Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Στατική Ανάλυση για Εύρεση Λαθών σε Προγράμματα JavaScript

Διπλωματική Εργασία

του

Θεοδώρου Κασαμπαλή

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Ιούλιος 2012

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Στατική Ανάλυση για Εύρεση Λαθών σε Προγράμματα JavaScript

Διπλωματική Εργασία

του

Θεοδώρου Κασαμπαλή

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23^{η} Ιουλίου, 2012.

....	....	....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Κώστας Κοντογιάννης
Αν. Καθηγητής Ε.Μ.Π.	Επικ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2012

....
Κασαμπαλής Θεόδωρος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η JavaScript είναι μία δημοφιλής γλώσσα προγραμματισμού, που χρησιμοποιείται κυρίως για προγραμματισμό στον ιστό, αλλά και για προγραμματισμό γενικού σκοπού. Η γλώσσα αυτή έχει δυναμικό και χαλαρό σύστημα τύπων, και κατά συνέπεια τα λάθη σε προγράμματα JavaScript είναι δύσκολο να εντοπιστούν. Παρόλα αυτά, η υπάρχουσα υποστήριξη από αυτόματα εργαλεία για τη γλώσσα είναι περιορισμένη. Η εργασία μας στοχεύει στη βελτίωση αυτής της κατάστασης. Αναπτύσσουμε μία διαδικασία στατικής ανάλυσης για προγράμματα JavaScript που καλύπτει όλα τα χαρακτηριστικά της γλώσσας. Η ανάλυσή μας είναι ικανή να εξάγει πληροφορίες σχετικές με τους τύπους κατά την εκτέλεση, τους γράφους κλήσεων, τη διάδοση εξαιρέσεων και τη δομή των αντικειμένων, χρησιμοποιώντας μια ικανοποιητική αφαίρεση για ολόκληρο το runtime σύστημα της γλώσσας και για τον ασυνήθιστο μηχανισμό προτοτύπων που διαθέτει. Σε αυτή τη διπλωματική, παρουσιάζουμε λεπτομερώς αυτή την αφαίρεση και την αντιστοιχία της με την προδιαγραφή της γλώσσας. Επίσης, περιγράφουμε τον αλγόριθμο στατικής ανάλυσης που χρησιμοποιήθηκε και δίνουμε ενδεικτικά παραδείγματα.

Λέξεις Κλειδιά

Στατική Ανάλυση, Εξαγωγή Τύπων, Ανάλυση Ροής Δεδομένω, JavaScript

Abstract

JavaScript is a popular programming language, that is mostly used for client-side web scripting, but also for general purpose programming. The language is dynamically and weakly typed, and thus errors in JavaScript programs are difficult to spot. However the existing tools that support JavaScript programmers are limited. Our work aims to the improvement of this situation: We develop a framework for static analysis of JavaScript programs that covers all the features of the language. Our framework is able to infer information about the runtime types, call graphs, exception flow and object structure by successfully abstracting the language's whole runtime environment and its uncommon prototype mechanism. In this theses, we present in detail our abstraction and its correspondence with the JavaScript specification. Also, the static analysis algorithm used by our framework is described and demonstrated.

Keywords

Static Analysis, Type Inference, Dataflow Analysis, JavaScript

Contents

List of Listings

Chapter 1

Introduction

JavaScript is a widely used programming language. Its main use lies in client-side web scripting: Most of the dynamic features of a web page are due to embedded JavaScript code in its HTML description and almost every web browser ships with a JavaScript Engine in order to be able to present such dynamic content to the user. The main reason behind JavaScript's success as a scripting language is its extremely flexible nature that encourages quick-and-dirty programming. Indeed, JavaScript always tries its best not to fail the programmer: it is dynamically typed, so that the programmer does not have to declare types for the variables, and supports almost every possible implicit type conversion in order to relieve the programmer from the need of checking type mismatches and correcting them with explicit type castings. Even reads of non-existing object properties will not fail, since the undefined value, which they return, can be further propagated with the necessary conversions.

However, this flexibility does not come without a cost. The inability of the language to impose a more strict programming paradigm, is a drawback when it comes to the design and implementation of larger-scale applications. What is more, the debugging process of such applications is very hard, since most semantic errors are not going to cause program failure, but instead they will lead to unexcepected results, the source of which is difficult to be traced back.

JsAnalyzer is a static analysis framework, that intends to facilitate the debugging process of JavaScript programs. The framework supports a full scale dataflow analysis of the input program, that infers the possible types of the program variables, the exception flow and the program call graph. The information can be used to discover errors or to assist other more sophisticated static analyses.

The rest of this diploma thesis is organized as follows. In Chapter 2 we briefly present the JavaScript language and some of its current implementations. We also refer to the monotone framework algorithm, which is used for dataflow analysis by the JsAnalyzer framework. In Chapter 3 we describe the design and the implementation of the dataflow analysis and give examples of use. Finally, Chapter 4 contains related work and suggests possibilities for further work on the topics of this thesis.

Chapter 2

Background

2.1 The JavaScript language

JavaScript is a dynamically typed, interpreted language. It is classified as a scripting language and, like most languages of this category, it is a procedural language with some functional and object oriented characteristics. JavaScript offers the usual C-like control flow structures (i.e. `if`, `switch`, `for` and `while` statements) as well as an exception handling mechanism with try-catch clauses, similar to that of Java. As far as the object oriented aspect of the language is concerned, JavaScript uses a prototype-based mechanism to implement classes and inheritance and Javascript objects are hash table-like structures, that map property names to Javascript values of any type. On the other hand, JavaScript's functional characteristics include first-class and higher-order functions and function literals that allow the programmer to define anonymous function closures, that can be used like ordinary Javascript values.

In the following paragraphs we will briefly present some important features of the JavaScript language, such as its type system and the prototype-based mechanism. Also, we will refer to the most widely used JavaScript implementations.

2.1.1 Type System

As stated before, JavaScript is a dynamically typed language. Also it is weakly typed, since almost any operation triggers type conversions instead of failing, if the types of the arguments are not the ones expected by this operation. The possible types of JavaScript values during the execution of a JavaScript program are:

Undefined Type The Undefined type contains exactly one value, the `undefined` value, which is used as the initial value of every variable that has not be assigned a value.

Null Type The Null type contains exactly one value: `null`.

Boolean Type The Boolean type contains two values: `true` and `false`. Any JavaScript value can be converted to a value of type Boolean with the internal procedure `To-Boolean`, as it is defined in the specification of the JavaScript language.

String Type Values of type String are sequences of zero or more 16-bit unsigned integers, that correspond to UTF-16 units. JavaScript strings are immutable values. Any JavaScript value can be converted to a value of type String with the internal procedure `ToString`, as it is defined in the specification of the JavaScript language.

