Московский Государственный Университет им. М. В. Ломоносова
Факультет Вычислительной Математики и Кибернетики

Статический анализ достижимости для программ на языке JavaScript

Дипломная работа
студента 522 группы
Довгаль Сергея Сергеевича

Научный руководитель
кандидат ф.-м. наук
Дмитрий Дмитриевич Козлов

Москва, 2011

Содержание

1 Введение

Язык JavaScript создавался для написания программ, добавляемых в текст веб-страницы и исполяемых внутри веб-браузера. Внедрение программного интерфейса DOM, предоставляющего программам удобный доступ к содержимому HTML документов, и технологии AJAX, позволяющей асинхронно обмениваться данными с веб-сервером, сделало язык самым используемым инструментом для создания веб-приложений. Этот факт и последующая стандартизация позволила языку проникнуть в области прикладного и серверного программирования.

В результате широкого распространения выявились и подверглись критике недостатки языка [1]. Двумя главными недостатками являются низкая скорость выполнения, свойственная всем интерпретируемым языкам, и проблемы безопасности, возникающие из-за следующих свойств языка:

- В языке реализована прототип-ориентированная парадигма программирования, согласно которой, в отсутствует понятие класса, но существует понятие объекта. Объекты могут создаваться либо заново, перечислением своих методов и атрибутов, либо путём клонирования существующих. Для большей гибкости в язык добавлена возможность добавлять, модифицировать и удалять методы и поля объекта во время исполнения.
- Язык является интерпретируемым, с возможностью во время выполнения формировать произвольную текстовую строку и, при помощи конструкции eval, интерпретировать её как программу, либо, при помощи конструкции Function, использовать её для определения тела функции.
- В языке используется слабая динамическая типизация, допускаются неявные преобразование объектов любого типа.
- В языке области видимости представлены в виде объектов. При запуске программы создается глобальный объект, методами которого являются библиотечные функции и функции взаимодействия с интерпретатором. Возможность переопределять методы объектов во время выполнения позволяет заместить вызов безопасной библиотечной функции произвольным кодом.

На текущий момент существует несколько подходов к решению проблемы безопасности программ на JavaScript. Самым простым является введение безопасного подмножества языка. Однако, из-за существенного снижения возможностей языка, применимость этого подхода ограничена. Другим подходом является инструментирование кода. Добавление внешней программы, следящей за выполнением набора условий в процессе интерпретации JavaScript не всегда возможно и усугубляет второю большую проблему языка – скорость интерпретации. Еще одним подходом является автоматической преобразование кода программы написанной на более безопасном языке, например Java, в JavaScript. Этот подход нельзя применить к уже существующим програмам. Кроме того, он приводит к появлению избыточного и мёртвого кода, что как и в предыдущем подходе, снижает скорость.

В последнее десятилетие из-за роста популярности “скриптовых” языков, в том числе JavaScript, были предприняты попытки использования средств статического анализа, хорошо зарекомендовавших себя при решении проблем компилируемых и статически типизированных языков, для решения проблемы безопасности динамических языков. При анализе *JavaScript* авторы столкнулись с многочисленными проблемами, возникающими из-за свойств языка, перечисленных в следующей главе. Зачастую, чтобы произвести корректный анализ, рассматривалось подмножество языка.

2 Цели и задачи дипломной работы работы

Целью дипломной работы является построение статического анализатора для программ на языке `JavaScript`, который осуществлял бы контекстно-зависимый, интерпроцедурный анализ `JavaScript`.

К разрабатываемому анализатору предъявляются следующие требования:

- Статический анализатор должен уметь отвечать на вопрос, достижима ли заданная точка программы, при каких значениях входных данных.
- Статический анализатор должен поддерживать все конструкции языка.
- Необходимо обосновать ограничения на используемые конструкции языка, при которых можно сделать обоснования полноты и точности анализа.
- Анализ должен быть консервативным. При этом для тех конструкций, которые существенно снижают точность анализа должны выводиться соответствующие предупреждения.
- Для статического анализатора должен быть построен набор автоматических функциональных тестов, проверяющих правильность его работы.
- Должно быть проведено экспериментальное исследование анализатора на задаче *tainted mode*.
- В дипломной работе должен быть приведен краткий обзор близких работ.

3 Краткое описание семантики языка

В таблице перечислен краткий список архитектурных особенностей языка `JavaScript` стандарта ECMA-262 третьей редакции [2], важных с точки зрения статического анализа.

Интерпретируемость

`JavaScript` это интерпретируемый язык. Перед выполнением не преобразовывается в промежуточное представление.

Система типов

Все типы языка разделяются на элементарные типы и объекты. К элеметарным типам относятся `Undefined`, `Null`, `Boolean`, `String` и `Number`. Все объекты, как стандартные, включая массивы и функции, так и определённые пользователем наследуются от стандартного встроенного объекта `Object`. В языке используется слабая динамическая типизация. Почти все элементарные типы неявно преобразовываются друг в друга (кроме значений типа `Undefined`). Если переменная содержит значение типа `Boolean`, `String` или `Number`, то при необходимости оно преобразовывается в соответствующий ему стандартный объект.

Объект `Object`

С точки зрения спецификации, объект представляет собой множество членов, каждый из которых содержит следующие аттрибуты:

- `[[Name]]` – имя члена. Имеет тип `string`.
- `[[Value]]` – значение сопоставленное члену. Может быть любого типа.
- Множество из нуля или более следующих аттрибутов:
  - `[[ReadOnly]]` – имеет тип `boolean`. Если значение истинно, то этот член доступен только для чтения. Попытки записать в этот член будут проигнорированы.
  - `[[DontEnum]]`– имеет тип `boolean`. Если значений истинно, то член не будет перечисляться внутри цикла `for-in`.
  - `[[DontDelete]]` – имеет тип `boolean`. Если значение истинно, то все попытки удалить свойство будут проигнорированы.
  - Internal – внутренние свойства напрямую недоступные через операторы языка. Необходимы для описания поведения. Каждый объект имеет следующие внутренние свойства:
    - `[[Prototype]]` – указатель на прототип этого объекта. Его назначение описано в пункте *наследование*.
    - `[[Class]]` – строка содержащая класс объекта.
    - `[[Get]]` – функция, возвращающая значение указываемого свойства.
    - `[[Put]]` – функция, устанавливающая значение указываемого свойства.
    - `[[CanPut]]` – функция, возвращающая значение типа `boolean`, описывающее можно ли успешно изменить значение указанного свойства.
    - `[[HasProperty]]` – функция, возвращающая значение типа `boolean`, описывающее, существует ли уже у объекта свойство с данным именем.
    - `[[Delete]]` – функция, удаляющая из объекта указываемое свойство.

С точки зрения пользователя, объект представляется собой ассоциативный массив, ключами которого являются строки с именами аттрибутов и методов, а значениями являются соответствующие объекты. К свойству объекту в программе можно неявно, используя его имя (строку). В случае если свойства с запрашиваемым именем не существует, то возвращается значение по умолчанию. Поэтому, можно сказать, что объект отображает множество всевозможных строк на множество объектов.

Наследование

В языке реализована прототип-ориентированная парадигма программирования: отсут-

ствует понятие класса, но существует понятие объекта; объекты могут создаваться либо заново, перечислением своих методов и атрибутов, либо путём клонирования существующих. В последнем случае атрибуты и методы базового объекта разделяются между ним и производными объектами. Если атрибут базового объекта будет изменён, то эти изменения будут видны и в производных объектах. Однако, если в производном объекте попытаться изменить атрибут базового объекта, например при помощи присваивания, то в производном объекте неявно создастся одноименный атрибут, а атрибут базового объекта неизменится.

Это реализовано при помощи цепей наследования. Каждый объект содержит атрибут `prototype`, который ссылается либо на создавший его объект. Если объект был создан заново, перечислением свойств, то этот атрибут ссылается на стандартный объект `Object.prototype`, атрибут `prototype` которого ссылается на `null`.

При обращении к некоторому свойству объекта (то есть вызове метода `[[Get]]`) он сначала ищется среди собственных атрибутов объекта. Затем, если он не был найден, это свойство запрашивается у объекта-прототипа. Если запрашиваемое свойство найдено не было, то возвращается `undefined`.

Контекст выполнения

Когда управление передаётся выполняемому коду, осуществляется вход в контекст выполнения. Активные контексты выполнения логически формируют стек. Верхним контекстом выполнения в этом стеке является текущий контекст исполнения.

Для каждого контекста выполнения имеется связанный с ним *объект переменных*. Переменные и функции, определённые в исходном коде, добавляются в качестве свойств объекта переменных. Для кода функции параметры добавляются как свойства объекта переменных. Просмотр кода и добавление происходит при входе в контекст выполнения:

```
function foo() {
  if (false) {
    var a = 5;
  } else {
    print(a);
    a = 10;
    print(a);
  }
}

function bar() {
  print(a);
  a = 11;
}

a = 20;
foo();
bar();
```

После вызова первой функции будет выведено `undefined`, `10`, а после вызова второй функции будет выведено `20` и значение глобальной переменной `a` станет `11`. Во время интерпретации кода в определённом контексте выполнения *объект переменных* не изменяется

Иерархия областей видимости

Для каждого контекста выполнения имеется связанная с ним иерархия областей видимости. Иерархия областей видимости — это список объектов, в которых производится поиск при определении значения идентификаторов. Когда управление входит в контекст выполнения, создаётся иерархия областей видимости и сразу заполняется начальным набором объектов, зависящим от типа кода:

- При входе в глобальный код (например при запуске), в иерархию добавляется только глобальный объект, который так же будет *объектом переменных* и *объектом this*.

- При входе в функцию восстанавливается иерархия, хранящаяся во внутреннем свойстве `[[Scope]]` вызываемой функции. Это свойство сохраняется в момент определения функции. На вершину восстановленной иерархии помещается `объект активации`.

При разрешение идентификаторов поочередно просматриваются все объекты в иерархии. В случае, если идентификатор отсутствует, при попытке чтения происходит ошибка, а при записи происходит неявное создание атрибута в текщем `объектке this`. Причина по которой атрибут не создается в текущем *объекте переменных* — семантика инструкции `with`.

Во время интерпретации кода в определённом контексте выполнения на иерархию областей видимости могут влиять только инструкции `with` и `catch`.

