Факультет математики и компьютерных наук СПбГУ
Программа «Современное программирование»

Поиск клонов и автоматический рефакторинг Lua-программ в контексте задачи уменьшения размера кода для сетевых устройств

Дружков Сергей Александрович

Выпускная квалификационная работа

Научный руководитель: профессор кафедры системного программирования СПбГУ, д.т.н. Д. В. Кознов
Рецензент: директор лаборатории ООО «Техкомпания Хуавэй», Д. А. Кудрявцев

Санкт-Петербург
2025

Программное обеспечение сетевых устройств

- Отвечает за маршрутизацию сетевых пакетов, обеспечение сетевого управления устройства и т.д.
- Существует на устройстве в виде прошивки, а также работает под управлением Linux
- Содержит базу данных устройства (до нескольких тыс. таблиц)
- Использует скриптовый язык Lua для доступа к данным (Cisco, Xiaomi и Huawei)
- Стоит задача уменьшения размеров Lua-кода, который загружается на маршрутизатор

Постановка задачи

Цель: Разработка *решения* для поиска клонов и автоматического рефакторинга Lua-программ, предназначенных для сетевых устройств крупной телекоммуникационной компании с последующей интеграцией решения в конвейер сборки прошивки.

Задачи:

1. Провести обзор предметной области
2. Сформулировать требования к решению
3. Спроектировать архитектуру решения, включая выбор методов поиска клонов и рефакторинга
4. Реализовать поиск клонов и рефакторинг Lua-программ
5. Провести экспериментальное исследование различных аспектов разработанного решения

Обзор предметной области

- Не существует специализированных детекторов клонов для Lua
- Большинство детекторов клонов нацелены на поиск наибольшего числа клонов, тогда как далеко не все из них возможно и выгодно рефакторить
- Средства рефакторинга клонов часто имеют различные ограничения: необходимость контроля со стороны человека, работа только с парами клонов или только с клонами на уровне функций

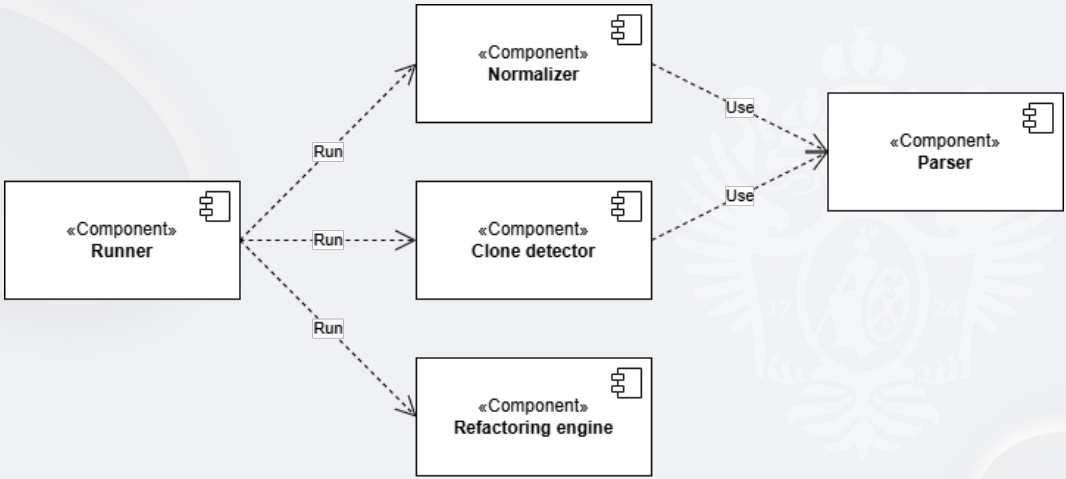
Требования к решению

- **Функциональные**
  1. Возможность поиска клонов на языке программирования Lua на уровне функций и блоков кода
  2. Возможность автоматического рефакторинга групп клонов
  3. Возможность поиска и рефакторинга вложенных клонов
  4. Конфигурируемость поиска и рефакторинга клонов
- **Нефункциональные**
  1. Поддержка работы с большими проектами ($\approx$400Мб, $\geq$10MLOC)
  2. Минимальное возможное число внешних зависимостей
  3. Детерминированность результатов работы решения
  4. Максимальное сохранение структуры кода при рефакторинге
  5. Безопасность всех преобразований кода