Number Type Values of type Number are 64-bit floating-point numbers, that conform with the double-precision 64-bit IEEE 754 format. Some special subtypes of the Number type are the Integer type (contains only integral Number type values), the Int32 type (contains the Integer type values less than 2^{31} and greater than or equal with -2^{31}), the UInt32 type (contains the Integer type values less than 2^{32} and greater than or equal with 0) and the UInt16 type (contains the Integer type values less than 2^{16} and greater than or equal with 0). Any JavaScript value can be converted to a value of type Number or its subtypes with the internal procedures `ToNumber`, `ToInteger`, `ToInt32`, `ToUInt32` and `ToUInt16`. These are defined in the specification of the JavaScript language.

Object Type Values of type Object are sets of properties. Each property has a name of type String, a value of any JavaScript type and zero or more attributes. There are three property attributes: `ReadOnly`, `DontDelete` and `DontEnum`. One can read and modify the value of a property, or even delete the property using its name, as long as the attributes of the property permit the operation. If the property has the `ReadOnly` attribute, attempts to modify its value will be ignored. If the property has the `DontDelete` attribute, attempts to delete it will be ignored. Properties with the `DontEnum` attribute do not appear in enumerations performed by `for/in` loops. Finally, trying to access a non-existing property of an object will return the `undefined` value.

Also every object value carries some additional information, like a prototype value and a default value. The prototype value will be explained later (see Section 2.1.3). The default value is used for the conversion of the object to other types and vice versa. There is no seperate function type in JavaScript, instead functions are special values of the Object type.

JavaScript values of types other than Undefined and Null can be converted to a value of type Object with the internal procedure `ToObject`, as it is defined in the specification of the JavaScript language. However, attempts to convert the `null` and `undefined` values to objects will raise an exception.

The conversion procedures described above are always used when a value of any type is used in a context, where a different type is needed. The only cases, that raise an exception, are the attempts to convert a `null` or `undefined` value to an object. Also, attempts to call a value that is not a function object always raise an exception.

2.1.2 Variable Scope

A JavaScript program consists of a script that contains statements, variable definitions and function definitions. The body of a function has the same structure, so nesting of function definitions is allowed. Also, functions may be defined as function literals anywhere a value is needed in the program. During the execution of the main script or a function, a scope chain is associated with the execution context. The scope chain is actually a list of objects with properties that correspond to the defined variables at the moment. These objects are added to the scope chain whenever a function is called, during parameter passing.

Initially, the scope chain associated with the execution of the main script contains only one object, the global object. The properties of this object are defined in the specification of the Javascript language and correspond to every built-in object or function of the language. Before the execution of the main script starts, a property is added to the global object for every variable definition contained in the main script, with the name of the variable as its name and `undefined` as its initial value. Also, for every function definition contained in the main script, a property is added to the global object, with name of the function as its name. The initial value of these properties are newly created function objects. Every such function object has also a scope chain associated with it, which is the current scope chain of the execution context of the main script. Function objects for function literals are similarly created, but their creation takes place when and if the value of the literal is needed, during execution.

When a function is called, an object, called variable object, is created. This object has properties with the names of the formal parameters of the function, and with the values of the actual parameters of the function call. The variable object followed by the scope chain of the called function object make the scope chain, that is associated with the execution context of the called function. Before the execution of the called function, properties are added to the variable object for every variable and function definition contained in the function body, in a way similar to the one described above for the global object.

Therefore, a variable name, anywhere in the program, refers to a corresponding property of the first object of the current scope chain, that contains a property with this name. If there is no such property in any of the scope chain's objects and the variable is used as an r-value then an exception is raised. Else, if the variable is used as an l-value then a new property is defined in the global object (which is always the last object of any scope chain), and the variable refers to that property.

Finally, one can add an abritary object to the front of the current scope chain by using a `with` statement. After the body of the statement is executed, the object is removed from the current scope chain.

2.1.3 Objects and the Prototype Mechanism

Objects are created using object literals or constructor functions. Object literals are expressions that evaluate to an object value, which contains the name-value pairs described in the literal. Constructor functions are used to initialize an object created by a `new` expression. A `new` expression consists of the keyword `new` and a function call. When evaluated, it creates an empty object and then it calls the function. What is special about this call is the fact that, during the execution of the called function the `this` keyword evaluates the newly created object. Thus, the constructor functions are able to use the `this` keyword to define and initialize the properties of the newly created object.

The `this` keyword is also important when a function is called as a method, meaning it is invoked as property of an object. In this case, during the execution of the method the `this` keyword evaluates to the object, through which the method was invoked. Therefore, the methods can access and modify properties of the object, through which they were invoked. Finally, in a normal function call, during the execution of the called function the `this` keyword evaluates to the global object.

Every object (but one) has a prototype value associated with it. This value is a pointer to another object, which is called the prototype object. Whenever a property of an object is requested, but it is not found in the object, the same property is requested from the prototype object. As a result, the object inherits all the properties of its prototype. The search continues until the last object of the prototype chain is examined. In case the property is still not found, it is considered to have the `undefined` value. However, this is true only when a property is requested for reading. If one tries to modify a non-existing property of an object, a new property is created in the object, instead of attempting to modify a property with the same name in the prototype object.

Every function object, either user-defined or built-in, has a property with name `prototype` and an object value. This value is different from the prototype object of the function object! When an object is created with a `new` expression, its prototype object is set to be the object that is the value of the `prototype` property of the constructor function. The value of the `prototype` property of a built-in constructor is usually another built-in object with various useful properties and methods, that will be inherited by the newly created objects. For instance, the `prototype` property of the built-in `Object` constructor contains methods to be inherited by the objects created with the constructor. Since every object, created by an object literal expression, is actually created by the `Object` constructor, every such object has the `Object.prototype` object as its prototype. On the other hand, the value of the `prototype` property of a user-defined function object is an empty object, in which one can add desired functionality in case the user-defined function is about to be used as a constructor. This empty object also has the `Object.prototype` object as its prototype. Generally speaking, every prototype chain ends with the `Object.prototype` object, which is the only object without a prototype.

2.1.4 Dynamically Generated Code

The JavaScript language specification defines a built-in function, `eval`, and a built-in constructor, `Function`, that allow part of the program code to be dynamically created and executed. The `eval` function accepts a string argument, that is parsed and executed as JavaScript code. The `Function` constructor accepts, various string arguments that stand for the name, the formal parameters and the body of a function, and it creates a corresponding function object. The main reason for using these features is for dynamic generation of code driven from some kind of external input. The static analysis of programs, that use these features, presents inherent difficulties, because the dynamically generated code cannot be easily predicted statically as well as the number and the bodies of the functions are not statically known.

