Содержание

Введение

Существует множество примеров того, как ошибки в программном обеспечении приводят к финансовым потерям и даже людским жертвам [1]. Но даже небольшие баги в коде приводят к большим временным затратам программиста, который обречен искать их часами или днями. Поэтому анализ кода на предмет нахождения уязвимостей, участков неэффективного кода и другого рода ошибок — актуальная тема в наше время.

Существует два основных подхода к анализу кода: статический анализ, который проводится без фактического выполнения программы, и динамический анализ, при котором код анализируется непосредственно во время выполнения. К преимуществам статического анализа можно отнести обнаружение ошибок на ранних этапах разработки программного обеспечения. Это существенно снижает стоимость устранения дефектов в программе, так как чем раньше выявлена ошибка, тем легче и, как следствие, дешевле её исправить. Другой плюс в том, что статический анализ помогает обнаружить уязвимости безопасности, такие как SQL-инъекции и XSS-атаки, без риска при запуске.

Одним из популярных методов статического анализа является абстрактная интерпретация. В данном подходе во время анализа для каждой точки программы строится абстрактное состояние, которое является аппроксимацией возможных конкретных состояний в этой точке, а операции над абстрактными элементами являются оверапроксимацией (over-approximation) реального поведения программы. Оверапроксимация означает, что абстрактный анализ покрывает все возможные сценарии, а также возможные, но не достижимые. Это свойство важно, поскольку оно позволяет гарантировать отсутствие ошибок при успешном выполнении.

Однако, оверапроксимация обладает недостатком — вероятность ложных срабатываний (False Positives). Это означает, что в некоторых ситуациях анализатор указывает на ошибку, которой на самом деле нет. Происходит из-за недостаточной точности. Во время анализа каждой переменной сопоставляется значение из абстрактного домена, представляющего собой спе-

циализированную математическую структуру — решетку, то есть частично упорядоченное множество, в котором для любых двух элементов можно определить наименьшую общую верхнюю границу (join) и наибольшую общую нижнюю границу (meet). Эти операции позволяют объединять и пересекать абстрактные состояния, обобщать или уточнять информацию в процессе анализа.

Основной проблемой является недостаточная точность строковых абстрактных доменов. В некоторых случаях анализ не может корректно определить недостижимость определенных веток исполнения кода, что приводит к ложным срабатываниям. Это снижает полезность статического анализа, так как разработчики вынуждены разбираться с ложными предупреждениями.

Цели и задачи

Целью данной работы является повышение точности строковой решетки, используемой в статическом анализаторе кода на базе абстрактной интерпретации.

Задачи:

- Провести обзор существующих строковых абстрактных доменов
- Спроектировать структуру более точной строковой решетки на их основе
- Реализовать операции для работы с абстрактными значениями строк и интегрировать в статический анализатор
- Протестировать новую функциональность, провести замеры производительности статического анализатора при использовании новой решетки

Обзор предметной области

Статический анализ

Статический анализ программного кода представляет собой метод исследования программ без их выполнения. Основная цель статического анализа — выявление потенциальных ошибок, уязвимостей и других характеристик программного обеспечения на основе его исходного кода или промежуточного представления. Этот метод используется в компиляторах, инструментах проверки кода, системах анализа безопасности и других приложениях.

Существует несколько техник статического анализа, включая анализ потока данных, анализ потока управления, типизацию, проверку соответствия спецификациям и другие. В отличие от динамического анализа, который требует выполнения программы, статический анализ проводится на уровне исходного кода, байт-кода или абстрактного синтаксического дерева.

Основные применения

- Поиск уязвимостей
- Оптимизация кода
- Верификация свойств
- Генерация тестов

Два важных метода статического анализа — символьное исполнение и абстрактная интерпретация — позволяют анализировать поведение программ без их непосредственного запуска, но имеют разные подходы и цели.

Символьное исполнение

Символьное исполнение (Symbolic Execution) [3] представляет собой метод анализа, при котором программа исполняется не на конкретных входных данных, а на символьных переменных . В процессе исполнения строится множество путей выполнения программы, а также логические ограничения, наложенные на переменные. Этот метод используется, например, для генерации тестов с высоким покрытием и обнаружения уязвимостей.

Преимущества

- Позволяет находить ошибки, недоступные при обычном тестировании [4]
- Даёт точные результаты в пределах анализируемых путей

Ограничения

- Страдает от проблемы комбинаторного взрыва, так как количество путей выполнения растёт экспоненциально [5]
- Требует сложных SMT Solver-ов для обработки выражений (Z3 [6], CVC5 [7] и др.)
- Нет гарантии оверапроксимации, солвер не может доказать, что решения нет

Абстрактная интерпретация

Абстрактная интерпретация (Abstract Interpretation) [2] основана на создании приближённых представлений возможных состояний программы. Вместо работы с конкретными или символьными значениями, этот метод использует абстрактные значения, которые обобщают множества возможных состояний.

Преимущества

- Эффективный анализ больших программ [8]
- Гарантированная завершаемость

Ограничения

- Ложные срабатывания
- Сложность разработки точных абстракций

Сравнение методов

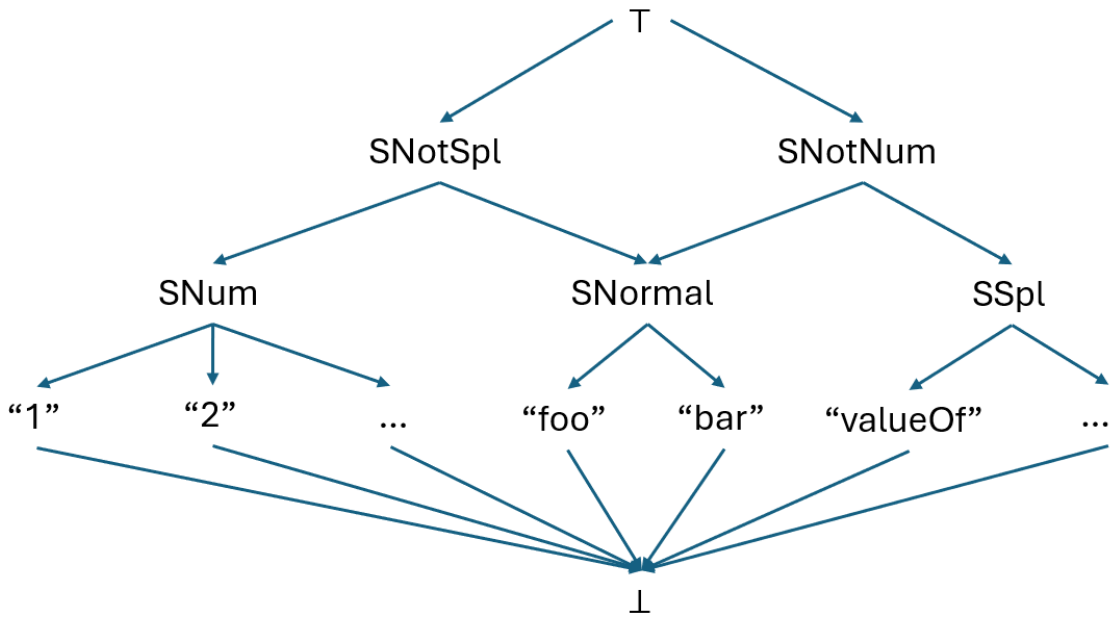
Характеристика	**Символьное исполнение**	**Абстрактная интерпретация**
Точность анализа	Высокая (по отдельным путям)	Приближённая (но глобальная)
Гарантия отсутствия уязвимостей	Нет	Есть
Масштабируемость	Ограниченная (комбинаторный взрыв)	Высокая (благодаря абстракции)
Применимость	Поиск ошибок генерация тестов	Обнаружение уязвимостей верификация
Требования к вычислениям	Высокие (SAT solver)	Более низкие

Таким образом, оба метода находят своё применение в статическом анализе, а их комбинирование может дать более точные и эффективные результаты.