Объект Function

```
function foo() {
  return bar();
}

function bar() {
  return new Date();
}

this.obj = condition() ? foo() : bar();
```

Значением члена `obj` объекта `this` будет ссылка на объект `Date`. Если условие `condition()` истинно, то объекту `Date` будет сопоставлен последовательность стековых фреймов `this` → `foo` → `bar`, если ложно то `this` → `bar`. Последовательность стековых фреймов объекта `this` не меняется в ходе этих операций.

Вызов функции

1. Иерархия областей видимости при инициализации содержит *объект активации*, за которым следуют объекты, сохранённой в свойстве [[Scope]] объекта Function.

   Когда управление входит в контекст выполнения кода функции, объект, называемый *объектом активации*, создаётся и связывается с контекстом выполнения. Объект активации инициализируется с именем свойства `arguments` и атрибутами { DontDelete }. Начальным значением этого свойства является *объект аргументов*, который инициализируется следующим образом:

   (a) Значение встроенного свойства [[Prototype]] для объекта аргументов равно первоначальному объекту-прототипу Object, т.е. тому, который представляет собой первоначальное значение Object.prototype

   (b) Создаётся свойство с именем callee и атрибутами { DontEnum }. Первоначальное значение этого свойства - объект Function, выполнение которого производится в данный момент. Это позволяет анонимным функциям быть рекурсивными.

   (c) Создаётся свойство с именем length и атрибутами { DontEnum }. Начальным значением этого свойства является число реальных значений аргументов, переданное при вызове.

   (d) Для каждого неотрицательного числа arg, меньшего значения свойства length создаётся свойство с именем ToString(arg) и атрибутом DontEnum . Начальным значением этого свойства является реальное значение соответствующего аргумента, переданное при вызове. Первое реальное значение аргумента соответствует arg = 0, второе - arg = 1 и так далее. В том случае, когда arg меньше количества формальных параметров объекта Function, значение свойства является общим с соответствующим свойством объекта активации. Это означает, что

изменение данного свойства изменяет соответствующее значение свойства у объекта активации и наоборот.

Объект активации затем используется в качестве *объекта переменных* при объявлении переменных.

Для программы на ECMAScript невозможно получить доступ к объекту активации. Она может получать доступ к полям этого объекта, но не к самому объекту.

Когда операция вызова применяется к значению Reference, базовым объектом которого является объект активации, в качестве значения this в таком вызове используется null.

2. Инстанциация переменных производится при помощи объекта активации в качестве объекта переменных и с использованием атрибутов свойств { DontDelete }.
3. Значение this передаётся вызывающим. Если значение this, переданное вызывающим, не является объектом (заметим, что null - не объект), то значением this является глобальный объект.

Функции это объекты первого класса

Функции являются объектами первого класса: их можно передать как параметр в другую функцию, возвращать как результат работы другой функции, присваивать переменной и создавать во время исполнения. Разрешение идентификаторов функций происходит так же как и идентификаторов переменных.

Интроспекция

В языке поддерживается интроспекция — возможно добавление и удаление атрибутов объекта во время исполнения.

Рефлексия

В языке поддерживается рефлексия возможность программы изменять свою структуру во время исполонения. Для этого используются инструкции `eval` и `Function`.

Исключения

В языке поддерживаются исключения и операция безусловного перехода. При входе в `try` и в `catch` блоки создаются новые объекты видимости и помещаются в цепь объектов видимости. Объекты исключений не различаются, поэтому поддерживается использование только одного `catch` блока. Блок очистки `finally` выполняется всегда, даже после операции безусловного перехода.

4 Существующие решения

Критерии различия работ: полнота анализируемого языка (весь язык или его подмножество), используемые методы статического анализа, использование дополнительных средств для анализа, назначение.

В работе [3] реализован первый *points-to* анализ языка для его оптимизации. Авторы напрямую применили идеи, использованные для анализа языка `C`. Анализ обладает следующими свойствами:

- анализ интрапроцедурный, без учета потока управления и контекста
- анализ производится на основе множеств ограничений [4]

Анализ производится для подмножества языка [**?**]. Из языка были полностью убраны инструменты поддерживающие рефлексию. Для улучшения точности каждое свойство объекта рассматривается отдельно. Это отличает его от анализа Андерсона, в котором массивы рассматриваются как единые сущности. Пусть свойствам некоторого объекта присваиваются некоторые значения, если обращаться к свойству как к элементу словаря, используя в качестве ключа динамически созданную строку, то можно изменить любое его свойство. Авторы перешли ограничили анализ этой проблемой.

В работе [5] реализован *корректный* анализ типов переменных программы. Анализатор типов применяется в средах разработки для обнаружения ошибок на ранней стадии написания программ, точного автодополнения и безопасного рефакторинга кода. Основным методом анализа является абстрактная интерпретация и вычисление неподвижной точки в монотонной структуре [6]. Высокая точность анализа достигается за счет создания подробной модели при абстрактной интрепретации и набора дополнительных техник:

- *recency abstraction* [7]
- *lazy propagation* [8]
- интерпроцедурный анализ с учетом контекста и потока управления
- абстрактная сборка мусора [9]
- выполнение *reference-to* анализа в процессе определения типов

В этой работе рассматривается весь язык, а не его подмножество.

В работе [10] предложен способ усиления безопасности программ на JavaScript. Для этого вводится набор политик, определяющих нежелательное поведение программы, например запрещающих переопределение встроенных функций. Нежелательное поведение выражается на специальном языке, оперирующим фактами, полученными в ходе статического анализа и выполнения инструментирования. Авторами реализован *корректный* points-to анализ программ из подмножества языка. На язык наложены следующие ограничения:

1. Язык не содержит инструкций обеспечивающих рефлексию – запрещены инструкции `eval` и `Function`.
2. Запрещено изменение свойства `innerHtml` элементов содержащихся в документе. В случае, когда исполняемый скрипт встроем в веб-страницу, при помощи интерфейса `DOM` можно получить доступ к объектам, соответствующим элементам веб-страницы. Свойство элемента `innerHtml` определяет код которым будет замещен этот элемент при отображении страницы.
3. Функция не является объектом первого класса (запрещается создавать тело функции во время выполнения)

4. Язык не содержит инструкции *with*, позволяющей временно изменять цепочку областей видимости

Инструментирование кода было использовано для выражения дополнительных требований и как альтернатива ограничению 2, являющемуся очень сильным.

5 Решение задачи

5.1 Общее описание

5.1.1 Вычисление достижимости

Из алгоритмической неразрешимости задачи останова машины Тьюринга следует, что невозможно в общем случае однозначно ответить на вопрос о достижимости точки программы. Чтобы ответить на этот вопрос с некоторой точностью, необходимо вычислить возможные значения условий в операторах ветвления и циклах. Для вычисления возможных значений всех переменных в каждой точке программы используется *абстрактная интерпретация* — метод статического анализа, производящий интерпретацию на абстрактной модели, построенной по анализируемой программе [6].
Его идея заключается в том, что интерпретация программы производится над абстрактным доменом данных. Выбирается он таким образом, чтобы содержать достаточно информации для формулирования интересующих свойств программы и не приводит к незавершимости анализа.

Далее будет описан алгоритм с рабочим списком, при помощи которого производится вычисления и задана абстрактная модель.

5.1.2 Алгоритм с рабочим списком

Для описания алгоритма с рабочим списком ниже введены определения из теории решеток. Во всех из них под S понимается произвольное множество.

Отношение частичного порядка — это отношение $\prec : S \times S \to \{true, false\}$, обладающее свойством рефлексивности, транзитивности и антисимметричности. Может быть определено не для всех пар множества.

Частично упорядоченное множество — множество, с определенным на нём отношением частичного порядка. Обозначается парой $(S, \prec)$.

Верхняя грань подмножества $T \subset S$ — это элемент $m \in S$, такой что $\forall \ell \in T : \ell \prec m$. Верхняя грань m называется наименьшей, если для любой другой верхней грани $\tilde{m}$ выполнено $m \prec \tilde{m}$

Нижняя грань подмножества $T \subset S$ — это элемент $m \in S$, такой что $\forall \ell \in T : m \prec \ell$. Нижняя грань m называется наибольшей, если для любой другой нижней грани $\tilde{m}$ выполнено $\tilde{m} \prec m$

Полная решетка — частично упорядоченное множество $(S, \prec)$, такое что любое подмножество из S имеет точную верхнюю и нижнюю грани. Под $\bot$ понимается наименьший элемент S, а под $\top$ — наибольший элемент S.

Монотонная функция $f : S \to S$ — это функция, сохраняющая отношение частичного порядка:

$$\forall \ell, \ell' \in S : \ell \prec \ell' \Rightarrow f(\ell) \prec f\left(\ell'\right)$$

Под $\mathcal{M}$ будет пониматься класс монотонных функций $f : S \to S$.

Неподвижная точка функции $f : S \to S$ — это точка ℓ, отображаемая функцией в саму себя: $\ell \in L : f(\ell) = \ell$.

Монотонная структура — пара $\mathcal{MF} = (L, F)$, где L — полная решётка, а F — множество монотонных функций.

Теорема о неподвижных точках в решетке $L = (S, \prec)$ конечной высоты каждая монотонная функция $f \in \mathcal{M}$ имеет минимальную неподвижную точку:

$$\begin{cases} fix(f) & = \bigcup f^n(\bot) \\ f(fix(f)) & = fix(f) \end{cases}$$

Анализ программы с графом потока управления $G = (V, E)$ при помощи монотонной структуры $\mathcal{MF}$ задается шестёркой $AnalysisInstance$:

$AnalysisInstance = (L, F, E, I, i, t, M, m, w)$

L – полная решетка $(S, \prec)$ из $\mathcal{MF}$

F – множество монотонных функций из $\mathcal{MF}$, называемых функциями переноса.

E – конечное множество пар (v_1, v_2), соответстующих рёбрам графа потока управления анализируемой программы при решении прямых задач, и инвертированным ребрам при решении обратных задач

I – конечное множество начальных состояний графа потока управления анализируемой программы

i – информация, известная в точках программы, соответствующих начальным состояниям, $i \in L$

t – отображение $t : V \to F$, ставит в соответствие точке программы функцию перевода в следующее состояние (функцию переноса)

Результатом работы алгоритма являются минимальные неподвижные точки решетки L для функций из F. Для описания алгоритма дополнительно вводятся следующие обозначения:

fst – функция $fst : S^* \to S$ возвращает первый элемент входного списка

snd – функция $snd : S^* \to S$ возвращает второй элемент входного списка

$head$ – функция $head : S^* \to S$ возвращает голову (первый элемент) входного списка

$tail$ – функция $tail : S^* \to S$ возвращает хвост (все кроме первого элемента) входного списка

$cons$ – функция $cons : S^* \times S \to S^*$ добавляет во входной список новый элемент

$prev$ – функция $prev : V \to P$ возвращает точку программы перед узлом графа потока управления

$post$ – функция $post : V \to P$ возвращает точку программы после узла графа потока управления

put – функция изменяет связанное с точкой программы состояние

get – функция возвращает связанное с точкой программы состояние

f_a – функция переноса соответствующая вершине a: $f_a = t(a) : V \to F$.