2.1.5 JavaScript Implementations

The JavaScript language is interpreted and the implementation should provide various functionalities in the form of built-in objects and functions (accessed as object methods). The core language and its built-in functionalities are standardized by ECMA International. The more widely-used standard is the ECMA 262, edition 3 [3], while the latest standard is ECMA 262, edition 5.1 [4]. Almost all existing implementations (mainly browser implementations) conform with the standard specification, but also provide various language extensions.

The most common use of JavaScript is for client-side web scripting. JavaScript scripts can be imported to HTML documents and sent to a browser along with the HTML. Almost every web browser comes with a JavaScript implementation, so it is able to execute JavaScript scripts in the user's terminal in order to interact with him and dynamically modify the presented content. To this end, the JavaScript implementations of browsers include additional functionality for interacting with the user and accessing and modifying the content of HTML documents. This additional functionality is included in the so called Document Object Model (DOM) interface of the browser. According to W3C [14], the DOM is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. W3C has created a specification of the features that a DOM must support in order to enhance inter-browser compatibility of client-side scripts (in any language, including JavaScript).

The Rhino Interpreter [9] is the most widely used implementation of the core JavaScript language. It is an open source project and it conforms with the 3rd edition of the ECMA standard. Another significant implementation is the SpiderMonkey JavaScript Engine [11], an open source project developed by Mozilla. It is used in Firefox web browser and many other Mozilla projects and conforms with the 5th edition of the ECMA standard. Finally, the v8 JavaScript Engine [13] is an open source project developed by Google. It is used in Google Chrome web browser, but it can also be used as a standalone interpreter. The v8 engine also conforms with the 5th edition of the ECMA standard.

This work is based on the 3rd edition of the ECMA standard for the JavaScript language.

2.2 Dataflow Analysis with the Monotone Framework Algorithm

The monotone framework algorithm is a fixpoint algorithm used for the solution of many dataflow related problems in flow graphs. To describe the algorithm we need the following definitions:

- A flow graph G with a set of nodes N and a set of directed edges E. The set of nodes N should contain two special nodes, *ENTRY* and *EXIT*. The *ENTRY* node should have only outcoming edges attached to it, while the *EXIT* node should have only incoming edges.
- A domain V of values and a binary join operator $\wedge$ for the values of V. The values will be assigned to the nodes and edges of the flow graph. The value assigned to the node $n \in N$ is stored in $V_N[n]$ and the value assigned to the edge $e \in E$ is stored in $V_E[e]$.
- A set F of transfer functions from V to V. Every node and edge of the flow graph is assigned one transfer function from the set F. Let f_n be the transfer function assigned to the node $n \in N$, and f_e be the transfer function assigned to the edge $e \in E$. The special nodes *ENTRY* and *EXIT* should be assigned an identity transfer function.

The join operator $\wedge$ should satisfy the following properties:

- $\forall x \in V$: $x \wedge x = x$
- $\forall x,y \in V$: $x \wedge y = y \wedge x$
- $\forall x,y,z \in V$: $x \wedge (y \wedge z)= (x \wedge y) \wedge z$
- There is a bottom element $\perp \in V$ such that: $\forall x \in V$: $x \wedge \perp = x$

The join operator defines a partial order ($\leq$) for the values of the domain V:

$$\forall x,y \in V\colon x \leq y \Longleftrightarrow x \wedge y = y$$

The proof the above defined relation ($\leq$) is a partial order of the values of the domain V has been omitted here, but it is based on the properties of the join operator. It is obvious that

$$\forall x \in V\colon \perp \leq x$$

The $\{ V, \wedge, F\}$ triple is called a dataflow framework. A dataflow framework is monotone if

$$\forall x,y \in V \text{ and } \forall f \in F\colon x \leq y \Longrightarrow f(x) \leq f(y)$$

The algorithm computes a value from the domain V for every node and edge of the flow graph. These values are computed as the fixpoint of an iterative process. Firstly, the algorithm assignes an initial input value v_{in} to the *ENTRY* node and the $\perp$ value to every other node and edge of the flow graph. Then, it applies the transfer functions related with the nodes and edges of the graph in order to update their values. The process is repeated until no change in the values of the nodes and edge is observed, thus a fixpoint has been reached. The algorithm is shown in Listing 2.1.

```
$V_N$[ENTRY] = $v_{in}$;

for (every $n \in N$ with $n \neq$ ENTRY)
  $V_N$[n] = $\perp$;

for (every $e \in E$)
  $V_E$[e] = $\perp$;

while (changes to states of nodes and edges occur)
  for (every $n \in N$ with $n \neq$ ENTRY) {
    $V_N$[n] = $\bigwedge_{e\ an\ incoming\ edge\ to\ n}$ $f_e$($V_E$[e]);
    for (every $e \in E$ with e an outcoming edge from n)
      $V_E$[e] = $f_n$($V_N$[n]);
  }
```

Listing 2.1: The monotone framework algorithm

The algorithm is guaranteed to terminate, if the dataflow framework is monotone and the height of the semilattice for V, defined from the partial ordering $\leq$, is finite. However, these are just sufficient conditions for the termination of the algorithm, so it may terminate for a framework with a semilattice of infinite height.

Chapter 3

JsAnalyzer Design and Implementation

In this Chapter we will present the design and some implementation details of a framework for performing static analysis to JavaScript programs. Our framework uses an abstract interpretation of the state of a JavaScript program and performs dataflow analysis to estimate the abstract state in every point of the input program. During the analysis, we can infer information about the types of the program variables and the actual arguments of functions, the call graph of the input program, and the possible exception flow. Of course, the analysis may produce over approximated results.

The Ocaml programming language [8] was used for the implementation of our framework.

3.1 Intermediate Representation of JavaScript Programs

The intermediate representation (IR) used by our framework consists of control flow graphs for every function or function literal that appears in the input program, as well as a call graph, that stores the caller-callee relations that are discovered through the analysis. The control flow graphs are constructed in the phase of parsing of the input program, while the call graph is updated on-the-fly during the actual dataflow analysis.