Строковые абстрактные домены

Пример простого строкового абстрактного домена реализован в системе **JSAI** [9] — платформе статического анализа JavaScript. В ней строки представляются в виде:

- любая ($\top$) и неинициализированная ($\bot$) строки
- константные строки
- категории “строка-число”, “спец-символ” и “любая строка”



Хотя этот подход обладает высокой производительностью, он страдает от недостатка точности. Например, пусть переменная s может быть равна строке “foo” или “bark”. Тогда значение выражения `s.length` может быть либо 3, либо 4. Теперь посмотрим, какое абстрактное значение будет у s при использовании JSAI. Для этого необходимо найти в решетке наименьший элемент, содержащий в себе все возможные принимаемые значения s. Это класс SNormal. Тогда если хотим посчитать для абстрактной переменной `s.length`, нужно учесть минимальное и максимальное значение внутри класса. Но там

есть и пустая строка, и сколь угодно длинная. Это делает невозможным отбрасывание условий вроде `if (s.length == 0)` как недостижимых, что ведёт к ложным срабатываниям

Листинг 1: Пример недостаточной точности в строковом домене JSAI

```
let s;
if (input == 0) {
    s = "foo";
} else {
    s = "bark";
}
if (s.length == 0) {
    throw new Error("Divide by zero!");
}
let repeats = 100 / str.length;
```

The String Set

Домен множества строк (SS_k) сохраняет не более k константных строк. В случае переполнения сбрасывается в $\top$. Этот домен заметно более затратен по времени и памяти. Достаточно просто рассмотреть функцию конкатенации двух абстрактных значений, при которой количество элементов будет произведением. А конкатенация самая часто использующаяся функция на строках, потому от её оптимизации будет многое зависеть

Точность этой решетки очень высокая, так как каждую операцию можно отдельно проделать с каждой строкой в множестве. Но из-за этого и производительность сильно проседает. А также есть проблемы с точностью при работе $\top$, например при конкатенации. Или для циклов, где при большом количестве итераций существует большое количество возможных значений

The Abstract Length string domain

Домен длины абстрактной строки (LS) [10] запоминает минимальную и максимальную длину строк, которые может принимать абстрактное значение. Например $LS(\text{"}foo\text{"};\text{"}bark\text{"}) = [3, 4]$

Для примера выше она отлично подойдет, так как запомнит, что минимальная длина 3, а максимальная 4, и даже сможет верно вернуть значение для repeats, то есть интервал [25, 34]. Однако для других операций, таких как substr или charAt, она не годится

The Character Inclusion domain

Домен включения символов (CI) [11] отслеживает символы, встречающиеся в строке. Каждая абстрактная строка имеет вид [L, U]. Нижняя граница L содержит символы, которые должны встречаться в конкретной строке (строках), в то время как верхняя граница U представляет символы, которые могут появиться

Этот домен полностью игнорирует структуру конкретных строк, которые аппроксимирует. То есть например

$$CI(\text{"}foo\text{"}) = CI(\text{"}of\text{"}) = [\{'f', 'o'\}, \{'f', 'o'\}]$$

Но CI, как правило, дешев в вычислительном отношении и иногда предоставляет полезную информацию. Например функция contains, для которой можно вернуть true в случае проверки на включение одной буквы, которая содержится в нижней границе, или же false, если слово содержит букву, которой нет в верхней

The Prefix-Suffix domain

Элементами префикс-суффикс домена (PS) [12] являются пары $[p, s]$, содержащие в себе все строки, которые начинаются с p и заканчиваются на s

$$PS(\text{"}abacab\text{"}; \text{"}abab\text{"}) = [\text{"}aba\text{"}; \text{"}ab\text{"}]$$

Также как и CI, PS не может хранить конкретные строки. Но есть ряд функций, такие как startsWith, endsWith, а иногда и substr, indexOf, для которых будет возвращаться точный ответ в удачных случаях

Конечные автоматы

Все перечисленные сверху решетки хороши каждая для своего ограниченного набора функций. Но в общем случае будет много неточностей. Поэтому стоит рассмотреть другие подходы, обладающие большей точностью, хоть и более сложные

Для повышения точности анализа был предложен подход, основанный на **конечных автоматах с переходами по буквам** [13]. В этом подходе множество возможных значений строк моделируется как язык, распознаваемый конечным автоматом. Операции над строками соответствуют операциям над языками: объединение, пересечение, конкатенация и пр.

Преимущества

- высокая точность

Недостатки

- высокая вычислительная сложность
- большое потребление памяти
- низкая масштабируемость при анализе больших программ

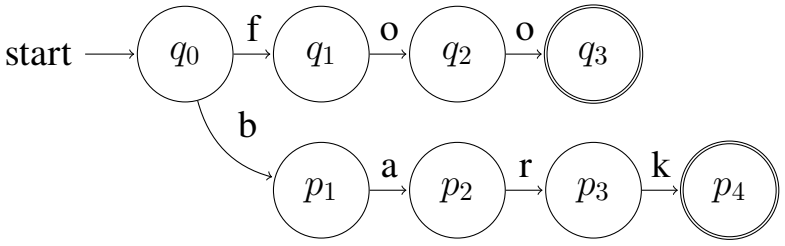


Рис. 1: Конечный автомат, распознающий строки “foo” и “bark”

Овераппроксимация строк в конечных автоматах повышает точность анализа строк во многих сценариях, но она не подходит для реальных программ, работающих со статически неизвестными входными данными и манипуляциями с длинным текстом

TARSIS

TARSIS (Template-based Abstract Representation of Strings Static analysis) — это современный строковый абстрактный домен, предложенный в статье [14]. Основное новшество Tarsis заключается в том, что он работает с алфавитом строк, а не с отдельными символами. С одной стороны, такой подход требует более сложного и уточненного определения расширяющего оператора и абстрактной семантики строковых операторов. С другой стороны, это позволяет получать более точные результаты

Основные функции

В статье описан алгоритм для аппроксимации с доказательством полноты наиболее часто использующихся функций на строках, а именно length, concat, substr, replace, contains и indexOf

$$[\text{length}(s)] = |\sigma|$$

$$[\text{concat}(s, s')] = \sigma \cdot \sigma'$$

$$[\text{substr}(s, a, a')] = \sigma_i \dots \sigma_j \quad \text{if} \quad i \leq j < |\sigma|$$

$$[\text{replace}(s, s', s'')] = \begin{cases} \sigma\,[s'/s''] & \text{if } \sigma' \curvearrowright_s \sigma \\ \sigma & \text{otherwise} \end{cases}$$

$$[\text{contains}(s, s')] = \begin{cases} \text{true} & \text{if } \exists i, j \in \mathbb{N}.\sigma_i \dots \sigma_j = s' \\ \text{false} & \text{otherwise} \end{cases}$$

$$[\text{indexOf}(s, s')] = \begin{cases} \min\{i \mid \sigma_i \dots \sigma_j = s'\} & \text{if } \exists i, j \in \mathbb{N}.\sigma_i \dots \sigma_j = s' \\ -1 & \text{otherwise} \end{cases}$$

Также важной функцией является widening [15], при помощи которой удается завершать анализ циклов за конечное время

Сравнение строковых решеток

Домен	Точность	Операции	Работа с $\top$	Скорость
String Set (SS_k)	Хранит до k строк	Все	Переполнение в $\top$	Низкая
Abstract Length	Длина	Только length	Минимальная длина	Высокая
Character Inclusion	Только символы	contains	Нет	Высокая
Prefix-Suffix	Префиксы суффиксы	starts/endsWith substring	Префиксы суффиксы	Высокая
Строковые автоматы	Конечный автомат	Все	Через $\top$-переходы	Очень низкая
TARSIS	Строковый автомат	Все (с ограничениями)	Через $\top$-переходы	Средняя