Алгоритм:

1. Инициализация:
   $W \leftarrow nil$
   `for` (v_1, v_2) `in` F `do`
       $W \leftarrow cons((v_1, v_2), W)$
   `for` v `in` V
       `if` $v \in I$ `then`
           $put(prev(v), i)$
           $put(post(v), \bot)$
       `else`
           $put(prev(v), \bot)$
           $put(post(v), \bot)$
2. Пополнение информации:
   `while` $W \neq nil$ `do`

$a \leftarrow fst(head(W))$
$b \leftarrow snd(head(W))$
$W \leftarrow tail(W)$
$l = f_a(get(prev(a)))$
$put(post(a), l)$
`if` $(get(pre(b)) \prec l)$ `then`
$put(pre(b), get(pre(b)) \sqcup l)$
`for` (b, c) `in` F `do`
$W \leftarrow cons((b, c), W)$

3. Сохранение результата:
`for` v `in` V `do`
$MFP_{\circ}(v) \leftarrow get(pre(v))$
$MFP_{\bullet}(v) \leftarrow get(post(v))$

Свойства алгоритма:

- Завершаемость
На этапах 1 и 3 алгоритма итерация производится над конечными множествами. В начале второго этапа в W находится конечное число элементов, и в каждой итерации цикла либо длина списка W уменьшается на 1 либо происходит уточнение информации и увеличение длины списка на конечное число элементов. Из-за наличия у решётки верхней грани, информация может уточняться только конечное число раз, поэтому второй этап завершим.

- Корректность
На первом этапе происходит инициализация начальными значениями. На втором этапе происходит вычисление неподвижной точки для каждой вершины $v \in V$ — значение $get(prev(v))$ после последнего уточнения и будет являться неподвижной точкой. На третьем этапе происходит сохранение результатов работы алгоритма.

Формальное доказательство завершаемости и корректности приведено в [6].

В данной работе используется модифицированный алгоритм с рабочим списком. В него внесены следующие изменения:

- Добавлено дополнительное предусловие на распространения информации между вершинами графа потока управления. Для того, чтобы информация распространилась по ребру (a, b), необходимо чтобы метка, которым оно помечено, присутствовала среди возможных меток. Множество возможных меток извлекается из состояния программы в точке после вершины a. Например, в множество меток можно включить метки возбуждения исключения, истинности и ложности условия в операциях ветвления.

- Добавлено условие завершения анализа. Алгоритм завершает свою работу при достижении вершины, достижимость которой нужно проверить. Проход через такую вершину означает наличие вычисления на модели приводящего в заданную точку программы.

- Результатом работы алгоритма не всегда является неподвижная точка. При попадании в проверяемую вершину работа алгоритма завершается. Состояния при этом содержат более точную информацию.

Классический	Модифицированный

`while` $W \neq nil$ `do`
$a \leftarrow fst(head(W))$
$b \leftarrow snd(head(W))$
$W \leftarrow tail(W)$
$l = f_a(get(prev(a)))$
$put(post(a), l)$
`if` $(get(pre(b)) \prec l)$ `then`
$put(pre(b), get(pre(b)) \sqcup l)$
`for` (b, c) `in` F `do`
$W \leftarrow cons((b, c), W)$
	`while` $W \neq nil$ `do`
	$a \leftarrow fst(head(W))$
	`if` $(a \in Z)$ `then break`
	$b \leftarrow snd(head(W))$
	$W \leftarrow tail(W)$
	$l = f_a(get(prev(a)))$
	$put(post(a), l)$
	`if` $(m(a, b) \in w(l) \wedge get(pre(b)) \prec l)$ `then`
	$put(pre(b), get(pre(b)) \sqcup l)$
	`for` (b, c) `in` F `do`
	`if` $((b, c) \notin W)$ `then`
	$W \leftarrow cons((b, c), W)$

Таблица 1: Различие между классическим и изменённым алгоритмами с рабочим списком

Для задания анализа с изменённым алгоритмом необходимо в дополнение к $AnalysisInstance$ описать еще 4 элемента:

$$ReachAnalysis = (L, F, E, I, i, t, M, m, w, Z)$$

- M – множество меток рёбер для задания условий прохождения потока по ребру
- m – отображение $m : V \times V \to M$, размечающее рёбра графа потока
- w – отображение $w : L \to 2^M$, извлекающее из рёшетки информацию по рёбрам с какими метками пойдет поток управления
- Z – множество вершин, достижимость которых необходимо проверить

Модификация затронула только 2 этап алгоритма. Различия между классическим и изменённым алгоритмами приведены в таблице 1.

Свойства изменённого алгоритма:

- Завершаемость
  Как и в классическом варианте, на каждой итерации на втором этапе длина рабочего списка уменьшается на 1. Однако для уточнения информации и пополнения рабочего списка должно быть выполнено дополнительное предусловие. Вычисление этого предусловия не влияет на завершаемость алгоритма. Поскольку классический алгоритм завершим и дополнительное условие не увеличивает число добавляемых в рабочий список рёбер, изменённый алгоритм так же завершим.
- Корректность
  Будет ли результатом работы этого алгоритма неподвижная точка функций? Кажется нет, но это и не нужно. Если во время работы алгоритма встречена точка, достижимость которой нужно проверить, то нужно тут же остановиться. При этом не будет получена неподвижная точка, но будет получена более точная информация о том, при каких значениях достигается эта точка.

5.2 Инструкции графа потока управления

Для построения абстрактного синтаксического дерева T с множеством вершин K_1 используется программа Rhino [11]. В процессе обхода дерева T строится граф потока управления $G(V, E)$ из множества вершин K_2, описанных в таблице 2. Аргументами операций, соответствующих вершинам из этого множества являются программные и временные переменные.

Последние введены для сокращения множества вершин и упрощения семантики. Более подробно они будут описаны в главе 5.6.

Таблица 2: Вершины графа потока управления

SKIP	Пустая команда. Никак не влияет на ход выполнения.
PSEUDO_ROOT	Псевдо-вершина представляющая входную точку программы.
PSEUDO_EXIT	Псевдо-вершина представляющая точку выхода.
DECLARE_VARIABLE(x)	Объявление программной переменной с именем x.
READ_VARIABLE(x, v)	Чтение значения программной переменной с именем x во временную переменную с именем v.
WRITE_VARIABLE(v, x)	Запись значения временной переменной с именем v в программную переменную с именем v.
CONSTANT(c, v)	Запись константы c во временную переменную v.
READ_PROPERTY(v_1, v_2, v_3)	Чтение атрибута с именем содержащимся во временной переменной v_2 из объекта находящегося в v_1 в переменную v_3.
WRITE_PROPERTY(v_1, v_2, v_3)	Запись в атрибут с именем содержащимся во временной переменной v_2 объекта находящегося в v_1 переменной v_3.
DELETE_PROPERTY(v_1, v_2, v_3)	Удаление атрибута с именем содержащимся во временной переменной v_2 из объекта находящегося в v_1. Результат операции записывается в переменную v_3.
IF(v)	Операция ветвления.
ENTRY(f, x_1, ... x_n)	Используется как уникальная метка входа в функцию с необязательным именем f и параметрами x_1, ..., x_n.
EXIT	Используется как уникальная метка выхода из функции.
EXIT_EXC	Используется как уникальная метка выхода из функции в результате исключения.
CALL(w, v_0, ... v_n)	Вызов функции с именем w и аргументами v_0, ... v_n. Нулевой аргумент содержит значение this.
AFTER_CALL(v)	Используется как точка возврата из вызова функции или конструктора. В v записывается результат вызова или созданный объект.
CONSTRUCT(w, v_0, ... v_n)	Инструкция создания объекта с именем w, инициализированного аргументами v_0, ... v_n. В v_0 содержится значение this.
RETURN(v)	Инструкция возврата из функции значения, содержащегося во временной переменной v.
THROW(v)	Возбуждение исключения. Временная переменная v содержит передаваемый с исключением объект.
CATCH(x)	Блок обработки исключений. Программная переменная с именем x содержит переданный с исключением объект.
FOR_IN(v_1, v_2)	Цикл поочередно записывающий во временную переменную v_1 атрибуты объекта, находящегося в переменной v_2.
WITH(v)	Инструкция изменения области видимости. На вершину стековых фреймов помещается объект, лежащий во временной переменной v.

Таблица 2: Вершины графа потока управления *(продолжение)*

`AFTER_WITH`	Инструкция изменения области видимости. Снимает объект с вершины стековых фреймов.
`UNOP(`v_1`, `v_2`)`	Унарные операции `NEG`, `POS`, `BITNOT`, `NOT`, `INSTANCEOF`, `TYPEOF`, `INC`, `DEC`. Временная переменная v_1 содержит операнд, а в v_2 записывается результат.
`BINOP(`v_1`, `v_2`, `v_3`)`	Бинарные операции `BITOR`, `BITXOR`, `BITAND`, `AND`, `OR`, `LSH`, `RSH`, `URSH`, `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `EQ`, `NE`, `SHNE`, `SHEQ`, `LT`, `LE`, `GT`, `GE`, `IN`. Временные переменные v_1 и v_2 содержат операнды, а в v_3 записывается результат.
`HOOK(`v_1`, `v_2`, `v_3`, `v_4`)`	Тернарная операция.

Для задания направления распространения потока управления рёбра размечены элементами множества меток `M`. Это множество определено следующим образом:

$$
\begin{aligned}
\texttt{M} = \{ & \\
& true, && \text{— при истинности условия в операторе ветвления} \\
& false, && \text{— при ложности условия в операторе ветвления} \\
& except, && \text{— при возбуждении исключения} \\
& uncond && \text{— без ограничений} \\
\} &
\end{aligned}
$$

Разметка задается функцией $m : V \times V \to \texttt{M}$, которая неявно описывается при построении графа потока.