In particular, for every function and function literal that appears in the input program, the following infromation is stored by out framework:

- The function allocation site id (see Section 3.1.1)
- The function kind (main script, defined function, anonymous function literal or named function literal)
- The names and order of the formal parameters
- The names and order of the variable definitions in the body of the function
- The names, the allocation site ids (see Section 3.1.1), and order of the function definitions in the body of the function
- The control flow graph of the function (see Section 3.1.2)

3.1.1 Basic data types for the IR

For our intermadiate representation we need some basic types, which are described below.

name The type `name` is used for the internal representation of JavaScript strings (see Section 2.1.1), and it is a list of integers. Listing 3.1 shows the interface for the `name` type.

tid The type `tid` is used for the internal representation of the temporary variables created during the construction of the control flow graphs. Composite expressions, which combine many operators and arguments, are being broken down to binary operations whith the introduction of such temporary variables. Listing 3.2 shows the interface for the `tid` type.

allocation_site_id The type `allocation_site_id` is used for the internal representation of the points in the program source code, where new objects are possibly created. These points are discovered during the dataflow analysis, with one exception: Function definitions and uses of function literals are assigned an allocation site id, during the parsing phase, and these ids are also used as references for the corresponding functions. Listing 3.3 shows the interface for the `allocation_site_id` type.

call_site_id The type `call_site_id` is used for the internal representation of the points in the program source code, where a function is called. These points are, also, discovered during the dataflow analysis. Listing 3.4 shows the interface for the `call_site_id` type.

```
type name = int list
module NameMap : Map.S with type key = name
val empty_name : name
val length_of_name : name -> int
val name_char_at : name -> int -> name
val concat_names : name -> name -> name
val name_of_string : string -> name
```

Listing 3.1: `name` type interface

```
type tid
module TmpMap : Map.S with type key = tid
val same_tid : tid -> tid -> bool
val compare_tids : tid -> tid -> int
```

Listing 3.2: `tid` type interface

```
type allocation_site_id
val same_allocation_site_id : allocation_site_id -> allocation_site_id -> bool
val compare_allocation_site_ids : allocation_site_id -> allocation_site_id -> int
val hash_allocation_site_id : allocation_site_id -> int
val get_alloc_site_id : unit -> allocation_site_id
```

Listing 3.3: `allocation_site_id` type interface

```
type call_site_id
val compare_call_sites : call_site_id -> call_site_id -> int
val dummy_call_site : call_site_id
val get_new_call_site_id : unit -> call_site_id
```

Listing 3.4: `call_site_id` type interface

3.1.2 Control Flow Graph

The control flow graphs, produced by our framework, represent all the possible execution paths of the source program, taking into account both the normal control flow and the exception handling. Every node of a control flow graph is labeled with the operation that this node represents along with the operands of this operation. The Listing 3.5) shows the data type for the node labels, but the definitions of the `unop` and `binop` types are suppressed for space efficiency. These types are enumerations of the unary and binary operators of the JavaScript language. The unary operators `delete` and `typeof` are associated with special control blocks, because their behaviour is slightly different when they are given an object property as argument. Also, there are no compound assignment operators (e.g. `+=`), since the compound assignments are further analyzed in their two steps.

We will further explain the use of some important constructors of the `control_block` type, which is the type for node labels:

`Start_block, End_block, Exception_end_block` These labels are given to the entry node and the exit nodes of the control flow graph, correspondingly. Execution paths that end at the `Exception_end_block` are those that contain a raise of an uncaught (inside the function) exception.

`Unop_block (temp, unop, arg), Binop_block (temp, binop, arg1, arg2)` These labels are given to nodes that represent various JavaScript operations.

`If_block cond, Ifop_block (binop, arg1, arg2)` These labels are given to nodes that represent a conditional flow. In the first case, the flow is determined by the boolean value of the argument `cond`, while in the second case, the flow is determined by the boolean result of the application of the `binop` operator to the arguments `arg1` and `arg2`. Only comparison operators are allowed as argumnets of the second construnctor.

`GetValue_block (temp, rvalue), Assign_block (lvalue, rvalue)` These labels are given to nodes that represent reads and writes from and to variables.

```
type control_block =
    Start_block
  | End_block
  | Exception_end_block
  | If_block of operand
  | Ifop_block of binop * operand * operand
  | ToObject_block of tid * operand
  | EnumProperty_block of tid * operand
  | Ret_block of operand option
  | Throw_block of operand
  | Enter_catch_block of name
  | Exit_catch_block
  | Enter_with_block of operand
  | Exit_with_block
  | Assign_block of operand * operand
  | GetValue_block of tid * operand
  | Unop_block of tid * unop * operand
  | Binop_block of tid * binop * operand * operand
  | Read_property_block of tid * operand * operand
  | Write_property_block of operand * operand * operand
  | Delete_variable_block of tid * name
  | Delete_property_block of tid * operand * operand
  | Typeof_variable_block of tid * name
  | Typeof_block of tid * operand
  | Call_block of tid * operand * operand list
  | Call_method_block of tid * operand * operand * operand list
  | New_block of tid * operand * operand list
  | ObjectInitializer_block of tid * (prop_name * operand) list
  | ArrayInitializer_block of tid * array_elem list
  | RegExpInitializer_block of tid * (string * string)
  | FunctionObjInitializer_block of tid * allocation_site_id
and prop_name =
    PropString of name
  | PropNum of float
and array_elem =
    SkippedElem
  | Elem of operand
and unop (* = ... *)
and binop (* = ... *)
and operand =
    Variable_operand of name
  | Temporary_operand of tid
  | Number_operand of float
  | String_operand of name
  | Boolean_operand of bool
  | Null_operand
  | This_operand
```

Listing 3.5: Control flow graph node labels

`Read_property_block (temp, base, index), Write_property_block (base, index, rvalue)`
These labels are given to nodes that represent reads and writes from and to object properties.

`Delete_property_block (temp, base, index), Delete_variable_block (temp, var)`
These labels are given to nodes that represent delete operations. The `temp` argument is of type `tid` and serves for storing the operation's result, which is a boolean value that indicates whether the delete was successful.

`Call_block (temp, func, args), Call_method_block (temp, base, index, args)`
These labels are given to nodes that represent function and method calls. After the return of the call the flow continues to the successors of the call node.

`New_block (temp, func, args)` This label is given to nodes that represent calls to constructor fuctions. After the return of the call the flow continues to the successors of the call node.

`Throw_block arg` This label is given to nodes that represent the explicit raise of an exception.

`Enter_catch_block name, Exit_catch_block` These labels are given to nodes that represent exception handling operations. Every execution path that contains an enter-catch node, also contains a corresponding exit-catch node.

`Enter_with_block arg, Exit_with_block` These labels are given to nodes that represent entrance to and exit from the body of a `with` statement. Every execution path that contains an enter-with node, also contains a corresponding exit-with node.

Every node of a control flow graph can be registered as an allocation site or/and a call site, by assigning him an appropriate id value. As stated before, this happens during the dataflow analysis.

The control flow graph nodes are connected with edges, according to the possible execution flow paths. Each edge is labeled with the kind of the flow that this edge represents (see Listing 3.6). The label `Normally` is given to edges that connect nodes that are possible to be consecutive in a normal excecution path. The labels `When_true` and `When_false` are given to edges which have a condition node as their source. Finally the `When_exception` label is given to edges that represent the execution flow, when the operation of their source node raises an exception.