Можно видеть, что наибольшей точностью обладают строковые конечные автоматы. Они реализуют все операции и умеют работать с $\top$ значениями, не теряя точности. Однако вычислительная сложность настолько большая, что для реального анализа не подходит

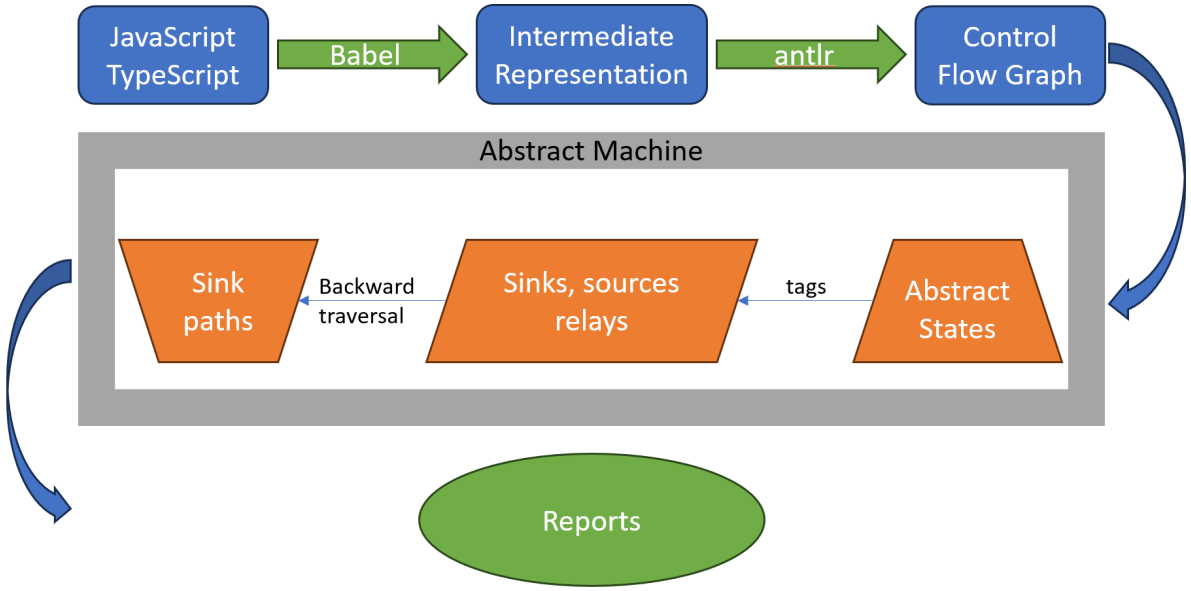
Далее идет String Set. Есть возможность точно просчитывать все операции, так как это просто вызов на каждой строке в множестве. Однако при переполнении, либо при работе с неизвестной строкой, вся точность теряется. То есть очень распространенный случай использования строк в виде `result = ‘‘foo returns: ’’ + foo()` никак не обрабатывается

Оставшиеся 3 решетки направлены каждая на свою часть функций, и в общем случае хоть и обладает высокой скоростью, но точность слишком мала. Поэтому решено взять за основу идею TARSIS, оптимизировав конечный автомат, разрешив переходы по словам

Реализация

Анализатор:

В компании есть работающий анализатор, основанный на абстрактной интерпретации. Поддерживаемые языки — JavaScript, TypeScript. Анализируемый код транспилируется при помощи Babel [16] в промежуточное представление (Intermidiate Representation) [18]. Затем IR с помощью ANTLR [17] переводится в граф потока управления (Control Flow Graph) [19] — множество всех возможных путей исполнения программы, представленное в виде графа



Этот граф передается в абстрактную машину, которая обычные переменные переводит в абстрактные, то есть принимающие множество возможных значений. Далее применяется Taint анализ [20], и в результате для каждой уязвимости выводится весь путь

Необходимо реализовать:

Реализован интерфейс для решёток, интервальная решетка для целых чисел и строковая решётка из JSAI, которую необходимо улучшить

В новой версии решетки должны присутствовать:

- Структура автомата, функции для работы с ним
- Общие операции для решеток (join, meet, strictEquals, ...)
- Операции на строках (length, concat, substring, ...)
- Оператор расширения (widening)
- Тесты на новую функциональность

На существующем бенчмарке из 23 реальных проектов:

- Суммарное время анализа не должно увеличиться более чем на 25%
- Потребление памяти не должно увеличиться более чем на 15%

Структура

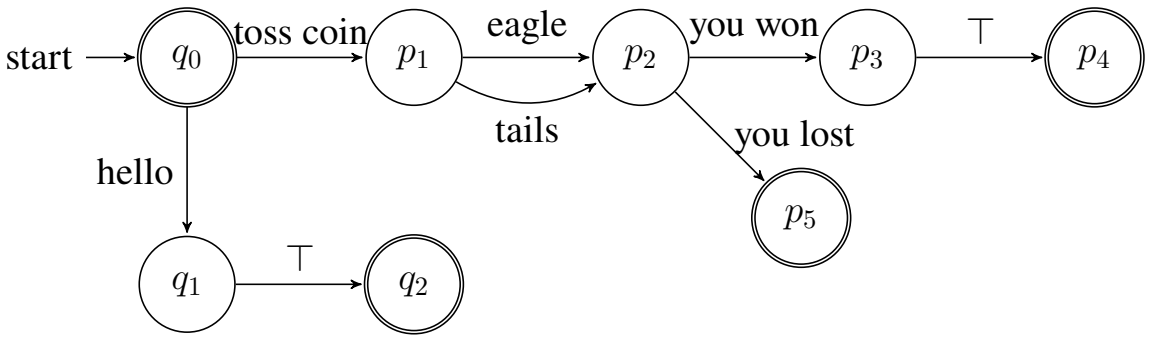
Структура автомата — есть вершины (states) и переходы между ними. Некоторые из вершин помечены как начальные или как конечные. На переходах стоят строки или символ $\top$, обозначающий любую строку. Для понимания рассмотрим такой пример:

```
str = ""
if (play) {
    coin = tossCoin()
    str = "toss coin" + coin + calcBet(coin)
} else if (greetings) str = "hello " + name()
```

Поймем, какое абстрактное значение будет у str после всех if-ов. Может быть 3 варианта — зайдем в ветку с подкидыванием монетки, зайдем в ветку с приветствием или вообще никуда не зайдем. Все эти варианты возможны, поэтому значение str — объединение всех трех. Также определим, что делают функции

- name() — Возвращает имя, которое пользователь ввел когда-то в процессе выполнения. Изначально оно неизвестно, поэтому абстрактное значение для него $\top$
- tossCoin() — С вероятностью 50% возвращает “eagle”, иначе “tails”
- calcBet() — Проверяет значение, которое выпало на монетке, на соответствие ожидаемому. И если оно совпадает, то возвращает “you won” вместе с призом, который зависит от ставки. Ставка также может быть неизвестна изначально, поэтому $\top$. В случае несоответствия просто возвращается “you lost”

  Итоговый автомат для str будет выглядеть следующим образом:



Объединение, пересечение и включение автоматов

Основной операцией является объединение автоматов — $A \cup B$. С её помощью находим наименьший язык, содержащий оба, порождаемых автоматами. Реализация: создаем общее начальное состояние, проводим из него переходы с пустой строкой в начальные состояния двух объединяемых автоматов. Затем минимизируем полученный автомат (чуть ниже описана функция minimize)

Другой важной операцией является пересечение автоматов — $A \cap B$. Она позволяет понять, есть ли в двух алфавитах общие слова. А также находить наибольший общий подъязык