5.3 Абстракция данных

Двумя фундаментальными типами данных в языке JavaScript являются элементарный тип и объект. Из-за перечисленных ниже особенностей языка в модели эти два фундаментальных типа обобщены в один – абстрактный объект `AbsObject`.

- Явно заданные строковые литералы и числа при обращении к ним могут неявно приводятся к объектам соответствующего типа.
- Все объекты передаются по ссылке.
- Все переменные являются членами глобального объекта.

К элементарным типам относятся `Undefined`, `Null`, `Boolean`, `String`, `Number`. Тип `Undefined` имеет одно значение, которое присваивается любой неинициализированной переменной. Все числа представляются как числа с плавающей запятой, которые при необходимости (например при индексации массива) неявно преобразовываются к целым числам.

Обращение к ранее явно не инициализированному члену объекта не является ошибкой. Запрашиваемый член неявно создается и инициализируется значением типа `Undefined`. Поэтому, можно сказать что объект – это отображение всевозможных строк, то есть значений типа `String`, на набор атрибутов и значений, при этом свойства объекта явно не объявленные имеют значение по-умолчанию.

Абстрактный объект `AbsObject` напрямую следует этому утверждению. Он представляет собой отображение всевозможных строк на декартово произведение множества значений `Value`

и множества свойств `Properties`. Множество `Value` представляет собой декартово произведение множеств `Undef, Null, Bool, Number, String`, моделирующих соответствующие элементарные типы, и множества `Object` всех подмножеств множества ссылок `Label` на абстрактные объекты.

$$\texttt{Value} = \texttt{Undef} \times \texttt{Null} \times \texttt{Bool} \times \texttt{Number} \times \texttt{String} \times \texttt{Object}$$

Множество `Properties` есть декартово произведение множеств свойств, соответствующих свойствам объекта `Object`, описанных в главе 3.

$$\texttt{Properties} = \texttt{ReadOnly} \times \texttt{DoNotDelete} \times \texttt{DoNotEnum}$$

Функции в языке JavaScript заменяют последовательность стековых фреймов при вызове на ту, при которой они были объявлены. Чтобы смоделировать эту особенности в модель абстрактного объекта `AbsObject` добавлено множество возможных значений последовательности стековых фреймов `ScopeChain`, при которых этот объект создавался.

В языке поддерживается интроспекция — возможность менять структуру объектов во время выполнения, поэтому необходимо учитывать возможность удаления свойств объекта. Например:

```
var a = 10;
a.b = 20;

if (condition()) {
  delete a.b;
}

var result = a.b;
```

Если не учитывать удаление свойств, то переменная `result` может потерять значение `undefined`. Для этого в абстрактный объект добавлено множество `Absent`.

Абстрактный объект может моделировать несколько конкретных объектов. Это необходимо для завершаемости анализа, например, в следующем цикле:

```
while (true) {
  a = new Object()
}
```

Пусть при входе в цикл идентификатор a указывает на объект со значением v_0. Если на каждой итерации создавать новый абстрактный объект с меткой l_k и добавлять ее к множеству возможных значений объекта v_0, то это приведёт к незавершаемости анализа.

$$v_0 = (\ ,\ ,\ ,\ ,\ ,\{l_0\})$$
$$v_k = (\ ,\ ,\ ,\ ,\ ,\{l_k\}) \sqcup (\ ,\ ,\ ,\ ,\ ,\{l_0, \cdots, l_{k-1}\}) = (\ ,\ ,\ ,\ ,\ ,\{l_0, \cdots, l_k\})$$

Поэтому, необходимо определить семантику операции `NEW` таким образом, чтобы объект создавался при первом вызове, а при последующих возвращалась ссылка на уже существующий объект.

Схожая проблема возникает при моделировании замыканий. При объявлении функции создается объект, который связывается в объекте области видимости с идентификатором. Создание объекта-функции происходит каждый раз когда встречается инструкция объявления. Рассмотрим, например, следующий код:

```
function f() {
  return function g() {}
}

while (true) {
  a = f()
}
```

Если при каждом вызове внешней функции `f()` создавать новый объект, представляющий функцию `g()`, то так же нарушится конечность множества `Label`. Необходимо чтобы каждой функции соответствовал один объект. Однако, изменить семантику операции `ENTRY` будет недостаточно — корректная модель функции должна содержать последовательность стековых фреймов, при которых она была создана. Поэтому `AbsObject` включает множество последовательностей стековых фреймов, при которых этот объект был объявлен.

Таким образом, абстрактный объект состоит из совокупности множества последовательностей стековых фреймов, в которых этот объект мог быть создан и отображения всевозможных строк `Names` на значение `Value` и атрибуты значения `Properties, Absent`:

$$\texttt{AbsObject} = (\texttt{Names} \hookrightarrow \texttt{Value} \times \texttt{Absent} \times \texttt{Properties}) \times 2^{\texttt{ScopeChain}}$$

Множества `Absent, Undef, Null, Bool, ReadOnly, DoNotDelete, DoNotEnum` являются решётками. Частичный порядок на них определяется соответствующими диаграммами.

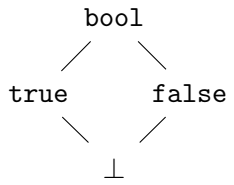

Рис. 1: Решётка `Bool`

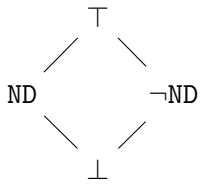

Рис. 2: Решётка `DoNotDelete`

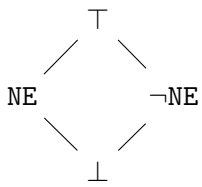

Рис. 3: Решётка `DoNotEnum`

Множество `String` состоит из множества всевозможных подмножеств строк длины не более $\mathcal{K}$. В качестве операции частичного порядка используется включение множеств. Пример решётки на рисунке 8.

$$\texttt{String} = 2^{\texttt{string}(\mathcal{K})}$$
$$\forall P, Q \in \texttt{String}:$$
$$P \prec Q \Leftrightarrow \forall p \in P,\ \exists q \in Q\ :\ p = q$$
$$P \sqcup Q = \{o\ :\ o \in P \vee o \in Q\}$$

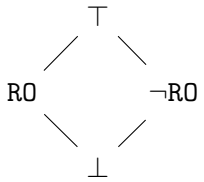

Рис. 4: Решётка `ReadOnly`

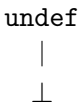

Рис. 5: Решётка `Undef`

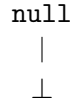

Рис. 6: Решётка `Null`

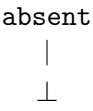

Рис. 7: Решётка `Absent`

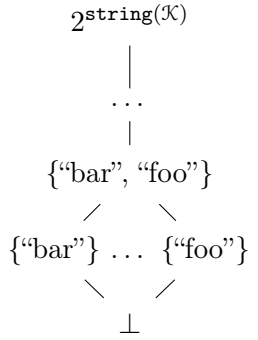

Рис. 8: Часть решётки `String`

Множество `Object` состоит из множества всевозможных подмножеств множества меток `Label`. В качестве операции частичного порядка используется включение множеств. Пример решётки на рисунке 9.

$$
\begin{aligned}
&\texttt{Object} = 2^{\texttt{Label}} \\
&\forall P, Q \in \texttt{Object}: \\
&\qquad P \prec Q \Leftrightarrow \forall p \in P,\ \exists q \in Q\ :\ p = q \\
&\qquad P \sqcup Q = \{o\ :\ o \in P \vee o \in Q\}
\end{aligned}
$$

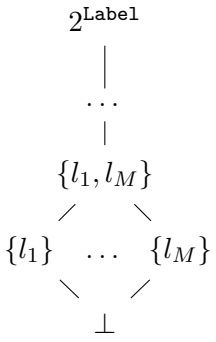

Рис. 9: Часть решётки `Object`

Множество `Number` состоит из всевозможных отрезков от `Min` до `Max`. В качестве операции частичного порядка используется включение одного отрезка в другой. Пример решётки на рисунке 10.

$$
\begin{aligned}
&\texttt{Number} = \{[a, b] : a \leq b, a, b \in [Min, Max]\} \\
&\forall A, B \in \texttt{Number},\ A = [a_1, a_2], B = [b_1, b_2]: \\
&A \prec B \Leftrightarrow a_1 < b_1, a_2 > b_2 \\
&A \sqcup B = \begin{cases} \bot & [a_1, a_2] \cap [b_1, b_2] = \emptyset \\ [a, b] & a = min(a_1, b_1), b = max(a_2, b_2) \end{cases}
\end{aligned}
$$

Для множества `Value` операции частичного порядка и объединения двух элементов определены следующим образом:

$$
\begin{aligned}
&\forall A, B \in \texttt{Value},\ A = (u_a, n_a, b_a, d_a, s_a, l_a),\ B = (u_b, n_b, b_b, d_b, s_b, l_b)\ : \\
&A \prec B \Leftrightarrow u_a \prec u_b \wedge n_a \prec n_b \wedge b_a \prec b_b \wedge d_a \prec d_b \wedge s_a \prec s_b \wedge l_a \prec l_b \\
&A \sqcup B = (u_a \sqcup u_b, n_a \sqcup n_b, b_a \sqcup b_b, d_a \sqcup d_b, s_a \sqcup s_b, l_a \sqcup l_b)
\end{aligned}
$$