3.1.3 Call Graph

The call graph that is stored in the intermediate representation is a typical one. It consists of nodes labeled with an allocation site id and edges labeled with a call site id. Every edge of the call graph indicates a caller-callee relation between the functions that correspond to the source's and destination's labels. The call point on the caller function source code is indicated by the call site id of the edge. The dataflow analysis adds edges to the call graph as it discovers them.

```
type flow_type =
    Normally
  | When_true
  | When_false
  | When_exception
```

Listing 3.6: Control flow graph edge labels

3.2 Abstract State for JavaScript Programs

During the dataflow analysis, an abstract state is assigned to the nodes and the edges of the control flow graphs of the IR. The abstract state of a node should summarize every possible program state when the execution reaches that node. We will present in detail the type of the abstract state used by our framework. This type should abstract the possible values of variables and properties, as well as the features of the runtime environment, such as the execution context and the scope chain. The type of the abstract state is based on the work of Jensen *et al.* [6].

3.2.1 Representation of JavaScript Values

JavaScript values can have any of the types described in Section 2.1.1. For every of these types we define an equivalent OCaml variant type, the constructors of which represent different subsets of the corresponding JavaScript type. The OCaml variant types are used to represent JavaScript values in our abstract state. A value is represented by an as strict as possible OCaml variant type. The following Listings 3.7 - 3.11 show the OCaml variant types along with the set of JavaScript values they represent.

Note that the compare functions return a tuple of three values. The first one, which has type `typeset_relation`, indicates the set relation between the compared types. (Possible relations are, the first being a subset/superset of the second or the two types being equal/intersecting/disjoint). The other two values are the union and the intersection of the compared types.

On the other hand, the cmp functions just provide an arbitary total ordering to the members of each variant type, which is useful for creating maps or sets for the type.

The compare, cmp, and equality testing fuctions for every OCaml variant type satisfy the following property for any pair of compared types: The equality testing fuction returns `true` if and only if the compare function also indicates that the compared types are equal and if and only if the cmp function returns a zero result.

We do not define an OCaml variant type for the JavaScript object type. The object values are represented by their allocation site ids. More precisely, they are represented by allocation sites. As stated before, allocation site ids are given to control flow nodes that describe the creation of an object. We define two allocation sites for every allocation site id: one singleton, that refers to the most recent object created at that site, and one summary that refers to the summary of the older objects created there. More details about allocation sites are given in the next section (see Section 3.3.2). The Listing 3.12 presents the definition and the interface of the `alloaction_site` type.

```
type js_undefined_type =
    Undefined        (* represents the JavaScript undefined value *)
  | Bottom_undefined (* represents the absence of an undefined value *)

val compare_undefined_types :
  js_undefined_type -> js_undefined_type ->
  typeset_relation * js_undefined_type * js_undefined_type
val equal_undefined_types : js_undefined_type -> js_undefined_type -> bool
val is_subtype_undefined : js_undefined_type -> js_undefined_type -> bool
val create_union_of_undefined : js_undefined_type list -> js_undefined_type
val cmp_undefined_types : js_undefined_type -> js_undefined_type -> int
```

Listing 3.7: Definition and interface for the `js_undefined_type` type

```
type js_null_type =
    Null        (* represents the JavaScript null value *)
  | Bottom_null (* represents the absence of a null value *)

val compare_null_types :
  js_null_type -> js_null_type -> typeset_relation * js_null_type * js_null_type
val equal_null_types : js_null_type -> js_null_type -> bool
val is_subtype_null : js_null_type -> js_null_type -> bool
val create_union_of_null : js_null_type list -> js_null_type
val cmp_null_types : js_null_type -> js_null_type -> int
```

Listing 3.8: Definition and interface for the `js_null_type` type

```
type js_string_type =
    Any_string       (* represents the JavaScript string type *)
  | UInt32Str        (* represents the set of JavaScript string values that *
                      * convert to UInt32 numbers                          *)
  | NotUInt32Str     (* represents the set of JavaScript string values that *
                      * do not convert to UInt32 numbers                   *)
  | ConstStr of name (* represents a sigleton set of a JavaScript string value *)
  | String_union of string_union (* represents a set of JavaScript string *
                                  * values that can only be described as   *
                                  * the union of sets represented by the   *
                                  * other constructors                    *)
  | Bottom_string    (* represents the absence of a string value *)
and string_union

val compare_string_types :
  js_string_type ->
  js_string_type -> typeset_relation * js_string_type * js_string_type
val equal_string_types : js_string_type -> js_string_type -> bool
val is_subtype_string : js_string_type -> js_string_type -> bool
val create_union_of_strings : js_string_type list -> js_string_type
val fold_string_union : (js_string_type -> 'a -> 'a) -> string_union -> 'a -> 'a
val filter_string_type :
  (js_string_type -> bool) -> js_string_type -> js_string_type
val cmp_string_types : js_string_type -> js_string_type -> int
```

Listing 3.9: Definition and interface for the `js_string_type` type

```
type js_number_type =
   Any_number        (* represents the JavaScript number type *)
 | Infinite          (* represents the positive and negative infinity *
                      * JavaScript values                             *)
 | UInt32            (* represents the UInt32 subtype of the JavaScript     *
                      * number type                                         *)
 | NotUInt32         (* represents the set of JavaScript number values that *
                      * do not belong to the UInt32 numbers and are not     *
                      * infinite neither NaN                                *)
 | NaN               (* represents the JavaScript NaN value *)
 | PosInf            (* represents the JavaScript positive infinity value *)
 | NegInf            (* represents the JavaScript negative infinity value *)
 | NegZero           (* represents the JavaScript negative zero value     *)
 | ConstNum of float (* represents a sigleton set of a JavaScript number value *)
 | Number_union of number_union (* represents a set of JavaScript number *
                                 * values that can only be described as   *
                                 * the union of sets represented by the   *
                                 * other constructors                     *)
 | Bottom_number     (* represents the absence of a number value *)
and number_union

val compare_number_types :
  js_number_type ->
  js_number_type -> typeset_relation * js_number_type * js_number_type
val equal_number_types : js_number_type -> js_number_type -> bool
val is_subtype_number : js_number_type -> js_number_type -> bool
val create_union_of_numbers : js_number_type list -> js_number_type
val fold_number_union : (js_number_type -> 'a -> 'a) -> number_union -> 'a -> 'a
val filter_number_type :
  (js_number_type -> bool) -> js_number_type -> js_number_type
val cmp_number_types : js_number_type -> js_number_type -> int
```