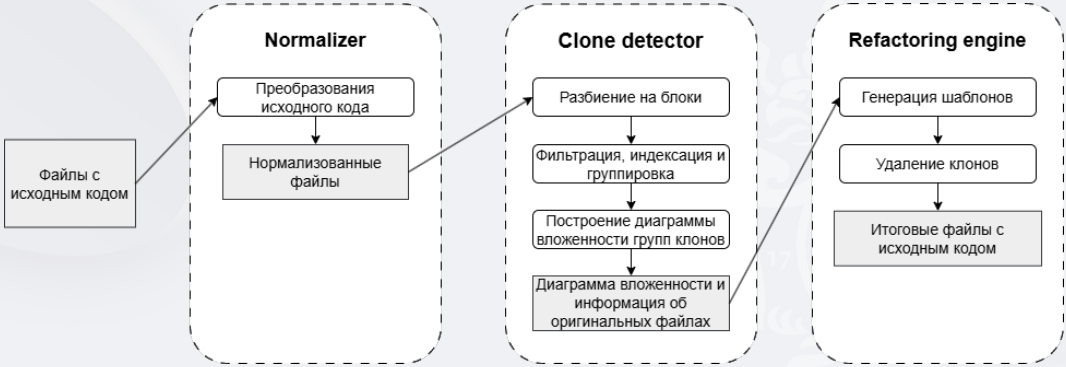
Требования к решению

- **Функциональные**
  1. Возможность поиска клонов на языке программирования Lua на уровне функций и **блоков кода**
  2. Возможность автоматического рефакторинга **групп клонов**
  3. Возможность поиска и рефакторинга **вложенных клонов**
  4. Конфигурируемость поиска и рефакторинга клонов
- **Нефункциональные**
  1. Поддержка работы с **большими проектами** ($\approx$400Мб, $\geq$10MLOC)
  2. Минимальное возможное число внешних зависимостей
  3. Детерминированность результатов работы решения
  4. Максимальное **сохранение структуры** кода при рефакторинге
  5. Безопасность всех преобразований кода

Архитектура решения



Сценарий работы решения



Нормализация[1]

1. Удаление лишних `return`
2. Удаление необязательной инициализации
3. Удаление лишних точек с запятой
4. Переименование меток
5. Удаление неиспользуемого кода (переменные, функции, метки)
6. Объединение объявлений локальных переменных
7. Преобразование обращения по полю в обращение по ключу

[1] Pizzolotto D., Inoue K. Blanker: a refactor-oriented cloned source code normalizer // 2020 IEEE 14th International Workshop on Software Clones (IWSC). – IEEE, 2020. – С. 22-25.

Поиск клонов

Алгоритм поиска клонов был вдохновлен инструментом Vuddy[2].
Предложены следующие улучшения алгоритма.

1. Уменьшение гранулярности с уровня функций до уровня блоков
2. Выделение дополнительных блоков кода
3. Консистентное переименование переменных при сопоставлении
4. Дополнительная фильтрация найденных клонов (возможность и выгодность рефакторинга, размер клона в токенах, использование «опасной» функциональности)

[2]Kim S. et al. Vuddy: A scalable approach for vulnerable code clone discovery // 2017 IEEE symposium on security and privacy (SP). – IEEE, 2017. – С. 595-614.

Поиск клонов

1. Строим АСД для каждого файла
2. Разбиваем исходный код на вложенные блоки
3. Фильтруем блоки на основании правил (размер, безопасность, возможность рефакторинга, …)
4. Производим переименование переменных и констант внутри блока
5. Индексируем блоки для каждого файла по отдельности
6. Объединяем индексы для всех файлов
7. Строим диаграмму вложенности групп клонов
8. Фильтруем группы блоков на основе выгодности рефакторинга

Рефакторинг

Выбранный метод рефакторинга — извлечение функции с последующей параметризацией разницы между клонами в группе. Реализованный однопроходный алгоритм рефакторинга имеет следующие основные преимущества.

1. Возможность рекурсивного рефакторинга вложенных клонов
2. Аккуратная работа с локальными и глобальными переменными
3. Учет внутренних особенностей интерпретатора Lua
4. Сохранение структуры исходного кода

Конфигурирование

- Область анализа решения
  - Каждый файл по отдельности
  - Весь проект
- Гранулярность клонов
  - Функция
  - Блок кода
- Минимальный размер клона в токенах
- Пороговое значение эвристики рефакторинга
- Опции нормализации