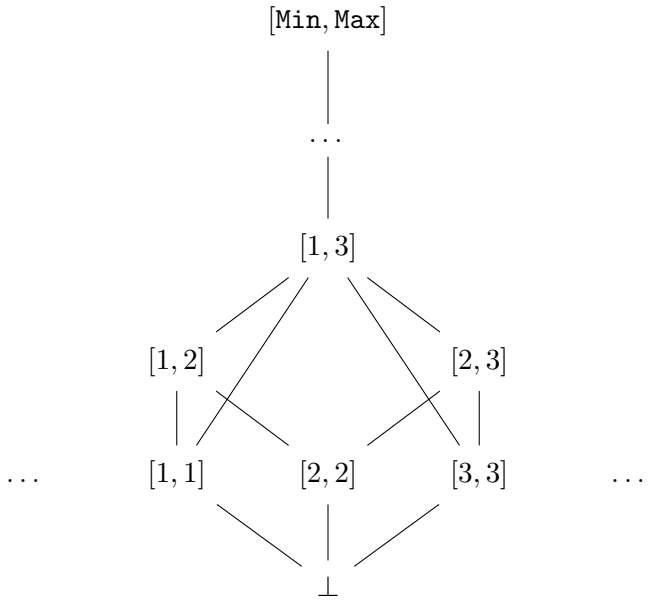

Рис. 10: Часть решётки `Number`

Для множества `AbsObject` отношение частичного порядка и объединения двух элементов определены следующим образом:

$$\forall A, B \in \texttt{AbsObject}\,, \forall p \in \texttt{Names}\,,$$
$$A = \left(\left(\texttt{Names} \overset{f}{\hookrightarrow} \texttt{Value} \times \texttt{Absent} \times \texttt{Properties}\right), SC_a\right)\,,$$
$$B = \left(\left(\texttt{Names} \overset{g}{\hookrightarrow} \texttt{Value} \times \texttt{Absent} \times \texttt{Properties}\right), SC_b\right)\,,$$
$$f(p) = (v_a, a_a, p_a)\,, g(p) = (v_b, a_b, p_b)\ :$$
$$A \prec B \Leftrightarrow \begin{cases} v_a \prec v_b \wedge a_a \prec a_b \wedge p_a \prec p_b \\ SC_a \subseteq SC_b \end{cases}$$
$$A \sqcup B = \left(\left(Names \overset{k}{\hookrightarrow} \texttt{Value} \times \texttt{Absent} \times \texttt{Properties}\right), SC\right)\,, \quad k(p) = (v_a \sqcup v_b, a_a \sqcup a_b, p_a \sqcup p_b)\,,$$
$$SC = SC_a \cup SC_b$$

5.4 Модель интерпретатора

Состояние абстрактного интерпретатора `State` будет описываться через состояние абстрактной кучи `Store` и состояние абстрактного стека выполнения `Stack`.

$$\texttt{State} = \texttt{Store} \times \texttt{Stack}$$

Абстрактная куча представляет собой отображение множества меток `Label` на абстрактные объекты `AbsObject`. Объекты добавляются в кучу операциями `CONSTRUCT, DECLARE_VARIABLE, ENTRY, CALL`.

$$\texttt{Store} = \texttt{Label} \hookrightarrow \texttt{AbsObject}$$

Для упрощения семантики были введены временные переменные, которые являются аналогом регистров. Явных ограничений на их количество нет, однако мощность множества всех временных переменных `Temp` ограничена сверху числом вершин в графе управления: $|\texttt{Temp}| = O(|G|)$. Множество временных переменных определяется в процессе построения графа потока и не изменяется в ходе интерпретации. Значение временной переменной может изменяться операция-

ми READ_VARIABLE, CONSTANT и READ_PROPERTY. Абстрактный стек выполнения состоит множества возможных контекстов выполнения ExecContext и отображения временных переменных Temp на их значения Value:

$$\texttt{Stack} = (\texttt{Temp} \to \texttt{Value}) \times 2^{\texttt{ExecContext}}$$

Контекст исполнения ExecContext состоит из последовательности стековых фреймов ScopeChain, представляющей иерархию *объектов области видимости*, метки, указывающей на текущий объект this и метки, указывающей на текущий *объект переменных*.

$$\texttt{ExecContext} = \texttt{ScopeChain} \times \texttt{Label} \times \texttt{Label}$$
$$\texttt{ScopeChain} = \texttt{Label}*$$

В абстрактном стеке учитывается множество возможных контекстов, а не один контекст из-за того, что абстракный объект-функция может моделировать несколько конкретных объектов-функций, созданных при разных конктекстах.

На множестве Store операции частичного порядка и объединения двух элементов объединены следующим образом:

$$\begin{aligned}
&\forall A, B \in \texttt{Store},\ A = \left(\texttt{Label} \overset{f}{\hookrightarrow} \texttt{AbsObject}\right),\ B = \left(\texttt{Label} \overset{g}{\hookrightarrow} \texttt{AbsObject}\right), \\
&\qquad f : \texttt{Label} \supset L_f \to \texttt{AbsObject}, \\
&\qquad g : \texttt{Label} \supset L_g \to \texttt{AbsObject} : \\
&A \prec B \Leftrightarrow L_f \subset L_g \wedge \forall l \in L_f\ :\ f(l) \prec g(l) \\
&A \sqcup B = \left(\texttt{Label} \overset{k}{\hookrightarrow} \texttt{AbsObject}\right) \\
&\qquad k(l) = \begin{cases} f(l) & l \in L_f \wedge l \notin L_g \\ g(l) & l \in L_g \wedge l \notin L_f \\ f(l) \sqcup g(l) & l \in L_f \cap L_g \end{cases}
\end{aligned}$$

На множестве Stack операции частичного порядка и объединения двух элементов объединены следующим образом:

$$\begin{aligned}
&\forall A, B \in \texttt{Stack},\ A = \left(\left(\texttt{Temp} \overset{f}{\to} \texttt{Value}\right) \times EC_a\right),\ B = \left(\left(\texttt{Temp} \overset{g}{\to} \texttt{Value}\right) \times EC_b\right) : \\
&A \prec B \Leftrightarrow \forall t \in \texttt{Temp} : f(t) \prec g(t) \\
&A \sqcup B = \left(\left(\texttt{Temp} \overset{k}{\to} \texttt{Value}\right) \times EC_a\right) \\
&\qquad k(t) = f(t) \sqcup g(t)
\end{aligned}$$

Для анализа необходимо чтобы в решётке содержалась информация о метках рёбер, по которым возможно распространение информации после очередной операции. Для этого в решётку анализа включено множество Marker. Оно состоит из всевозможных подмножеств множества меток M, описанного в главе 5.2. Число меток ограничено M, поэтому Marker конечно. В качестве операции частичного порядка используется включение множеств. Часть решётки изображена на рисунке 11.

$$\begin{aligned}
&\texttt{Marker} = 2^{\texttt{M}} \\
&\forall P, Q \in \texttt{Marker} : \\
&\qquad P \prec Q \Leftrightarrow \forall p \in P,\ \exists q \in Q\ :\ p = q \\
&\qquad P \sqcup Q = \{o\ :\ o \in P \vee o \in Q\}
\end{aligned}$$

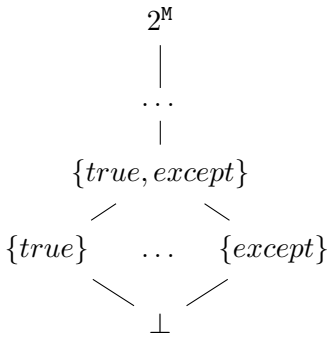

Рис. 11: Часть решётки `Marker`

Для моделирования поведения операций создания объектов `CONSTRUCT` и объявления функций `ENTRY` вводится функция `FunMap`, отображающая уникальный идентификатор вершины, на соответствующий объект, если он уже был создан, либо на l_{null}, если не был.

$$\mathtt{FunMap} = V \times \mathtt{Label}$$

Таким образом, решётка анализа $\mathcal{L}$ содержит информацию о состоянии абстрактного интерпретатора, множестве меток, по которым разрешено распространение информации и отображения идентификаторов порождающих вершин на указатели на абстрактные объекты.

$$\mathcal{L} = \mathtt{State} \times \mathtt{Marker} \times \mathtt{FunMap}$$

5.5 Интерпроцедурный анализ

При интерпретации программы контекст функции определяется последовательностью стековых фреймов в момент вызова. Попытка полностью смоделировать эту последовательность при абстрактной интерпретации приведет к незавершаемости анализа. Это произойдет из-за наличия рекурсии. Пример части графа потока управления, содержащий рекурсивный вызов изображен на рисунке 12. При каждом вызове к вершине иерархии объектов областей види-

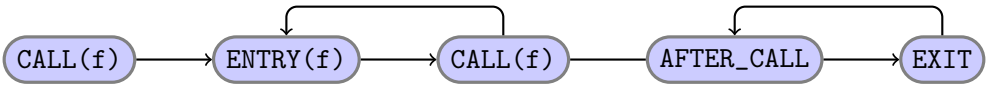

Рис. 12: Рекурсивный вызов функции `f`

мости будет (`ScopeChain` в терминах решётки) будет добавляться новый объект активации. Это приводит к неограниченной решётке, то есть к невозможности применения изменённого алгоритма с рабочим списком. Чтобы этого избежать, обычно [6] вводится ограничение на отслеживаемую глубину последовательности стековых фреймов, или, в терминах `JavaScript`, иерархии объектов областей видимости. Однако, в данном случае этот подход неприменим. Если ввести такое ограничение, то, начиная с некоторой глубины, из иерархии областей видимости исчезнет глобальный объект. Это приведет к невозможности узнать значение переменных, идентификаторы которых хранились в этом глобальном объекте, то есть, к нарушению свойства корректности анализа.

Поэтому, в этой работе, введено ограничение на максимальную длину иерархии объектов областей видимости:

$$|\mathtt{ScopeChain}| \leq \mathcal{S}$$

Поскольку иерархия областей видимости может изменяться только инструкциями `catch` и `with`, бесконечный цикл на рисунке 12 не повлияет на завершимость анализа.

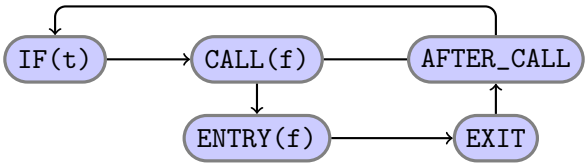

Рис. 13: Вызов функции `f` внутри цикла

Для возврата данных из функции, то есть для передачи данных между инструкциями `RETURN` и `AFTER_CALL`, используется специальная временная переменна v_{call}.

Для инициализации программной переменной объявленной в блоке `CATCH` значением, брошенным инструкцией `THROW`, используется временная переменная v_{ex}.