Listing 3.10: Definition and interface for the `js_number_type` type

```
type js_boolean_type =
    Any_boolean    (* represents the JavaScript boolean type *)
  | True           (* represents the JavaScript true value *)
  | False          (* represents the JavaScript false value *)
  | Bottom_boolean (* represents the absence of a boolean value *)

val compare_boolean_types :
  js_boolean_type -> js_boolean_type ->
  typeset_relation * js_boolean_type * js_boolean_type
val equal_boolean_types : js_boolean_type -> js_boolean_type -> bool
val is_subtype_boolean : js_boolean_type -> js_boolean_type -> bool
val create_union_of_booleans : js_boolean_type list -> js_boolean_type
val cmp_boolean_types : js_boolean_type -> js_boolean_type -> int
```

Listing 3.11: Definition and interface for the `js_boolean_type` type

```
type allocation_site =
    Singleton of allocation_site_id
  | Summary of allocation_site_id
module AllocSitesSet : Set.S with type elt = allocation_site
val same_allocation_site_id : allocation_site_id -> allocation_site_id -> bool
val compare_allocation_site_ids : allocation_site_id -> allocation_site_id -> int
val hash_allocation_site_id : allocation_site_id -> int
val get_alloc_site_id : unit -> allocation_site_id
```

Listing 3.12: `allocation_site` type interface

Finally, the type `value`, that is used to abstract the JavaScript values, is a union of all the previous types along with a set of allocation sites for the possible object values. The values of the `value` type are joined using the type union, returned by the corresponding compare function, for the non-object fields and the set union operation for the sets of object allocation sites.

```
type value = {
  number_value : js_number_type;
  string_value : js_string_type;
  boolean_value : js_boolean_type;
  undefined_value : js_undefined_type;
  null_value : js_null_type;
  object_locations : AllocSitesSet.t;
}
val join_values : value -> value -> value
val cmp_values : value -> value -> int
val equal_values : value -> value -> bool
```

Listing 3.13: Definition and interface for the `value` type

3.2.2 Representation of Objects and Execution Contexts

The abstract state should have a representation of the running execution context. This representation consists of the scope chain, that is associated with the execution context, and the allocation sites of the this object and the variable object. The scope chain is a list of object allocation sites. These concepts have been described in detail in Section 2.1.2. The definition of the `execution_context` and the `scope_chain` types is as follows is shown in the Listing 3.14.

Every allocation site should refer to an object in the heap of the abstract state. The `obj` type contains the information stored in a JavaScript object and its definition is shown in the Listing 3.15. It contains a map from property names to property values and attributes, the possible allocation sites of the prototype object, the default value, and a set of possible scope chains in case it is a function object. The `default_other` and `default_index` fields contain a possible default value for a property that does not exist in the property map of the object (the `default_index` field is used for property names that convert to UInt32 numbers). These values are used in read and write opearations when the property names are not specific (e.g. `Any_string` or `UInt32Str`). The `is_array` field indicates whether

```
type execution_context = {
  scope : scope_chain;
  var_object : allocation_site;
  arg_object : allocation_site;
  this_object : allocation_site;
}
and scope_chain = allocation_site list
module ExecutionContextSet : Set.S with type elt = execution_context
module ScopeChainSet : Set.S with type elt = scope_chain
val cmp_execution_contexts : execution_context -> execution_context -> int
val cmp_scope_chains : scope_chain -> scope_chain -> int
val equal_execution_contexts : execution_context -> execution_context -> bool
val equal_scope_chains : scope_chain -> scope_chain -> bool
```

Listing 3.14: Definition and interface for the `execution_context` type

or not the object is an array object. These objects require some special treatment, when a property of them is updated.

Every object property has an `is_absent` flag, that indicates whether the property is possibly absent. When two objects are joined to form a summarized object, if a property is absent in the one and present in the other, then it is present in the summary object but its `is_absent` flag is set to the value `Maybe_absent`.

3.2.3 Representation of the Abstract State

The representation of the abstract state combines the previous abstractions and is shown in the Listing 3.16. It contains a heap, that is a map from allocation sites to objects (values of the `obj` type), a similar map for the values of the temporary variables, and a set of possible execution contexts. We can always be sure that every allocation site anywhere in the abstract state representation is mapped to an actual object in the heap of the state. The field `object_refs` contains the set of all these allocation sites. So, using them as roots, one can perform abstract garbage collection to the heap of the state.

3.3 Details for the Dataflow Analysis

The dataflow analysis uses the monotone framework algorithm (see Section 2.2) in order to compute a state for every node and edge of the control flow graphs of the functions. So we need to define a join function for the abstract states and a set of transfer functions for every node and edge label. Also, we need to highlight various details about the analysis, such as how control flow is passed from a caller to a callee function and vice versa, how objects are assigned to their allocation sites and how many and which states for every node or edge are kept apart, in order for the analysis to be flow-sensitive. All these are presented to the following paragraphs.

```
type descriptor = {
  value       : value;
  is_absent   : absent;
  attributes  : attributes
}
and absent = Maybe_absent | Not_absent
and attributes = {
  is_read_only    : read_only;
  is_configurable : configurable;
  is_enumerable   : enumerable;
}
and read_only = Maybe_RO | RO | Not_RO | Undefined_RO
and configurable = Maybe_config | Config | Not_config | Undefined_config
and enumerable = Maybe_enum | Enum | Not_enum | Undefined_enum
type obj = {
  properties     : descriptor NameMap.t;
  prototype      : AllocSitesSet.t;
  default_value  : value;
  default_index  : value;
  default_other  : value;
  is_array       : js_boolean_type;
  function_scope : ScopeChainSet.t
}
val join_objects : obj -> obj -> obj
val equal_objects : obj -> obj -> bool
```

Listing 3.15: Definition and interface for the `obj` type

```
module ObjectMap : Map.S with type key = allocation_site
module TmpMap : Map.S with type key = ControlFlow.tid
type abstract_state = {
  object_store : obj ObjectMap.t;
  call_stack : abstract_stack;
}
and abstract_stack = {
  temp_store : value TmpMap.t;
  exec_context : ExecutionContextSet.t;
  object_refs : AllocSitesSet.t;
}
val join_abstract_states : abstract_state -> abstract_state -> abstract_state
val equal_abstract_states : abstract_state -> abstract_state -> bool
val gc_abstract_state : abstract_state -> abstract_state
```

Listing 3.16: Definition and interface for the `abstract_state` type

3.3.1 Join and Equal Functions for the Abstract States

The components of the abstract state that are maps or products (represented by OCaml maps and records respectively), are joined pointwise, while the sets (represented by OCaml sets) are joined using the set union function. As stated before, joining property maps of objects requires special treatment of the `maybe_absent` flag. As far as the values are concerned, they are joined by using the compare functions of the corresponding OCaml variant types. In particular, the join of two types is their union type.