Теорема 1 (Прямое произведение автоматов)**.** *Для всяких двух DFA* $\mathcal{A} = (\Sigma, P, p_0, \eta, E)$ *и* $\mathcal{B} = (\Sigma, Q, q_0, \delta, F)$*, пересечение* $L(\mathcal{A}) \cap L(\mathcal{B})$ *распознаётся DFA* $\mathcal{C} = (\Sigma, P \times Q, (p_0, q_0), \pi, E \times F)$*, где функция переходов действует покомпонентно* $\pi\big((p, q), a\big) = \big(\eta(p, a), \delta(q, a)\big)$ *[21]*

Проверка на то, что один язык полностью содержит другой реализуется с помощью дополнения и пересечения по формуле

$$A \subset B <=> B' \cap A = \emptyset$$

Теорема 2. *Для всякого DFA* $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$*, DFA* $\mathcal{A}' = (\Sigma, Q, q_0, \delta, Q \setminus F)$ *распознаёт дополнение* $L(\mathcal{A})$ *[21]*

minimize

Возвращает минимальный автомат, эквивалентный данному, с помощью алгоритма минимизации Бржозовского [22]

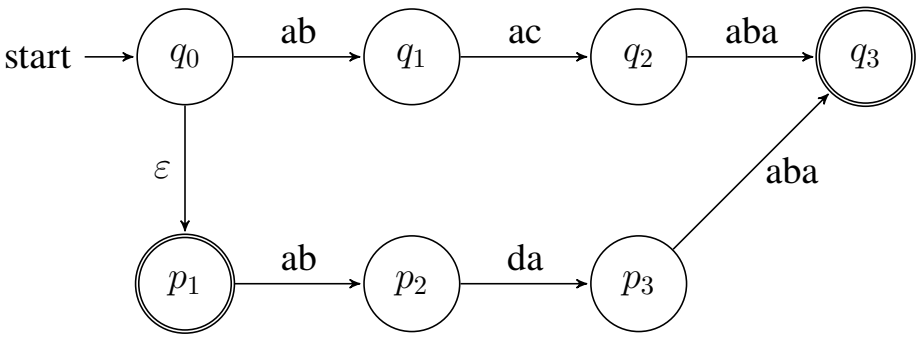
Листинг 2: Brzozowski's minimization algorithm

```
Automaton a = this;
if (!a.isDeterministic())
    a = a.determinize();
a = a.reverse();
a = a.determinize();
a = a.reverse();
return a;
```

Алгоритм минимизации такой - сначала проверяем, является ли автомат детерменированным, то есть что нет двух исходящих переходов из одной вершины с одинаковыми символами на них. А также что нет пустых переходов. Если есть, то склеиваем эти 2 вершины. Это операция детерминирования

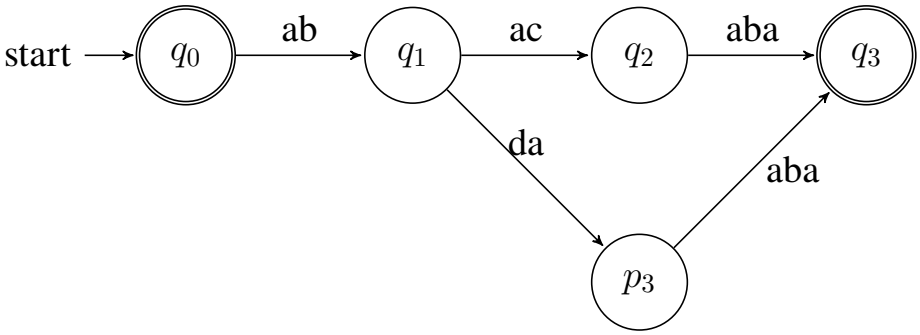
Затем, чтобы избавиться от двух одинаковых входящих переходов, мы делаем reverse, то есть перенаправляем все ребра и помечаем конечные вершины начальными и наоборот. Теперь вызываем еще раз determinize. Повторив еще раз reverse, получим начальный автомат, но без лишних переходов и вершин

Рассмотрим пример минимизации такого автомата:

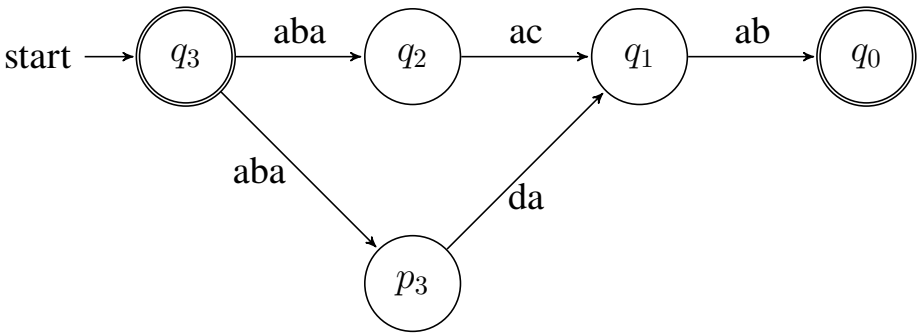


Сначала происходит детерминация, при которой q_0 и p_1 склеиваются в одну, которая становится терминальной. Затем так как из полученной вершины 2 одинаковых ребра, то вершины, в которые они ведут, склеиваются. Получаем такой автомат

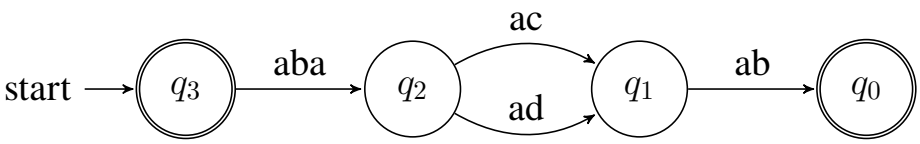
determinize



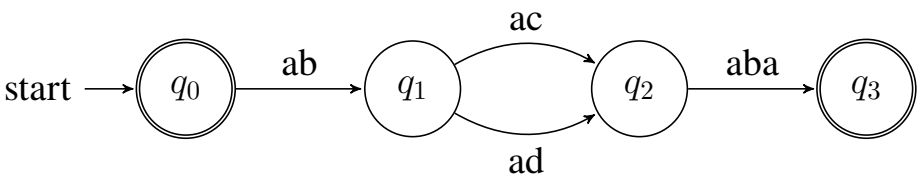
reverse



determinize



reverse



Получили упрощенную версию начального автомата

explode

Для реализации сложных операций на строках, такие как Replace и Substr, в которых нужно искать подстроки в автоматах, есть операция, которая переводит автомат к конечному автомату с переходами по буквам. Она просто заменяет каждое ребро на путь, где на каждом переходе лишь один символ вместо целого слова. Также есть обратная операция, которая схлопывает вершины с одним входящим и одним выходящим ребром в один переход

collapce

Обратная к explode операция, которая схлопывает пути из букв в одно ребро со словом. Реализуется при помощи DFS, находим всевозможные пути и каждый разбираем отдельно, проверяя у каких вершин одно входящее и одно выходящее ребро, то есть можно схлопнуть, удалив ее и объединив 2 ребра в одно, сконкатенировав слова на них. Стоит вызывать каждый раз после исполнения операций с использованием explode для уменьшения затрат памяти и размера автоматов

getLanguage

Собираем все слова, порождаемые автоматом, с помощью прохода DFS. Если в графе есть цикл или $\top$ ребро, то возможных слов бесконечно, и этот случай нужно отдельно разобрать. Эта операция полезна для оптимизаций, когда вместо выполнения громоздких операций над автоматом проще выписать все слова и по отдельности разобрать