5.6 Функция переноса

В этой главе будет описано влияние каждой операции на состояние интерпретатора. Каждой вершине v графа потока управления $G(V, E)$ функциями $post : V \to P$ и $pre : V \to P$ ставится в соответствие две точки. Этим точкам функцией $get : P \to Lattice$ ставится в соответствие состояние интерпретатора. До прохода через вершину v он находится в состоянии s:

$s = (h, t, e) = get\,(pre\,(v))$	
$h : Label \to AbsObject$	состояние кучи
$t : T \to Value$	значение временных переменных
$e = \{(chain_i, var_i, this_i)\}$	контексты выполнения, $i \in [1, W]$
$\tilde{m} : T \to Value$	возможные метки рёбер

После прохода через вершину v интерпретатор перейдет в состояние $\tilde{s}$, определяемое функцией переноса $f_v : \mathcal{L} \to \mathcal{L}$:

$\tilde{s} = (\tilde{h}, \tilde{t}, \tilde{e}, m) = get\,(post\,(v))$	
$\tilde{s} = f_v\,(s)$	
$\tilde{h} : Label \to AbsObject$	состояние кучи
$\tilde{t} : T \to Value$	значение временных переменных
$\tilde{e} = \{(chain_i, var_i, this_i)\}$	контексты выполнения, $i \in [1, \tilde{W}]$
$\tilde{m} : T \to Value$	возможные метки рёбер

Ниже определены функции переноса f_v для каждой из возможных вершин в графе потока управления. Так же неявно описана функция $\tau : V \to \mathcal{F}$, ставящяя вершине в соответствие функцию.

`DECLARE_VARIABLE(x)`

$$
\begin{aligned}
(l, o) &= newObject(e) \\
env_i &= put\left(h\left(var_i\right), x, l\right) \\
\tilde{h}\left(id\right) &= \begin{cases} h(id) & id \neq l, id \neq var_i \\ o & id = l \\ env_i & id = var_i \end{cases} \\
\tilde{t} &= t \\
\tilde{e} &= e \\
\tilde{m} &= \{uncond\}
\end{aligned}
$$

`READ_VARIABLE(x, v)`

$$
\begin{aligned}
l_i &= resolve(h, chain_i, x) \\
res_i &= \begin{cases} \texttt{undef} & l = l_{null} \\ getProp(h(l_i), x) & l \neq l_{null} \end{cases} \\
res &= \bigsqcup res_i \\
\tilde{h} &= h \\
\tilde{t}\left(id\right) &= \begin{cases} t\left(id\right) & id \neq v \\ res & id = v \end{cases} \\
\tilde{e} &= e \\
\tilde{m} &= \{uncond\}
\end{aligned}
$$

`WRITE_VARIABLE(v, x)`

$$
\begin{aligned}
chain_i &= l_1^i, \cdots, l_k^i \\
h(l_k^i) &= (names, chains) \\
pr &= (v, \bot, (\neg\texttt{RO}, \neg\texttt{ND}, \neg\texttt{NE})) \\
\tilde{names}(id) &= \begin{cases} names(id) & id \neq l_k^i \\ names(id) & id = l_k^i \wedge (\bot, \bot, (\texttt{RO}, \bot, \bot)) \prec names(id) \\ names(id) \sqcup pr & id = l_k^i \wedge (\bot, \bot, (\top, \bot, \bot)) \prec names(id) \\ pr & id = l_k^i \wedge (\bot, \bot, (\neg\texttt{RO}, \bot, \bot)) \prec names(id) \\ pr & id = l_k^i \wedge \neg hasProp(h(l_k^i), x) \end{cases} \\
\tilde{h}(id) &= \begin{cases} h(id) & id \neq this \\ (\tilde{names}, chains) & id = this \end{cases} \\
\tilde{t} &= t \\
\tilde{e} &= e = \{(chain_i, this_i, var_i)\} \\
\tilde{m} &= \{uncond\}
\end{aligned}
$$

`CONSTANT(c, v)`

$$\tilde{h} = h$$
$$\tilde{t}(id) = \begin{cases} t(id) & id \neq v \\ c & id = v \end{cases}$$
$$\tilde{e} = e$$
$$\tilde{m} = \{uncond\}$$

`READ_PROPERTY(v1, v2, v3)`

$$v_1 = (u_1, n_1, b_1, a, d_1, \{l_1, \cdots, l_k\})$$
$$v_2 = (u_2, n_2, b_2, \{s_1, \cdots, s_p\}, d_2, o_2)$$
$$res = \bigsqcup_{\substack{i \in [1,k] \\ j \in [1,p]}} res(l_i, s_j)$$
$$res(l, s) = \begin{cases} getProp(h(l), s) & hasProp(h(l), s) \\ res(getProp(h(l), \text{"}\texttt{prototype}\text{"}), s) & getProp(h(l), \text{"}\texttt{prototype}\text{"}) \neq l_{null} \\ \texttt{undefined} & getProp(h(l), \text{"}\texttt{prototype}\text{"}) = l_{null} \end{cases}$$
$$\tilde{h} = h$$
$$\tilde{t}\,(id) = \begin{cases} t\,(id) & id \neq v_3 \\ res & id = v_3 \end{cases}$$
$$\tilde{e} = e$$
$$\tilde{m} = \{\texttt{uncond}\}$$

`WRITE_PROPERTY(v1, v2, v3)`

$$v_1 = (u_1, n_1, b_1, a, d_1, \{l_1, \cdots, l_k\})$$
$$v_2 = (u_2, n_2, b_2, \{s_1, \cdots, s_q\}, d_2, o_2)$$
$$h(p) = (names_p, chains_p)$$
$$v = (v_3, \bot, (\neg\texttt{RO}, \neg\texttt{ND}, \neg\texttt{NE}))$$
$$\tilde{names}_p(id) = \begin{cases} names_p(id) & id \notin \{s_1, \cdots, s_q\} \\ names_p(id) \sqcup v & (\bot, \bot, (\top, \bot, \bot)) \prec names_p(id) \\ names_p(id) & (\bot, \bot, (\texttt{RO}, \bot, \bot)) \prec names_p(id) \\ v & (\bot, \bot, (\neg\texttt{RO}, \bot, \bot)) \prec names_p(id) \\ v & \neg hasProp(h(p), id) \end{cases}$$
$$\tilde{h}(id) = \begin{cases} h(id) & id \notin \{l_1, \cdots, l_k\} \\ (\tilde{names}_p, chains_p) & id = l_p, p \in [1, l] \end{cases}$$
$$\tilde{t} = t$$
$$\tilde{e} = e = (chain, this, var)$$
$$\tilde{m} = \{uncond\}$$

`DELETE_PROPERTY(v1, v2, v3)`

$$
\begin{aligned}
v_1 &= (u_1, n_1, b_1, a, d_1, \{l_1, \cdots, l_k\}) \\
v_2 &= (u_2, n_2, b_2, \{s_1, \cdots, s_p\}, d_2, o_2) \\
res &= \bigsqcup_{\substack{l \in [1,k] \\ n \in [1,p]}} res(l, n) \\
res(l, n) &= \begin{cases} \texttt{true} & \neg hasProp(h(l), n) \vee canDel(h(l), n) \\ \texttt{false} & \neg canDel(h(l), n) \end{cases} \\
mod &= (\bot, \top, (\bot, \bot, \bot)) \\
\tilde{names}_p(id) &= \begin{cases} names_p(id) & id \notin \{s_1, \cdots, s_p\} \\ names_p(id) \sqcup mod & id \in \{s_1, \cdots, s_p\} \wedge hasProp(h(p), id) \end{cases} \\
\tilde{h}(id) &= \begin{cases} h(id) & id \notin \{l_1, \cdots, l_k\} \\ (\tilde{names}_p, chains_p) & id = l_q, q \in [1, k] \end{cases} \\
\tilde{t}\,(id) &= \begin{cases} t\,(id) & id \neq v_3 \\ res & id = v_3 \end{cases} \\
\tilde{e} &= e = (chain, this, var) \\
\tilde{m} &= \{uncond\}
\end{aligned}
$$

`IF(v)`

$$
\begin{aligned}
\tilde{h} &= h \\
\tilde{t} &= t \\
\tilde{e} &= e \\
\tilde{m} &= \begin{cases} \{true\} & \text{toBool(v)} = \texttt{true} \\ \{false\} & \text{toBool(v)} = \texttt{false} \\ \{true, false\} & \text{toBool(v)} = \texttt{bool} \end{cases}
\end{aligned}
$$

`ENTRY(f, x1, ... xn)`

Используется как уникальная метка входа в функцию с необязательным именем `f` и параметрами `x1, ..., xn`.

`CALL(w, v0, ... vn)`

Вызов функции с именем `w` и аргументами `v0, ... vn`. Нулевой аргумент содержит значение `this`.

`CONSTRUCT(w, v0, ... vn)`

Инструкция создания объекта с именем `w`, инициализированного аргументами `v0, ... vn`. В `v0` содержится значение `this`.

`EXIT`

Используется как уникальная метка выхода из функции.

`EXIT_EXC`

Используется как уникальная метка выхода из функции в результате исключения.

`AFTER_CALL(v)`

$$\begin{aligned}\tilde{h} &= h \\ \tilde{t}(id) &= \begin{cases} t(id) & id \neq v_{call} \\ v & id = v_{call} \end{cases} \\ \tilde{e} &= e \\ \tilde{m} &= \{uncond\}\end{aligned}$$

`RETURN(v)`

$$\begin{aligned}\tilde{h} &= h \\ \tilde{t}(id) &= \begin{cases} t(id) & id \neq v_{call} \\ v & id = v_{call} \end{cases} \\ \tilde{e} &= e \\ \tilde{m} &= \{uncond\}\end{aligned}$$

`CATCH(x)`

Блок обработки исключений. Программная переменная с именем `x` содержит переданный с исключением объект.

`THROW(v)`

$$\begin{aligned}\tilde{h} &= h \\ \tilde{t}(id) &= \begin{cases} t(id) & id \neq v_{ex} \\ c & id = v_{ex} \end{cases} \\ \tilde{e} &= e \\ \tilde{m} &= \{except\}\end{aligned}$$

`FOR_IN(v1, v2)`

$$\begin{aligned}\tilde{h} &= h \\ v_2 &= (u_2, n_2, b_2, a, d_2, \{l_1, \cdots, l_k\}) \\ res &= \bigsqcup_{l \in [1,k]} res(h(l)) \\ m &= (\bot, \bot, (\bot, \bot, \neg\texttt{NE})) \\ res((names, chains)) &= \bigsqcup_{s\,:\,m \prec names(s)} val(names(s)) \\ val((v, a, (r, d, e))) &= v \\ \tilde{t}(id) &= \begin{cases} t(id) & id \neq v_1 \\ v & id = v_1 \end{cases} \\ \tilde{e} &= e \\ \tilde{m} &= \{uncond\}\end{aligned}$$

`WITH(v)`

Инструкция изменения области видимости.

`AFTER_WITH`

`UNOP(v1, v2)`

Унарные операции `NEG, POS, BITNOT, NOT, INSTANCEOF, TYPEOF, INC, DEC`. Временная переменная v_1 содержит операнд, а в v_2 записывается результат.