Two abstract states are tested for equality by the `equal_abstract_states` function (see Listing 3.16). The function uses the equality testing functions for every component of the abstract states.

3.3.2 Object Allocation Sites and Recency Abstraction

Properties can be added and removed from a JavaScript object at any time during its life time, but it is most usual, that the object keeps the properties added to it when it was created, and only the values of these properties are modified from this point on. In order to accurately capture this common pattern in our analysis, we need to be able to perform strong updates in the abstract representation of an object. This is achieved with the use of recency abstraction [2]. In particular, we have already mentioned that objects are related with the points in the program code where they are created, or with their allocation sites. Moreover, for every point in the program where an object may be created, we define two allocation sites: one signleton and one summary allocation site. The singleton allocation site refers to the object most recently created from that site. The abstract representation of this object can be strongly updated from following operations. However, more than one objects may be created from the same site (i.e. if the creation happens inside a loop). The summary allocation site refers to an abstract object that summarizes all these objects. When a new object should be created, the object that refers to the corresponding singleton allocation site is joined with the object refering to the summary allocation site and also every reference to that signleton allocation site in the abstract state representation is changed to reference to the summary allocation site. Finally, a new object is created and it is assigned the singleton allocation site as its value.

An example is provided in Section 3.4.1

3.3.3 Interprocedural Analysis

Our analysis is interprocedural, meaning it takes into account function calls and returns. The semantics of JavaScript function calls have been described in detail in Section 2.1.2. Our analysis abstracts the process as follows: A new variable object is created and initialized according to the actual arguments of the call. The control flow graph node that represents the function call is registered as the allocation site of the variable object and as a call site. A new state is produced for every function object, that is a possible target of the call, by adding the variable object in the front of the scope chain (or the scope chains) associated with the object, and using the new scope chain to create the execution context (or the execution context set) for the new state. The `this` value of the execution context(s) depends on the type of the call. The object heap of the new state is that of the caller state. Also, for every possible callee function an edge is added in the call graph of the IR, if it has not already been added previously. The new state serves as the initial state of the callee function.

As far as the return process is concerned, the states, which are assigned to the exit nodes of the callee function, are returned to the caller. The return state of the caller has the same execution context with the state before the call, but the object heap is taken from the states returned from the callee. This return state is assigned to the outcoming edges of the node that represents the function call. As a result, the transfer functions of nodes that represent calls do not have a different behaviour from other transfer functions (see Section 3.3.5).

3.3.4 Flow-sensitive Analysis

Our analysis is flow-senstivite, that is it keeps more than one abstract states for every node, depending of the possible different call sequences that may lead to the function, in which the node belongs. A call sequence is the sequence of function calls that end to a specific function. The input states arriving to a function from different call sequences are kept separate and multiple instances of the dataflow analysis are performed. Although, the flow-sensitivity adds greatly to the accuracy of the analysis, it is not possible to keep an arbitrary number of states for the nodes of every function, if we want the analysis to terminate. Therefore, there is a configurable limit to that number. If the different call sequences leading to a function exceed that limit, the corresponding entry states are joined together and the analysis of this function (and of every function it calls) becomes flow-insensitive.

3.3.5 Transfer Functions

The transfer functions of the control flow graph nodes transform the input state that reaches the node and produce one, two or three output states. A normal output state is produced when there is no possibility of an exception to be raised in the node. If it is certain, given the input state and the node's label, that an exception will be raised, then only an exception output state is produced. Both a normal and an exceptional state are produced, when both possibilites exist. Finally, if the node represents a conditional flow, then two normal states are produced instead of one, for the cases the condition is true or false. The transfer functions assign the resulted states to appropriate outcoming edges. The normal output state can be assigned to edges with the `Normally` label, while the exception output state is assigned only to edges with the `When_exception` label. Normal output states of conditional flow nodes are assigned to edges with either the `When_true` or the `When_false` label.

There are no transfer functions for the control flow graph edges, since the assignment of different output states to differently labeled edges substitutes the functionality of edge transfer functions.

If the input state for a transfer function is not defined (i.e. because the algorithm has not propagated a state to the corresponding node yet), the input is considered to be a bottom state that always results to a bottom output state.

When a transfer function has to create an object, i.e. because of a necessary to-object conversion, it always registers the corresponding node as an allocation site. If the node is already registered, the transfer functions use recency abstraction (see Section 3.3.2) to create singleton and summary objects allocated from this node.

A brief description of some important transfer functions is following. The transfer functions are associated with the different node labels. Their actions abstract the corresponding procedures defined by the JavaScript language.

`Start_block, End_block, Exception_end_block` The transfer functions for these labels are identity functions, as the corresponding nodes serve only as starting and ending points of the analysis.

`GetValue_block (temp, rvalue)` The transfer function will assign the value of the `rvalue` operand to the temporary with id `temp`. In case the `rvalue` operand is a `Variable_operand` then its value will be looked for at the properties of the objects of the scope chain of the input state's execution context. If the input state has multiple execution contexts, a union of values will be produced (one for every scope chain).

`Assign_block (lvalue, rvalue)` The transfer function will assign the value of the `rvalue` operand to the l-value of the `lvalue` operand (it should be either a temporary, a variable or `this`). The variables are located as described above. If the l-value is variable and its name is not found in a scope chain, the variable is added as a property of the global object.

`Read_property_block (temp, base, index)` The transfer function will assign the value of the indexed property to the temporary with id `temp`. First, the value of the `base` operand should be converted to an object, using the abstract `ToObject` procedure, and the value of the `index` operand should be converted to a string, with the abstract `ToString` procedure. These and other abstract conversion procedures simulate the procedures defined for the respective JavaScript types, but they operate in abstract values. The computed property names are looked for at the possible objects, that resulted from the evaluation of the base, as well as in their prototype chains, if some possible property names are not found in them. In case of more general property names (like `Any_string`) the values of the fields `default_other` and `default_index` are also included in the result of the read.

`Write_property_block(base, index, rvalue)` The transfer function will assign the value of the `rvalue` operand to the object's property. As described above, the values of the `base` and `index` operands are converted to values of appropriate types, and the value of the `rvalue` operand is fetched from the input state. However, the r-value is assigned to properties of the objects that resulted from the `base` operand, and the prototype chain is not considered in case the property name is not found in some of them. Instead, a new property with that name is created, if this is the case. Again, if the property names that resulted from the `index` operand are general (like `UInt32Str`), the values of the fields `default_other` and `default_index` are updated. The update of an object may be strong or not, depending on whether it is a singleton or summary object. The not strong updates just join the existing values with the new ones, while the strong updates overwrite them. Properties with the `RO` atribute are never updated, while properties with the `Maybe_RO` attribute are never strongly updated.