Листинг 3: Язык, порождаемый автоматом

```
getLanguage(cur, str):
    if (hasCycle or acceptsTop) return null
    if (cur is terminal) result.add(str)
    for next in cur.getNext():
        getlanguage(next, str + transition(cur, next).symbol)
    return result
```

Операции для решеток

Операции **join** и **meet** аналогичны union и intersect для автоматов

strictEquals проверяет то, что есть хоть одно общее слово. Можно сделать при помощи операции пересечения для автоматов, то есть что оно не пустое. Однако в общем случае это громоздкая операция, так как требует 3 раза вызвать дополнение и объединение. Поэтому для конечных автоматов (без циклов и $\top$ переходов) выгоднее сравнить 2 их языка как множества слов

Листинг 4: Поиск общего слова

```
l1 = this.getLanguage()
l2 = other.getLanguage()
if (l1 != null and l2 != null)
    return l1.intersection(l2).isEmpty()
return this.intersect(other).acceptsEmptyLanguage()
```

equals проверяет, что 2 элемента решетки равны, или в данном случае что автоматы порождают одинаковые языки

Листинг 5: equals

```
l1 = this.getLanguage()
l2 = other.getLanguage()
if (l1 != null and l2 != null)
    return l1.equals(l2)
return this.subset(other) and other.subset(this)
```

weakerThen проверяет, что один элемент решетки строго выше другого, то есть язык одного автомата содержится в другом

Листинг 6: weakerThen

```
l1 = this.getLanguage()
l2 = other.getLanguage()
if (l1 != null and l2 != null)
    return l2.containsAll(l1)
return this.subset(other)
```

Операции на строках

Length

Возвращает интервал $[c_1, c_2]$, такой что $c_1 \leq |s| \leq c_2$. При этом если s содержит в себе цикл, или же содержит в себе переход, на котором стоит $\top$, то есть любая строка, то максимальная длина будет сколь угодно большой. Минимальная при этом заменяет все $\top$ на пустые строки

$$\text{length}(s) \triangleq \begin{cases} [|\text{minPath}(A)|, +\infty] & \text{if cyclic}(A) \vee \text{readsTop}(A) \\ [|\text{minPath}(A)|, |\text{maxPath}(A)|] & \text{otherwise} \end{cases}$$

Для подсчета длины запускаем DFS из начальных стейтов. При вхождении в терминальный, считаем длину пути (заменяя $\top$ на 0 или $+\infty$ соответственно) и пересчитываем максимальную и минимальную длину

Если автомат содержит в себе цикл, то с помощью DFS мы его найдем и максимальная длина пути будет бесконечной, так как можно по этому циклу крутиться сколь угодно много раз

Concat

При конкатенации нужно соединить все конечные состояния первого автомата с начальными состояниями второго ребрами с пустыми строками на них. И затем убрать конечные метки с терминальных вершин первого и начальные метки с начальных вершин второго

Другой вариант реализации немного отличается добавлением дополнительной вершины, в которую будут вести необходимые ребра. В таком случае количество ребер будет суммой конечных и начальных вершин, а не их произведением

Contains

Абстрактная семантика contains должна возвращать значение $true$, если любая строка из A' содержится в любой строке из A, значение $false$, если какая-то строка из A' не содержится в какой-то строке из A и $\{true, false\}$ ($\top$) в остальных случаях

$$\text{contains}(s, s') \triangleq \begin{cases} \text{false} & \text{if } A' \sqcap \text{FA}(A) = \text{Min}(\emptyset) \\ \text{true} & \text{if } \neg\text{cyclic}(A) \wedge \text{singlePath}(A') \\ & \wedge \forall \pi \in \text{paths}(A).\sigma_{sp} \curvearrowright_s \sigma_\pi \\ \{\text{true}, \text{false}\} & \text{otherwise} \end{cases}$$

Для начала определим, в каком случае нужно возвращать $false$. Для этого определим фактор-автомат $FA(A)$, который принимает все подстроки A. Для этого нужно просто пометить все стейты конечными. Затем пересечь $FA(A)$ и A', чтобы проверить, что никакая подстрока любой строки из A не является строкой из A'

Теперь, в каком случае $true$. Если A' содержит 2 слова, ни одно из которых не является префиксом другого, то они оба не могут быть одновременно префиксами какого-либо слова из A. Значит все слова в A' должны быть по цепочке включены друг в друга. Или же автомат для них выглядит как путь, некоторые вершины на котором конечные. И в этом случае достаточно проверить, что самое длинное слово входит во все слова из A, тогда и его префиксы будут

Во всех остальных случаях точного ответа нет, поэтому возвращаем $\{true, false\}$

Substr

Возвращаемое значение — автомат, который порождает нужные подстроки. Строить его будем так, что сначала выделим нужные пути, а затем склеиваем все полученные пути, добавляя одну общую начальную, соединенную с началами путей. В конце минимизируем

$$\mathrm{substr}(s, a_1, a_2) \triangleq \bigsqcup \mathrm{Min}\left(\{\sigma \mid (\sigma, 0, 0) \in \mathbf{Sb}(\mathbf{r}, i, j - i)\}\right)$$

Делаем рекурсивный обход в глубину, запоминая длину текущего пути. При переходе по ребру, после которого длина накопленного пути станет больше, чем индекс начала обрезания подстроки, режем ребро в нужном месте (чтобы в месте разреза было начало подстрок), и помечаем этот разрез началом. Иногда разрезать не придется, когда длина ровно совпадает с индексом начала подстроки. Далее идем вглубь, пока длина не станет больше конечного индекса, и аналогично обрезаем и помечаем конечной. Если на пути встречаем $\top$, то на этом шаге останавливаемся, добавляя в конец подстроки $\top$ с меткой конца. Действительно, $\top$ ребро означает сколь угодно длинную любую строку, а значит за его пределы не всегда получится выйти

Если в графе есть цикл, то по нему можем проходить сколько угодно раз, то есть одну вершину можем посетить несколько раз, это никак не запрещается. При этом анализ все равно будет конечным, так как длина пути при каждом переходе увеличивается, и есть ограничение сверху

Однако выбывают случаи, когда ограничения нет, то есть просто обрезание начала без указания длины. В этом случае вместо поиска путей нужной длины, мы находим вершины начала обрезанных подстрок, то есть проделываем те же начальные операции для поиска индекса начала. Но далее обходом в глубину определяем подграф для этой вершины, то есть куда можем из нее попасть. Получаем вместо объединения путей объединение автоматов

Replace

Перед началом поиска участков для изменения нужно привести автомат к автомату с переходами по буквам, то есть вызвать explode. Без этого проблематично заменять части путей, так как одно ребро со словом может участвовать в нескольких путях, и при этом резать его нужно будет в двух или более разных местах в зависимости от пути

Следующий шаг — перебор всех путей и запоминание тех, на которых встретился паттерн, который будет заменяться. Если таких путей не оказалось, значит возвращаем изначальный автомат. Если пути есть, то будем их перебирать отдельно, заменяя найденный паттерн на требуемый. В конце склеиваем все пути. Затем стоит вернуть автомат к виду с переходами по словам, а не буквам. Для этого вызываем обратную explode операцию — collapse

В отличие от Substr, здесь не получится корректно обрабатывать $\top$ переходы и циклы. Дело в том, что для циклических автоматов не получится выписать все возможные пути, так как их бесконечно много. А если цикла 2, то возможно такое, что при проходе первого например 2 раза, при проходе второго 3 раза, получаем строку, содержащую необходимый для замены паттерн, но при других количествах проходов этих циклов паттерна нет. Поэтому для всех бесконечных автоматов (порождающих бесконечные языки) возвращаем $\top$

Листинг 7: Replace алгоритм

```
Data: A, σ, R (substitutes strings of σ with R inside A)

foreach π ∈ paths(A):
    if σ ∈ π:
        foreach q_i →_σ q_j ∈ π:
            δ = q_1 → q_i + R + q_j → q_end
            result = result ∪ δ
    else result = result ∪ π

return minimize(result)
```