`BINOP(v1, v2, v3)`

Бинарные операции `BITOR, BITXOR, BITAND, AND, OR, LSH, RSH, URSH, ADD, SUB, MUL, DIV, MOD, EQ, NE, SHNE, SHEQ, LT, LE, GT, GE, IN`. Временные переменные v_1 и v_2 содержат операнды, а в v_3 записывается результат.

`ADD(v1, v2, v3)`

$$
\begin{aligned}
res_i &= \begin{cases} \texttt{undef} & l = l_{null} \\ getProp(h(l_i), x) & l \neq l_{null} \end{cases} \\
(\bot, \bot, \bot, [a_1, b_1], \bot, \bot) &= toNumber(t(v_1)) \\
(\bot, \bot, \bot, [a_2, b_2], \bot, \bot) &= toNumber(t(v_2)) \\
res &= (\bot, \bot, \bot, [a_1 + a_2, b_1 + b_2], \bot, \bot) \\
\tilde{h} &= h \\
\tilde{t}\,(id) &= \begin{cases} t\,(id) & id \neq v_3 \\ res & id = v_3 \end{cases} \\
\tilde{e} &= e \\
\tilde{m} &= \{uncond\}
\end{aligned}
$$

`HOOK(v1, v2, v3, v4)`

Тернарная операция.

Ниже определены вспомогательные функции, использованные при определении функций переноса.

$newobject : ScopeChain \to Label \times AbsObject$ Создание нового объекта

$toBool : Value \to Bool$

$$toBool((u, n, b, d, s, o)) = u_b \sqcup n_b \sqcup b \sqcup d_b \sqcup s_b \sqcup o_b$$

$$u_1 = \begin{cases} \bot & u = \bot \\ \texttt{false} & u \neq \bot \end{cases}$$

$$n_1 = \begin{cases} \bot & n = \bot \\ \texttt{false} & n \neq \bot \end{cases}$$

$$d_1 = \begin{cases} \texttt{true} & [0,0] \sqcup d \neq [0,0] \\ \texttt{false} & [0,0] \sqcup d = [0,0] \end{cases}$$

$$s_1 = \begin{cases} \texttt{true} & \{\text{"}\} \notin s \\ \texttt{false} & \{\text{"}\} \in s \end{cases}$$

$$o_1 = \begin{cases} \bot & o = \bot \\ \texttt{true} & o \neq \bot \end{cases}$$

$resolve : (Label \to AbsObject) \times Label^* \times String \to Label$

$$resolve(h, l_1, \cdots, l_k, x) = \begin{cases} l_k & hasProp(l_k, x) \\ resolve(h, l_1, \cdots, l_{k-1}, x) & \neg hasProp(l_k, x) \wedge k > 1 \\ l_{null} & \neg hasProp(l_k, x) \wedge k = 1 \end{cases}$$

$getProp : AbsObject \times String \to Value$

$$getProp((names, chains), x) = val_x$$
$$names(id) = (val_{id}, abs_{id}, prop_{id})$$

$hasProp : AbsObject \times String \to Value$

$$hasProp((names, chains), x) = val_x$$
$$names(id) = (val_{id}, abs_{id}, prop_{id})$$

$canDel : AbsObject \times String \to Bool$

$$canDel(o, s) = \begin{cases} \texttt{true} & \neg hasProp(o, s) \vee (\bot, \bot, (\bot, \neg\texttt{ND}, \bot)) \prec getProp(o, s) \\ \texttt{false} & (\bot, \bot, (\bot, \texttt{ND}, \bot)) \prec getProp(o, s) \end{cases}$$

5.7 Начальное состояние

В начальном состоянии в абстрактной куче находятся стандартные объекты, все временные переменные в абстрактром стеке не инициализированны, а множество меток, которыми должны быть помечены ветви, содержит *uncond*.

У функции есть свойство length

$$
\begin{aligned}
&i \in \mathcal{L},\ i = (stack, state, marker)\\
&marker = \textit{'uncond'}\\
&\quad state\ :\ Label \supset S_0 \xrightarrow{h} AbsObject\\
&\quad stack\ :\ ((T \xrightarrow{t} Value), L_g, L_g), \quad \forall\, id \in T\ : t(id) = \bot
\end{aligned}
$$

Множество меток стандартных объектов S_0 состоит из l_{null}, L_g,

Глобальный объект: невозможно использовать в качестве конструктора с оператором new. невозможно вызвать как функцию.

$$
\begin{aligned}
h(L_g) &= (this, l_{null})\\
singleObj(label) &= (\bot, \bot, \bot, \bot, \bot, \{label\})\\
this(\text{``}\texttt{Object}\text{''}\ ) &= singleObj(L_o)\\
this(\text{``}\texttt{Function}\text{''}) &= singleObj(L_f)\\
this(\text{``}\texttt{Number}\text{''}\ ) &= singleObj(L_n)\\
this(\text{``}\texttt{String}\text{''}\ ) &= singleObj(L_s)\\
this(\text{``}\texttt{Boolean}\text{''}\ ) &= singleObj(L_b)\\
this(\text{``}\texttt{Date}\text{''}\ ) &= singleObj(L_d)\\
this(\text{``}\texttt{Array}\text{''}\ ) &= singleObj(L_a)\\
this(\text{``}\texttt{Math}\text{''}\ ) &= singleObj(L_m)\\
this(\text{``}\texttt{NaN}\text{''}\ ) &= unimplemented()\\
this(\text{``}\texttt{Infinity}\text{''}\ ) &= unimplemented()\\
this(\text{``}\texttt{eval(x)}\text{''}\ ) &= unimplemented()\\
this(\text{``}\texttt{parseInt(string, radix)}\text{''}\ ) &= unimplemented()\\
this(\text{``}\texttt{parseFloat(string)}\text{''}\ ) &= unimplemented()\\
this(\text{``}\texttt{isNaN(number)}\text{''}\ ) &= unimplemented()\\
this(\text{``}\texttt{isFinite(number)}\text{''}\ ) &= unimplemented()
\end{aligned}
$$

$$
\begin{aligned}
h(L_o) &= (obj,)\\
obj(\text{``}\texttt{prototype}\text{''}\ ) &= singleObj(L_{op})\{DontEnum, DontDelete, ReadOnly\}\\
obj(\text{``}\texttt{constructor}\text{''}\ ) &= singleObj(L_{oc})\\
obj(\text{``}\texttt{call}\text{''}\ ) &= \text{toObject, если не undefined или null, иначе создать новый объект такой же, как если бы был вызван конструктор}
\end{aligned}
$$

Когда конструктор Object вызывается (как функция) без аргументов или с единственным аргументом value, предпринимаются следующие шаги:

1. Если value опущено - переход на шаг 8.
2. Если тип value не равен Object - переход на шаг 5.
3. Если value - встроенный объект ECMAScript object, то не создавать нового объекта и просто вернуть value.
4. Если value является объектом среды, предпринимаемые шаги и возвращаемый объект зависят от конкретной реализации и, возможно, от объекта среды.
5. Если тип value равен String, вернуть ToObject(value).
6. Если тип value равен Boolean, вернуть ToObject(value).

7. Если тип value равен Number, вернуть ToObject(value).

8. (Аргумент value не был передан или его тип был Null или Undefined.) Создать новый встроенный объект ECMAScript. Свойство [[Prototype]] создаваемого объекта устанавливается в прототип Object. Свойство [[Class]] создаваемого объекта устанавливается в "Object". У создаваемого объекта нет свойства [[Value]]. Вернуть созданный встроенный объект.

$$
\begin{aligned}
h(L_{op}) &= (obj,) \\
obj(\text{``}\texttt{prototype}\text{''}\ ) &= singleObj(l_{null}) \\
obj(\text{``}\texttt{constructor}\text{''}\ ) &= createObj() \\
obj(\text{``}\texttt{toString}\text{''}\ ) &= \\
obj(\text{``}\texttt{toLocaleString}\text{''}\ ) &= \\
obj(\text{``}\texttt{valueOf}\text{''}\ ) &= \\
obj(\text{``}\texttt{hasOwnProperty}\text{''}\ ) &= \\
obj(\text{``}\texttt{isPrototypeOf}\text{''}\ ) &= \\
obj(\text{``}\texttt{propertyIsEnumerable}\text{''}\ ) &=
\end{aligned}
$$

$$
\begin{aligned}
h(L_{oc}) &= (obj,) \\
obj(\text{``}\texttt{prototype}\text{''}\ ) &= singleObj(L_{fp})
\end{aligned}
$$

$$
\begin{aligned}
h(L_f) &= (fun,) \\
fun(\text{``}\texttt{prototype}\text{''}\ ) &= singleObj(L_{fp})\{DontEnum, DontDelete, ReadOnly\} \\
fun(\text{``}\texttt{constructor}\text{''}\ ) &= unimplemented() \\
fun(\text{``}\texttt{call}\text{''}\ ) &= unimplemented()
\end{aligned}
$$

Методы call и constructor создают функцию с телом, определяемым во время выполнения. Эти инструкции не рассматриваются в анализе.

$$
\begin{aligned}
h(L_{fp}) &= (obj,) \\
obj(\text{``}\texttt{prototype}\text{''}\ ) &= singleObj(L_{fpp}) \\
obj(\text{``}\texttt{constructor}\text{''}\ ) &= unimplemented() \\
obj(\text{``}\texttt{toString}\text{''}\ ) &= \\
obj(\text{``}\texttt{apply}\text{''}\ ) &= \\
obj(\text{``}\texttt{call}\text{''}\ ) &=
\end{aligned}
$$

$$
\begin{aligned}
h(L_{fpp}) &= (obj,) \\
obj(\text{``}\texttt{prototype}\text{''}\ ) &= singleObj(L_{op}) \\
obj(\text{``}\texttt{call}\text{''}\ ) &= \text{фукция, всегда возвращающая undefined}
\end{aligned}
$$

У каждого экземпляра Function есть дополнительные атрибуты

- length — ожидаемое число аргументов
- prototype — инициализация внутреннего свойства prototype создаваемого объекта до того, как объект Function вызовется для него в качестве конструктора. Свойство обладает атрибутом DontDelete.