Пример работы

before.lua

```lua
-- run command: os.vrunv
function _runcmd_vrunv(cmd, opt)
    if cmd.program then
        if opt.dryrun then
            vprint(os.args(table.join(cmd.program, cmd.argv)))
        else
            os.vrunv(cmd.program, cmd.argv, cmd.opt)
        end
    end
end

-- run command: os.execv
function _runcmd_execv(cmd, opt)
    if cmd.program then
        if opt.dryrun then
            print(os.args(table.join(cmd.program, cmd.argv)))
        else
            os.execv(cmd.program, cmd.argv, cmd.opt)
        end
    end
end

-- run command: os.vexecv
function _runcmd_vexecv(cmd, opt)
    if cmd.program then
        if opt.dryrun then
            print(os.args(table.join(cmd.program, cmd.argv)))
        else
            os.vexecv(cmd.program, cmd.argv, cmd.opt)
        end
    end
end
```

after.lua

```lua
function generated(_ENV, GLOBAL1, LITERAL7, cmd, opt)
    if cmd.program then
        if opt.dryrun then
            _ENV[GLOBAL1](os.args(table.join(cmd.program, cmd.argv)))
        else
            os[LITERAL7](cmd.program, cmd.argv, cmd.opt)
        end
    end
end

-- run command: os.vrunv
function _runcmd_vrunv(cmd, opt)
    generated(_ENV, "vprint", "vrunv", cmd, opt)
end

-- run command: os.execv
function _runcmd_execv(cmd, opt)
    generated(_ENV, "print", "execv", cmd, opt)
end

-- run command: os.vexecv
function _runcmd_vexecv(cmd, opt)
    generated(_ENV, "print", "vexecv", cmd, opt)
end
```

Экспериментальное исследование: проекты

Проект	Размер, Мб	Число файлов	Число строк	Версия Lua
CE6866	216.41	22736	5593690	Lua5.3, LuaJIT
VRP	220.23	23437	5385891	Lua5.3, LuaJIT
Xmake	5.7	1477	101920	Lua5.4, LuaJIT
Lua Language Server	1.56	242	47823	Lua5.3
Kong	3.03	549	74092	LuaJIT
APISIX	1.65	264	39976	LuaJIT

Экспериментальное исследование: результаты

1. Решение позволяет добиться уменьшения размера кода на 5-7%
2. Дополнительный этап нормализации кода увеличивает число найденных клонов на 1-2%
3. Уменьшение гранулярности и расширение области анализа приводят к значительному увеличению числа найденных и отрефакторенных клонов (в 2-3 раза)
4. Время работы решения составляет < 1.5 мин. при потреблении памяти < 4 Гб даже для самых больших проектов
5. Получена эпмирическая верхняя оценка замедления, вызванного рефакторингом клонов, в среднем равная 5%
6. Проведена валидация корректности работы решения на основе тестов проектов и ручной проверки отдельных групп клонов

Результаты работы

1. Выявлены функциональные и нефункциональные требования к решению на основе взаимодействия с заказчиком
2. Спроектирован конвейер работы решения, состоящий из трех этапов: нормализация, поиск клонов и рефакторинг
3. Выполнена реализация решения
4. Проведено экспериментальное исследование различных аспектов решения
5. Начато внедрение разработанного решения в конвейер сборки прошивки крупной телекоммуникационной компании
6. Результаты данной работы были представлены на конференции «Современные технологии в теории и практике программирования»
7. Планируется публикация статьи на основе полученных результатов

Проблемы переносимости

- Необходимость использовать новый парсер для каждого языка
- Учет особенностей другого языка при поиске и рефакторинге клонов
  - синтаксический сахар
  - декораторы или аннотации
  - ООП
  - работа с полями и глобальными переменными
- Большие отличия в системе модулей и пакетов

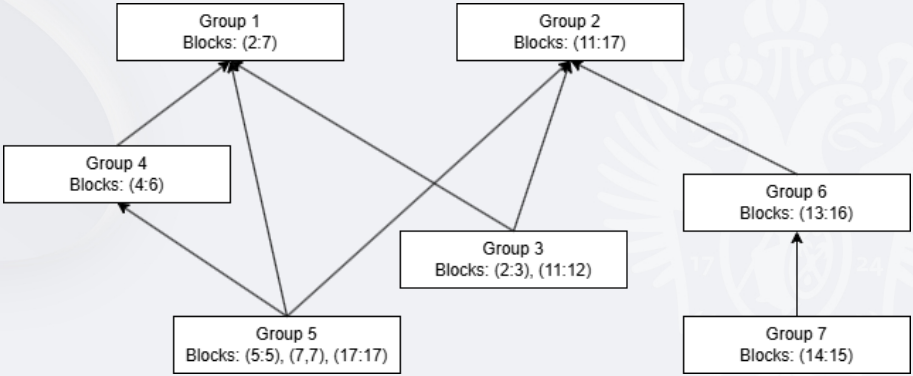
Главная идея (поиск групп клонов и их выделение в новые функции) подходит для всех языков, так же как и предложенный алгоритм работы с вложенными клонами. Однако проще реализовать отдельное решение для каждого языка программирования, чтобы учесть все вышеописанные нюансы.