IndexOf

Реализуется аналогично Replace, так как тут тоже нужно находить интересующий паттерн в автомате. То есть сначала делаем explode, затем перебираем все пути, выделяем из них интересующие. Каждый отдельно обрабатываем, запоминая минимальный и максимальный индексы, где встретился искомый паттерн. В конце возвращаем интервал, в котором будет содержаться значение indexOf. Либо если таких путей нет, возвращаем $[-1, -1]$. По аналогичным Replace причинам, циклические автоматы и автоматы с $\top$ переходом не получится обработать, то есть для них возвращаем $[-1, +\infty]$

$$\text{indexOf}(s, s') \triangleq \begin{cases} [-1, +\infty] & \text{if cyclic}(A) \vee \text{cyclic}(A') \vee \text{readsTop}(A') \\ [-1, -1] & \text{if } \forall \sigma' \in \mathcal{L}(A') \nexists \sigma \in \mathcal{L}(A). \sigma' \curvearrowright_s \sigma \\ \bigsqcup\limits_{\sigma \in \mathcal{L}(A')} \text{IO}(A, \sigma) & \text{otherwise} \end{cases}$$

Остальные операции

В статье TARSIS описан алгоритм только для 6 операций, тогда как на деле их гораздо больше. Часть из них похожа на описанные сверху, какие-то понятно как реализовывать. Но есть и более сложные. Разберем некоторые из них

startsWith, endWith — Эти операции можно реализовать при помощи substr и contains. Для этого сначала обрезаем с помощью substr конец, то есть символы, которые находятся дальше от начала, чем длина паттерна, переданного в startsWith. А затем с помощью contains проверяем, что начало может включать в себя паттерн. Для endsWith просто сначала делаем reverse, и затем те же самые операции

charAt, charCodeAt — Частный случай substr длины 1. Для charCodeAt нужно дополнительно преобразовать символы в числовой интервал

trim, toUpperCase — Перебираем все переходы в автомате и для каждого слова на них отдельно проделываем операцию. Затем стоит минимизировать, так как большой шанс, что появятся дублирующиеся ребра

Листинг 8: startsWith, charAt, trim

```
startsWith(pattern):
    begin = this.substr(0, pattern.length)
    return begin.contains(pattern)

charAt(idx):
    return this.substr(idx, 1)

trim():
    foreach t in transitions:
        t.symbol = t.symbol.trim()
```

pad

Дополняет строку до нужной длины определенными символами. Проблема в том, что слова в автомате имеют разную длину. И одно и то же начальное ребро может содержаться в словах разной длины. Другая проблема это $\top$ ребра, у которых может быть любая длина. И наконец циклы, с которыми такая же проблема. Поэтому в общем случае результат операции не удастся посчитать

Решение — для конечных автоматов выписать весь язык, для каждого слова отдельно проделать операцию, и затем все слова склеить. Для бесконечных возвращаем $\top$

toBool

Приведение строки к $true/false$. По правилам JavaScript, пустая строка и '0' приводятся к false, все остальные true. То есть задача проверить, встречаются ли в автомате эти 2 строки. Если их нет, то ответ $true$. Если же автомат только из них состоит, то $false$. В остальных случаях $\{true, false\}$

strLt

Нужно сравнить максимальную возможную строку с минимальной попарно в обоих автоматах. Для этого сначала их нужно найти. Первым шагом делаем explode, чтобы была возможность выбирать самый маленький или самый большой символ для перехода. Затем шагаем жадно, на каждом шагу переходя по нужному символу. Если приходим к $\top$ символу, то в случае максимальной заполнен до бесконечности самым большим символом, в случае минимальной - заполнен минимальным. Если находим цикл, то по нему выгодно пройти бесконечно много раз. То есть стоит обрезать конец строки, добавив к последнему символу 1, получив ограничение сверху (либо просто обрезать для минимальной строки, получив ограничение снизу)

Сравнив 2 пары, либо получим однозначно, что один автомат порождает язык, строго превосходящий другой, и тогда ответ будет $true$ или $false$. Либо ответ, что бывает и так и так, то есть $\{true, false\}$

Листинг 9: strLt

```
minString():
    while !cur.isTerminal():
        if cur.hasTopTransition:
            return π
        next = cur.minTransition()
        if next is used:
            return π
        π += transition(cur, next)
        cur = next
    return π

maxString():
    while !cur.isTerminal():
        if cur.hasTopTransition:
            return π + 1
        next = cur.maxTransition()
        if next is used:
            return π + 1
        π += transition(cur, next)
        cur = next
    return π

strLt(l1, l2):
    l1 = l1.explode()
    l2 = l2.explode()
    if (l1.minString() > l2.maxString())
        return false
    if (l2.minString() > l1.maxString())
        return true
    return {true, false}
```

toNum

Приведение строки к числу. Для абстрактных значений нужно вернуть интервальную решетку для чисел, то есть ограничение сверху и снизху на значение. А также проверить, что строки состоят из корректных символов, то есть цифр и возможно минус в начале. Сделаем это первым шагом, проверив каждый переход отдельно

Далее нужно найти минимальное и максимальное возможное принимаемое значение. В отличие от strLt, длина числа имеет больший приоритет, чем первая цифра. Поэтому нужно искать наиболее короткое число, либо наиболее длинное. Это значит, что если есть цикл или $\top$, значит получаем сколь угодно большое. Если же цикла нет, то далее нужно перебрать всевозможные числа и выбрать из них наибольшее и наименьшее

Листинг 10: toNum

```
toNum():
    this.explode()
    foreach π ∈ paths(A, ⊤ → ⊘):
        if π has minusFirst:
            maxNum = max(maxNum, π)
            if π has cycle or top: minNum = -∞
            else minNum = min(minNum, π)
        else:
            minNum = min(minNum, π)
            if π has cycle or top: maxNum = ∞
            else maxNum = max(maxNum, π)

    return Int.Interval(minNum(), maxNum())
```

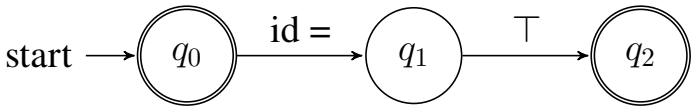
Оператор расширения

Важным методом является оператор расширения (widening). Именно благодаря ему удается завершать циклы за конечное время, создавая аппроксимацию. Рассмотрим пример того, как это реализованно

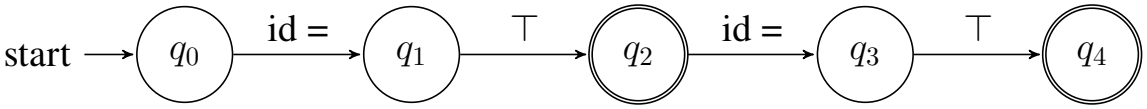
Листинг 11: Пример применения widening

```
function f(v) {
    res = "";
    while(?) res = res + "id = " + v;
    return res;
}
```

Автомат, задающий значения res в начале второй итерации цикла while, выглядит так:

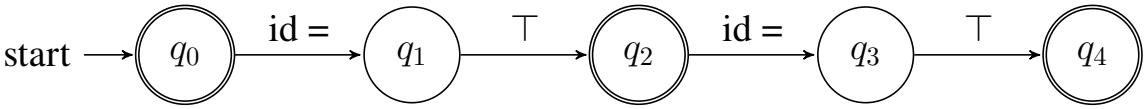


В конце второй итерации вот так:

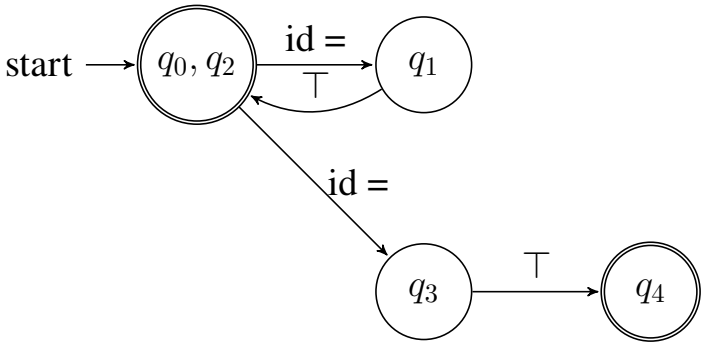


Далее, для этих двух автоматов приминяем саму операцию расширения. Алгоритм такой:

- Делаем union для этих автоматов:

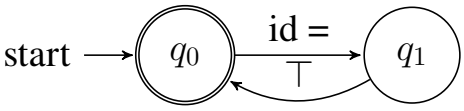


- Стейты, у которых исходящие пути длины 2 совпадают, объединяем. В данном случае q_0 и q_2 оба принимают $id = \top$, и потому склеиваются в одну вершину:

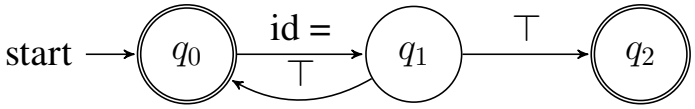


  В общем случае длину пути для склейки вершин можно регулировать, чем она больше, тем точнее результат, однако и размер автомата больше

- Минимизируем полученный автомат:



Теперь заметим, что полученный автомат является неподвижной точкой. Для этого попробуем проделать ещё одну итерацию цикла. После объединения автоматов в начале и конце цикла получаем такой:



Склеивать вершины не придется, однако при минимизации q_0 и q_2 станут одной, то есть получим начальный автомат, что означает отсутствие изменений и значит можно остановиться

Доказано [15], что такой алгоритм расширения соответствует условиям оверапроксимации

Тестирование

Были написаны тесты на новую финкциональность, разберем несколько примеров

Листинг 12: Пример CWE тестов

```
function tossCoin(): string {
    if (Math.Random() > 0.5) {
        return "eagle"
    } else {
        return "tails"
    }
}

if (tossCoin() == "edge") {
    /* POTENTIAL FLAW GOOD: */
    console.log(document.getElementById('source').value)
}
```

В данном примере в консоль выводятся помеченные данные (то есть те, которые влекут за собой уязвимость), если выполняется условие. Для определения абстрактного значения `tossCoin()`, происходит join двух веток исполнения. В старой строковой решетке результатом был класс обычных строк (SNormal). Однако, “edge” также может принадлежать этому классу, из-за чего сравнение на эквивалентность строк должно возвращать $\{true, false\}$. И тогда ветка исполнения, в которой в консоль выводятся опасные данные, также возможна. И значит анализатор указывал на то, что в этой строчке возможна уязвимость, тогда как на деле этого не происходит

С новой решеткой мы точно можем сказать, что ни одно возвращаемое знаение функции `tossCoin()` не равно “edge”, а следовательно мертвый код игнорируется. Потому можем поставить метку `POTENTIAL FLAW GOOD`, означающую отсутствие уязвимости

Листинг 13: Пример IR тестов

```
function tossCoin(): string {
    if (Math.Random() > 0.5) {
        return "eagle"
    } else {
        return "tails"
    }
}

let x = tossCoin()
console.log(x.indexOf("a")) // [number: [1, 1]]
console.log(x.indexOf("e")) // [number: [-1, 4]]
console.log(x.indexOf("r")) // [number: [-1, -1]]
```

Другой пример тестов это IR (intermidiate representation) тесты. Они проверяют то, какие абстрактные значения принимают переменные. По аналогичным причинам старая решетка не могла как либо оценить значение `indexOf`. Новая же позволяет это сделать. Благодаря этому, не только строковые значения стали более точными, но и некоторые числовые

Листинг 14: Еще пример IR тестов

```
var x = "a" + document.getElementById('source').value + "c"
console.log(x)                  // [string: a⊤c tags=[XSS,WEB]]
console.log(x.length)           // [number: [2, +Inf]]
console.log(x.startsWith("a")) // [boolean: true]
console.log(x.substr(0, 1))     // [string: a tags=[XSS,WEB]]
```

В данном примере видно, что даже если в операциях участвует $\top$, все еще можно знать что-то о строке, как напимер минимальную длину или то, что `startsWith('a')` это всегда true

Также можно увидеть `tags=[XSS,WEB]`. Это как раз и есть метка, которая указывает на уязвимые данные. И при некоторых операциях она сохраняется, как например конкатенация. Одним из уточнений в будущем может стать то, что эта метка будет принадлежать конкретному переходу, а не всему абстрактному значению. То есть например если сейчас сделать `x.substr(0, 1)`, то результат будет 'a', то есть вообще не содержать опасных символов. Однако метка будет передана

Результаты

Для подсчета итоговой статистики были произведены запуски анализатора с использованием старой и новой решеток на внутренних тестовых проектах, а также на Open Source JavaScript проектах.

Таблица 1: Сравнение результатов анализа тестовых проектов

Тип	Возвращаемое	Было	Стало	Итог
Concat	Top	319,381	25,331	**0.08**
	Const	58,321	58,767	1.01
	Automaton	0	298,822	-
String	Top	4587	3610	0.78
	Const	795	1363	**1.71**
	Automaton	0	201	-
Number	Top	823	426	0.52
	NonTop	477	801	**1.68**
Boolean	Top	15,470	14,326	0.93
	NonTop	199	287	**1.44**

Тут все функции над строками разбиты на 4 категории

1. Concat — конкатенация строк. Самая частая функция над строками, так как это и оператор +

2. String — фукции над строками, возвращаемое значение которых это строка: substr, charAt, replace, trim и т.д.

3. Number — функции, возвращающие число: length, charCodeAt, indexOf, toNumber

4. Boolean — функции, возвращающие true/false: toBool, startsWith, strLt, includes

Можно видеть, что точность выполнения конкатенации возрасла в 12 раз. Это связано в первую очередь с тем, что часто в коде встречаются конструкции вида `console.log(‘Hello ’’ + name)`. Раньше такая конструкция

сразу становилась $\top$, то есть терялась точность. Сейчас же есть возможность понимать, что первые 5 символов это “Hello”, а значит и значения вызовов `startsWith(‘‘Hell’’)` или `substr(0,3)` вернут константное значение, а не $\top$

Другой важный момент, что количество возвращаемых функциями констант возросло на от 44% до 71% в зависимости от типа операции. Самое важное увеличение это Boolean, так как чаще всего именно эти функции проходят проверку в операторах ветвления, позволяя определить dead код

Таблица 2: Сравнение производительности

Тип	Метрика	Было	Стало	Итог
Суммарно	Время (мин)	28:25	33:23	1.18
	Состояния	35.7 млн	35.3 млн	0.99
Малые	Время	0:22	0:22	1.00
до 30 сек	Память	4к МБ	4.3к МБ	1.07
Средние	Время	1:19	1:38	1.24
0.5-5 мин	Память	7.1к МБ	6.8к МБ	0.96
Большие	Время	8:26	8:46	1.04
от 5 мин	Память	17.9к МБ	16.7к МБ	0.93

Суммарное время исполнения увеличилось на 18%, что является разумной платой за увеличение точности. При этом количество проанализированных состояний стало меньше лишь на 1%. Это связано с тем, что не во всех проектах часто используются строки. Ниже разберем тот, в котором их много. А также с тем, что благодаря оператору расширения обработка строк в циклах стала точнее, а потому итераций может быть чуть больше

