC-20, реализованный на языке Elixireum, в тестовой сети Sepolia, следующие шестнадцатеричные байты - адрес развернутого токена 0x9DC699F6F8F3E42D4C6ae368C751325dC4106279[1]. Затем мы приступили к сравнению Elixireum с Solidity. Мы сравнили публичные функции реализации ERC-20 в Elixireum и ее полного аналога в Solidity, а также провели аналогичный бенчмарк для реализации ERC-721. В таблицах 4.1 и 4.2 представлены подробные метрики, которые мы получили.

Сравнение показывает, что хотя Elixireum успешно реализует функциональность контрактов ERC-20 и ERC-721, он в целом потребляет больше

[1] https://eth-sepolia.blockscout.com/token/0x9dc699f6f8f3e42d4c6ae368c751325dc4106279

Method	Elixireum Gas Used	Solidity Gas Used
mint	58197.4	54799.6
approve	47619.8	46570.8
transferFrom	54812.0	48492.2
transfer	55659.8	51813.4
burn	33061.6	29433.25
deploy of contract	1939441	732543

Таблица 4.1: Сравнение расхода газа по контракту ERC-20 (20 запусков)

Method	Elixireum Gas Used	Solidity Gas Used
setApprovalForAll	46596.0	46304.0
mint	166817.4	162675.4
transferFrom	87123.0	55652.6
approve	54798.0	49007.8
burn	58539.6	29944.0
deploy of contract	3067454	1113786

Таблица 4.2: Сравнение расхода газа по контракту ERC-721 (20 запусков)

газа по сравнению с Solidity для выполнения тех же операций. Таблица 4.1 показывает, что в среднем Elixireum на 8% менее эффективен, чем Solidity, для вызовов функций методов ERC-20 и примерно на 164% менее эффективен с точки зрения потребления газа в процессе развертывания. Таблица 4.2 показывает еще менее благоприятные показатели: Elixireum примерно на 20,5% менее эффективен для методов ERC-721 и потребляет на 175% больше газа во время развертывания. Эти данные свидетельствуют о том, что, несмотря на функциональность и совместимость Elixireum с блокчейном Ethereum, существует необходимость в оптимизации для снижения потребления газа и повышения эффективности.

4.2 Дискуссия

Результаты, полученные в этой главе, вносят значительный вклад в достижение более широкой цели работы - оценки осуществимости и потенциала Elixireum в контексте разработки смарт-контрактов. Мы подтвердили нашу гипотезу о том, что Elixir-подобный язык может быть скомпилирован в EVM.

Кроме того, мы провели комплексное сравнение потребления газа между Elixireum и Solidity. Результаты показали, что Elixireum потребляет в 2,7 раза больше газа, чем Solidity при развертывании, однако расход газа для методов ERC-20 и ERC-721 сопоставим. Это показывает, что можно иметь функциональный язык с динамической типизацией и типами, хранящимися в памяти, который потребляет примерно на 14% больше газа, чем Solidity.

С ростом цепочек свертывания, следует рассмотреть компромисс между потреблением газа и простотой разработки. Хотя вызов контракта, написанный в Elixireum, может стоить $1,15 вместо $1 в Solidity из-за большего потребления газа, простота и скорость разработки в Elixireum может привести к более раннему развертыванию. Такое более раннее внедрение может привлечь больше пользователей, что потенциально перевесит небольшое увеличение расходов на газ.