Свойства объекта Number:

$$
\begin{aligned}
h(L_n) &= (num,) \\
num(\text{``}\texttt{prototype}\text{''}\ ) &= singleObj(L_{np})\{DontEnum, DontDelete, ReadOnly\} \\
num(\text{``}\texttt{constructor}\text{''}\ ) &= singleObj(L_{nc})() \\
num(\text{``}\texttt{call}\text{''}\ ) &= toNumber()
\end{aligned}
$$

$$
\begin{aligned}
h(L_{np}) &= (prt,) \\
prt(\text{``}\texttt{prototype}\text{''}\ ) &= singleObj(L_{op}) \\
prt(\text{``}\texttt{constructor}\text{''}\ ) &= \text{встроенный конструктор} \\
prt(\text{``}\texttt{[[Value]]}\text{''}\ ) &= 0 \\
prt(\text{``}\texttt{toString(radix)}\text{''}\ ) &= \\
prt(\text{``}\texttt{toLocaleString()}\text{''}\ ) &= \\
prt(\text{``}\texttt{valueOf()}\text{''}\ ) &= \\
prt(\text{``}\texttt{toFixed(fractionDigits)}\text{''}\ ) &= \\
prt(\text{``}\texttt{toExponential(fractionDigits)}\text{''}\ ) &= \\
prt(\text{``}\texttt{toPrecision(precision)}\text{''}\ ) &=
\end{aligned}
$$

$$
\begin{aligned}
h(L_{nc}) &= (obj,) \\
obj(\text{``}\texttt{prototype}\text{''}\ ) &= singleObj(L_{fp}) \\
obj(\text{``}\texttt{length}\text{''}\ ) &= 1 \\
obj(\text{``}\texttt{MAX\_VALUE}\text{''}\ ) &= \{DontEnum, DontDelete, ReadOnly\} \\
obj(\text{``}\texttt{MIN\_VALUE}\text{''}\ ) &= \{DontEnum, DontDelete, ReadOnly\} \\
obj(\text{``}\texttt{NaN}\text{''}\ ) &= \{DontEnum, DontDelete, ReadOnly\} \\
obj(\text{``}\texttt{NEGATIVE\_INFINITY}\text{''}) &= \{DontEnum, DontDelete, ReadOnly\} \\
obj(\text{``}\texttt{POSITIVE\_INFINITY}\text{''}) &= \{DontEnum, DontDelete, ReadOnly\}
\end{aligned}
$$

Свойства объекта Boolean:

$$
\begin{aligned}
h(L_b) &= (boo,) \\
boo(\text{``}\texttt{prototype}\text{''}\ ) &= singleObj(L_{bp})\{DontEnum, DontDelete, ReadOnly\} \\
boo(\text{``}\texttt{constructor}\text{''}\ ) &= singleObj(L_{bc})() \\
boo(\text{``}\texttt{call}\text{''}\ ) &= toBoolean() \\
boo(\text{``}\texttt{[[Value]]}\text{''}\ ) &=
\end{aligned}
$$

$$
\begin{aligned}
h(L_{bp}) &= (prt,) \\
prt(\text{``}\texttt{prototype}\text{''}\ ) &= singleObj(L_{fp}) \\
prt(\text{``}\texttt{constructor}\text{''}\ ) &= \text{встроенный конструктор} \\
prt(\text{``}\texttt{[[Value]]}\text{''}\ ) &= false \\
prt(\text{``}\texttt{[[Class]]}\text{''}\ ) &= Boolean \\
prt(\text{``}\texttt{toString()}\text{''}\ ) &= \\
prt(\text{``}\texttt{valueOf()}\text{''}\ ) &=
\end{aligned}
$$

$$
\begin{aligned}
h(L_{bc}) &= (con,) \\
con(\text{``}\texttt{prototype}\text{''}\ ) &= singleObj(L_{fp}) \\
con(\text{``}\texttt{length}\text{''}\ ) &= 1
\end{aligned}
$$

Свойства объекта String:

$$h(L_s) = (str,)$$
$$str(\text{``}\texttt{prototype}\text{''}\ ) = singleObj(L_{sp})\{DontEnum, DontDelete, ReadOnly\}$$
$$str(\text{``}\texttt{constructor}\text{''}\ ) = singleObj(L_{sc})()$$
$$str(\text{``}\texttt{call}\text{''}\ ) = toString()$$
$$str(\text{``}\texttt{[[Value]]}\text{''}\ ) =$$
$$str(\text{``}\texttt{fromCharCode([char0[, char1 [,...]]])}\text{''}) =$$

$$h(L_{sp}) = (prt,)$$
$$prt(\text{``}\texttt{prototype}\text{''}\ ) = singleObj(L_{fp})$$
$$prt(\text{``}\texttt{constructor}\text{''}\ ) = \text{встроенный конструктор}$$
$$prt(\text{``}\texttt{[[Value]]}\text{''}\ ) = \text{``}''$$
$$prt(\text{``}\texttt{[[Class]]}\text{''}\ ) = String$$
$$prt(\text{``}\texttt{toString()}\text{''}\ ) =$$
$$prt(\text{``}\texttt{valueOf()}\text{''}\ ) =$$
$$prt(\text{``}\texttt{charAt(pos)}\text{''}\ ) =$$
$$prt(\text{``}\texttt{charCodeAt(pos)}\text{''}\ ) =$$
$$prt(\text{``}\texttt{concat([string1[, string2 [,...]]])}\text{''}\ ) =$$
$$prt(\text{``}\texttt{indexOf(searchString, position)}\text{''}\ ) =$$
$$prt(\text{``}\texttt{lastIndexOf(searchString, position)}\text{''}\ ) =$$
$$prt(\text{``}\texttt{localeCompare(that)}\text{''}\ ) =$$
$$prt(\text{``}\texttt{match(regexp)}\text{''}\ ) =$$
$$prt(\text{``}\texttt{replace(searchValue, replaceValue)}\text{''}\ ) =$$
$$prt(\text{``}\texttt{search(regexp)}\text{''}\ ) =$$
$$prt(\text{``}\texttt{slice(start, end)}\text{''}\ ) =$$
$$prt(\text{``}\texttt{split(separator, limit)}\text{''}\ ) =$$
$$prt(\text{``}\texttt{substring(start, end)}\text{''}\ ) =$$
$$prt(\text{``}\texttt{toLowerCase()}\text{''}\ ) =$$
$$prt(\text{``}\texttt{toLocaleLowerCase()}\text{''}\ ) =$$
$$prt(\text{``}\texttt{toUpperCase()}\text{''}\ ) =$$
$$prt(\text{``}\texttt{toLocaleUpperCase()}\text{''}\ ) =$$

Экземпляры строк обладают свойствами `[[Value]]` и length.

$$h(L_{sc}) = (con,)$$
$$con(\text{``}\texttt{prototype}\text{''}\ ) = singleObj(L_{fp})$$
$$con(\text{``}\texttt{length}\text{''}\ ) = 1$$

$$L_{undef} = \text{undefined}$$

5.8 Завершаемость анализа

При ограничении на длину строк $\mathcal{K}$ и ограничении на максимальную длину иерархии объектов областей видимости $\mathcal{S}$ решётка $\mathcal{L}$ будет конечна. Поэтому, по теореме о неподвижных точках, сформулированной в главе 5.1.2, решётка $\mathcal{L}$ имеет минимальную неподвижную точку. Каждая функция переноса из множества $\mathcal{F}$ монотонна. Поэтому, девяткой $(\mathcal{L}, \mathcal{F}, E, I, i, \tau, \mathcal{M}, m, w)$,

где E — рёбра графа потока управления программы, размеченного функцией m, I — множество её точек входа, можно задать анализ вычисляемый изменённым алгоритмом, описанным в главе 5.1.2.

6 Экспериментальное исследование

6.1 Исследование точности анализа значений

6.1.1 Цель

Точность анализа достижимости зависит от точности анализа значений. Целью этого экспериментального исследования является оценка

6.1.2 Методика

1. Запуск анализатора, вычисление абстрактного состояния после каждой инструкции ветвления. Сбор трассы $\tilde{T}$.
2. Инстументирование скрипта
   - Преобразование исходного кода скрипта в абстрактное ситнаксическое дерево при помощи Closure Compiler.
   - Добавление вызова специальной функции, записывающей в журнал значение всех переменных в текущей области видимости, после каждой инструкции цикла и ветвления
   - Генерация исходного кода по измененному абстрактному синтаксическому дереву при помощи Closure Compiler.
3. Запуск инструментированного скрипта при помощи ringojs. Сбор трассы T.
4. Сравнение трасс: вычисление метрик точного совпадения $\mu\left(T, \tilde{T}\right)$.

6.1.3 Исходные данные

https://github.com/mootools/mootools-core

https://github.com/mootools/mootools-core-specs

6.1.4 Результаты

7 Список литературы

[1] *Crockford, D.* The world's most misunderstood programming language has become the world's most popular programming language. — 2008. — 03. http://javascript.crockford.com/popular.html.

[2] Standard ecma-262 // *ECMA Standardizing Information and Communication Systems.* — 2009. — Vol. 5.

[3] *Jang, D.* Points-to analysis for javascript / D. Jang, K. Choe // Proceedings of the 2009 ACM symposium on Applied Computing / ACM. — 2009. — Pp. 1930–1937.

[4] *Heintze, N.* Set based program analysis: Ph.D. thesis / Citeseer. — 1992.

[5] *Jensen, S.* Type analysis for javascript / S. Jensen, A. Møller, P. Thiemann // *Static Analysis.* — 2009. — Pp. 238–255.

[6] *Nielson, F.* Principles of program analysis / F. Nielson, H. Nielson, C. Hankin. — Springer-Verlag New York Inc, 1999.

[7] *Balakrishnan, G.* Recency-abstraction for heap-allocated storage / G. Balakrishnan, T. Reps // *Static Analysis.* — 2006. — Pp. 221–239.

[8] *Jensen, S.* Interprocedural analysis with lazy propagation / S. Jensen, A. Møller, P. Thiemann // *Static Analysis.* — Pp. 320–339.

[9] *Might, M.* Improving flow analyses via ΓCFA: Abstract garbage collection and counting / M. Might, O. Shivers // *ACM SIGPLAN Notices.* — 2006. — Vol. 41, no. 9. — Pp. 13–25.

[10] *Guarnieri, S.* Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code / S. Guarnieri, B. Livshits // Proceedings of the 18th conference on USENIX security symposium / USENIX Association. — 2009. — Pp. 151–168.

[11] Rhino: Javascript for java. http://www.mozilla.org/rhino/.