Проекты разбиты на 3 группы. Первая группа — маленькие проекты. Среднее время на них не изменилось, но потребление памяти стало чуть больше (<10%). Вторая группа — средние проекты, на которых время анализа больше 30 секунд. Заметная прибавка ко времени анализа (24%), но в рамках разумного. При этом потребление памяти уменьшилось на 4%. Третья группа — большие проекты, время анализа больше 5 минут. Такой проект один, разберем его подробней

Таблица 3: Сравнение результатов анализа одного проекта

Метрика	Было	Стало	Итог
Всего состояний	10,5 млн	10,1 млн	**0.96**
Concat: Top	100,419	10,503	0.10
Concat: Const	12,849	12,913	1.01
Concat: Automaton	0	89,357	-
String: Top	430	379	0.88
String: Const	100	128	1.28
String: Automaton	0	34	-
Number: Top	221	163	0.74
Number: NonTop	52	98	1.88
Boolean: Top	3,716	3,412	0.92
Boolean: NonTop	5	10	2.00

Самый большой проект в пуле тестовых, в котором большое количество работы со строками. Размер 250к строк. Время анализа составило 506к ms (8 минут 26 секунд) на старой решетке и 526к ms (8:46) на новой, то есть прирост 4%. Потребление памяти было 17935 МБ на старой и 16706 МБ на новой, то есть уменьшилось на 7%

Снижение обусловлено тем, что благодаря повышению точности было проигнорировано больше состояний программы из-за их недостижимости. Состояние программы — строчка кода вместе с состоянием переменных, то есть возможных значений, на момент её исполнения. Это число коррелирует с количеством проанализированных строк, но например для циклов одна строка может быть исполнена несколько раз при итерировании. Или одна функция может быть вызвана из разных мест с разными аргументами

Анализатор сохраняет все состояния во время анализа, и это потребляет очень много памяти. Когда память переполняется, вызывается Garbage Collector [23], который тратит много времени. Но благодаря уменьшению возможных состояний на 4% в данном случае, удается сохранить больше памяти, что в том числе влияет на производительность

Таблица 4: Сравнение результатов анализа Open Source

Метрика	Было	Стало	Итог
Время (сек)	159	192	1.20
Состояния	5.47 млн	5.29 млн	**0.97**
Concat: Top	17,457	3,908	0.22
Concat: Const	2038	2029	0.99
Concat: Automaton	0	13,247	-
String: Top	130	117	0.90
String: Const	55	62	**1.13**
String: Automaton	0	12	-
Number: Top	24	5	0.20
Number: NonTop	3	22	**7.33**
Boolean: Top	3090	3010	0.97
Boolean: NonTop	3	8	**2.67**

Для запуска анализа на open source проектах были выбраны 6 открытых JavaScript проектов на github, связанных с сериализацией и парсингом текстов

- https://github.com/lightswitch05/table-to-json/tree/master
- https://github.com/defunctzombie/form-serialize
- https://github.com/eran-pinhas/osg-serializer-js/tree/master
- https://github.com/SheepSeb/H-Scope
- https://github.com/postlight/parser/tree/main
- https://github.com/markedjs/marked

Результаты анализа на них подтверждают прирост точности, хотя и количество пропущенных строк незначительно. Можно также заметить, что количество возвращаемых константных строк при конкатенации стало немного меньше. Вполне вероятно, что среди пропущенных строк мог быть отчет об ошибке исполнения, но благодаря уточнению анализа удалось проверить недостижимость этой ветки

Заключение

Результатами данной работы является реализация уточненной строковой решетки для абстрактного интерпретатора, улучшение результатов статического анализа при обработке строк, тестирование новой функциональности и проведение замеров на тестовых и реальных проектах.

- Удалось повысить точность строковых операций в 1.5-2.5 раза в среднем
- Суммарное время исполнения увеличилось не более, чем на 20%
- Количество потребляемой памяти в больших проектах уменьшилось за счет сокращения количества проанализированных состояний
- Обработка строк, полученных конкатенацией с неизвестной строкой. Ранее такие строки считались произвольными, теперь часть символов известна
- Добавлен оператор расширения, позволяющий проводить анализ строк при обработке циклов
- Разработаны и протестированы алгоритмы для тех операций, которых нет в статье — сравнение строк, преобразование строки к числу и т.д.
- Новая функциональность позволяет избежать ложных срабатываний в тех местах, которые раньше было невозможно грамотно обработать

Также в планах дальнейшее улучшение решетки. Один из вариантов — использование классов из JSAI решетки. То есть дополнительная информация на $\top$ ребра, если содержит в себе только цифры, или же имеет фиксированную длину. Другой вариант — метки об уязвимостях ставить на переходы, а не весь автомат. Это позволит хранить меньше потенциально опасных переменных после обрезания подстрок

Список литературы

[1] https://www.russianadvertisingmagazine.com/article/1779

[2] Cousot P., Cousot R. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints* // POPL. 1977. https://doi.org/10.1145/512950.512973

[3] King J.C. *Symbolic Execution and Program Testing* // Communications of the ACM. 1976. https://doi.org/10.1145/360248.360252

[4] Cadar C., Sen K. "Symbolic Execution for Software Testing"// IEEE Software. 2013.

[5] Baldoni R. et al. "A Survey of Symbolic Execution Techniques"// ACM Computing Surveys. 2018.

[6] de Moura L., Bjørner N. "Z3: An Efficient SMT Solver"// TACAS. 2008.

[7] Barbosa H. et al. "cvc5: A Versatile and Industrial-Strength SMT Solver"// TACAS. 2022.

[8] Blanchet B. et al. "A Static Analyzer for Large Safety-Critical Software"// PLDI. 2003.

[9] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/2635868.2635904

[10] Gange G, Navas J.A, Stuckey PJ, Søndergaard H, and Schachte P. Unbounded model-checking with interpolation for regular language constraints. In Tools and Algorithms for the Construction and Analysis of Systems, vol. 7795 of LNCS, pp. 277–291, 2013.

[11] Madsen M, and Andreasen E. String analysis for dynamic field access. In: Compiler Construction, vol. 8409 of LNCS, pp. 197–217 (2014).

[12] Costantini G, Ferrara P, and Cortesi A. A suite of abstract domains for static analysis of string values. Software Practice and Experience, 2015;45(2):245–287.

[13] Arceri, V., Mastroeni, I., Xu, S.: Static analysis for ecmascript string manipulation programs. Appl. Sci. 10, 3525 (2020) https://doi.org/10.3390/app10103525

[14] Arceri V. et al. *Abstract Domains for String Analysis* // ACM Computing Surveys. 2022. https://doi.org/10.48550/arXiv.2006.02715

[15] D'Silva, V.: Widening for Automata. MsC Thesis, Inst. Fur Inform.- UZH (2006)

[16] Babel is a JavaScript compiler https://babel.dev/docs/

[17] ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees https://www.antlr.org/

[18] An intermediate representation (IR) is the data structure or code used internally by a compiler or virtual machine to represent source code. An IR is designed to be conducive to further processing, such as optimization and translation https://llvm.org/devmtg/2017-06/1-Davis-Chisnall-LLVM-2017.pdf

[19] Control Flow Graph In Software Testing https://medium.com/@amaralisa321/control-flow-graph-in-software-testing-f12756a5b88c

[20] https://pvs-studio.com/en/blog/terms/6496/

[21] https://users.math-cs.spbu.ru/~okhotin/teaching/tcs_fl_2021/okhotin_tcs_fl_2021_l3.pdf

[22] https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%91%D1%80%D0%B6%D0%BE%D0%B7%D0%BE%D0%B2%D1%81%D0%BA%D0%BE%D0%B3%D0%BE

[23] https://medium.com/@rahulbaghel_40/garbage-collector-550cfee7de72