Дальнейшее развитие

- Добавление новых трансформаций для нормализации
  - удаление `else` после `return`
  - унификация условных выражений
  - вычисление константных выржажений
- Более точный расчет выгодности рефакторинга для группы
- Оптимизация рефакторинга групп клонов
  - дополнительная фильтрация блоков в группе
  - генерация более чем одной новой функции на группу
  - новые методы рефакторинга
- Инкрементальные сценарии работы (IDE, …)

МКН

Диаграмма вложенности



Типы клонов

Оригинальный код

```
int f(int x, int y) {
  int z = x + y;
  return 2 * z;
}
```

Клон типа T1

```
int f(int x, int y) {
  int z =
    x + y; // sum
  return 2 * z;
}
```

Клон типа T2

```
char f(char a, char b) {
  char c = a + b;
  return 3 * c;
}
```

Клон типа T3

```
char f(char a, char b) {
  char p = b * a;
  return 4 * p;
}
```

МКН

Результаты нормализации

Режим	Изначальное число клонов	Опции нормализации	Новое число клонов	Δ, %
Cross + Block	225855	returns	225973	+0.05
		nils	225731	-0.05
		semicolons	227883	+0.90
		labels	225864	+0.003
		unused	225705	-0.06
		join	226487	+0.28
		all	228556	+1.20

МКН

Результаты поиска клонов

Проект	Режим	Число групп клонов	Общее число клонов
CE6866	Single + Func	5432	14488
	Cross + Func	12505	38881
	Single + Block	37407	104812
	Cross + Block	53910	225855
VRP	Single + Func	5329	14050
	Cross + Func	12855	40626
	Single + Block	41140	118933
	Cross + Block	54896	247409

МКН

Результаты рефакторинга

Проект	Базовый размер, Кб	Режим	Новый размер, Кб	Δ, %	Время работы, с	Потребление памяти, Мб
CE6866	129005	Single + Func	127820	-0.919	35	735
		Cross + Func	125102	-3.025	40	2917
		Single + Block	125940	-2.375	39	743
		Cross + Block	122019	-5.415	62	3704
VRP	139473	Single + Func	138008	-1.050	40	665
		Cross + Func	134339	-3.681	45	3101
		Single + Block	134642	-3.463	43	670
		Cross + Block	129357	-7.253	70	3935

Результаты бенчмарка

Режим	Lua 5.1	Lua 5.2	Lua 5.3	Lua 5.4	LuaJIT 2.1 interpreter	LuaJIT 2.1
Vanilla	43.22 с.	41.99 с.	35.71 с.	24.86 с.	15.12 с.	5.39 с.
Single + Func	45.17 с. (+4.51%)	43.11 с. (+2.67%)	36.59 с. (+2.46%)	25.88 с. (+4.10%)	15.78 с. (+4.36%)	5.39 с. (+0.00%)
Cross + Func	45.36 с. (+4.95%)	43.54 с. (+3.69%)	36.74 с. (+2.88%)	25.84 с. (+3.94%)	15.76 с. (+4.23%)	5.41 с. (+0.37%)
Single + Block	57.05 с. (+31.99%)	55.14 с. (+31.32%)	43.79 с. (+22.63%)	32.89 с. (+32.30%)	18.38 с. (+21.56%)	5.61 с. (+4.08%)
Cross + Block	45.17 с. (+36.26%)	43.11 с. (+34.69%)	44.96 с. (+25.90%)	34.03 с. (+34.88%)	19.04 с. (+25.92%)	5.62 с. (+4.27%)

МКН

Компромиссы

- **Single + Func mode.** Новые функции оказываются локальными, и при их вызове на месте тела исходной функции по стандарту происходит оптимизация хвостового вызова, что оказывает минимальное влияние на производительность.
- **Cross + Func mode.** Все еще происходит оптимизация хвостовых вызовов, но шаблоны функций для межфайловых клонов сохраняются в общую глобальную таблицу, поэтому их вызов может быть замедлен.
- **Single + Block mode.** Новые функции оказываются локальными, но нет оптимизации хвостовых вызовов, поэтому может быть большой рост глубины стека и накладных расходов на вызов функции.
- **Cross + Block mode.** В данном режиме проявляются все виды замедления, но при этом удается добиться максимального сжатия.