`Delete_variable_block (temp, var), Delete_property_block (temp, base, index)` The transfer function of the `Delete_property_block` label is similar with the one for the `Write_property_block` label, but instead of updating a property's value it deletes the property from the object. If a property is not found the prototype chain is not considered, and no action is done. Properties of singleton objects may be completely removed, while properties of summary objects can only be flagged as `Maybe_absent`. Properties with the `Not_config` atribute are never removed, while properties with the `Maybe_config` attribute can only be flagged as `Maybe_absent`. The transfer function of the `Delete_variable_block` label first searches the scope chain(s) for variable objects that contain the variable `var` as their property and then proceeds in the same way. Note that, declared variables have always the `Not_config` attribute as properties of variable objects, so they cannot be deleted. Finally, the `true` value is assigned to the `temp` temporary, if the deletion was permitted and the `false` value if it was denied. In case the deletion was not strong or it happened in some objects, the `Any_boolean` value is used.

`Unop_block (temp, unop, arg), Binop_block (temp, binop, arg1, arg2)` The transfer function will assign the value of the described by the label operation to the temporary with id `temp`. First, the values of the argument operands are fetched from the input state and converted as needed by the operation. Then, the abstract procedure for that operation is used to produce the result. The abstract procedures simulate the procedures defined for the respective JavaScript operations, but they operate in abstract values.

`If_block cond, Ifop_block (binop, arg1, arg2)` The transfer function will compute the condition result, in the first case by fetching the value of the `cond` operand and converting it to boolean, and in the second case by fetching the values of the argument operands and executing the coresponding abstract operation. The result is two states, for the cases the condition is true or false, which will be assigned to the outcoming `When_true` and `When_false` edges respectively. If the condition result is known to be true or false the the bottom state is used for the oppositely labeled edge. This way, the flow of this edge is effectively discarded. Moreover, if the condition result is `Any_boolean`, the condition itself may be used to constraint some variable values in a different way in the two resulting states.

`EnumProperty_block (temp, arg)` This label is needed in the representation of `for/in` statements. The value of the `arg` operand is fetched from the input state and is converted to an object value. The transfer function assigns the union of property names of the objects, resulted from the `arg` operand, to the temporary with id `temp`. The property names of the objects of the prototype chain are also included. However, properties with the `Not_enum` attribute are not included. The resulting state is given to the outcoming `When_true` edge. The state that is given to the outcoming `When_false` edge has the `undefined` value assigned to the `temp` temporary. This flow represents the exit from the `for/in` loop.

`Call_block (temp, func, args),`

`Call_method_block (temp, base, index, args),`

`New_block (temp, func, args)` The transfer function will assign the return value of the call to the temporary with id `temp`. First, the operands in the `args` list are fetched from the input state. Second, the possible callee functions are identified either by fetching the value of the `func` operand from the input state or by performing a read as described above. The details and mechanics for the function call and return are given in the paragraphs 2.1.2 and 3.3.3. The `this` value is the global object in the first case, the objects resulting from the `base` operand in the second and a newly created object in the third.

`Throw_block arg` The transfer function creates only an exception state (the outputed normal state is the bottom state) in order to begin the porpagation of the value of the `arg` operand, which is the thrown exception.

`Enter_catch_block name, Exit_catch_block` The transfer function for the `Enter_catch_block` label will add a new object in the scope chain(s) of the current execution context(s). The object will have a property with name `name` and with value the exception that was propagated to the node. The transfer function for the `Exit_catch_block` label will remove that object from the scope chain(s).

`Enter_with_block arg, Exit_with_block` First, the value of the `arg` operand is fetched from the input state and it is converted to an object value, that may include one or more objects. The transfer function for the `Enter_with_block` label will add every new object in the scope chain(s) of the current execution context(s). Therefore a new set of execution contexts will be created for the output state. The transfer function for the `Exit_with_block` label will remove the front object from the scope chain(s) of the current execution context(s).

3.3.6 Termination

We will not strictly prove the termination of our analysis. Instead we will give some intuitive reasons, why our analysis should always terminate.

- The number of different allocation sites is bounded by the number of points of new object creation in the input program.
- The number of different property names is bounded by the number of property names used in the input program.
- The size of number and string unions (see Section 3.2.1) is limited by a constant value. When a union gets bigger than allowed, it is extended to a more general type.
- The number of separate states for every node and edge due to flow-sensitive analysis is also bounded. As we saw, if there is need for more different states, the analysis of the function becomes flow-insensitive, and the states are joined together.

3.4 Examples

In this section, we will provide some examples to demonstrate the use and the results of the analysis.

3.4.1 Recency Abstraction Example

In the example of Listing 3.17, the function `make_list` creates a list of objects with an integer value property. Since `make_list` is recursive, more than one objects (actually 4 of them) will be created at the same point (line 3). Using recency abstraction, the analysis creates two abstract objects: the singleton, that contains only the number value `1`, which indeed is the value of the most recent object created at line 3, and the summary, that contains any of the number values `2`, `3` and `4`. The function returns the singleton object (line 9) and therefore, we are able to strongly update its `value` (line 10). The results of the analysis are shown in Listing 3.18.

```
1  function make_list(i, l) {
2    if (i === 0) return l;
3    var cell = {};
4    cell.value = i;
5    cell.next = l;
6    return make_list(i-1,cell);
7  }
8
9  var list = make_list(4);
10 list.value = 42;
```

Listing 3.17: Example of use: recency abstraction

```
> ./jsanalyzer -v exampleRecency.js
Estimated Types
Script variables:
Normally
    list:{objects: {Singleton 43,}}
objects:
  ...
Summary 43 ->
    properties:
    next->[value: {undefined, objects: {Summary 43,}}]
    value->[value: {numbers: {2,3,4,}}]
    prototype: {Object.prototype,}
    default value: {}
    default index: {}
    default other: {}
    scope: none
Singleton 43 ->
    properties:
    next->[value: {objects: {Summary 43,}}]
    value->[value: {numbers: 42}]
    prototype: {Object.prototype,}
    default value: {}
    default index: {}
    default other: {}
    scope: none
  ...
```

Listing 3.18: Analysis result: recency abstraction

3.4.2 Error Example

In this example, we show a common error in JavaScript programs. The Listing 3.19 contains the definition of a variable `s` and its initialization to a string value (line 1). Then, we attempt to set a property of `s` (line 2). This a valid JavaScript expression, and what happens is that, the value of `s` is converted to an object and the property of that object is set. This is shown in the analysis result at Listing 3.20: the Singleton 41 object is the wrapper object, that was created by the conversion, and its `append` property is set to the correct function literal. However, when we use the variable `s` as an object again is line 3, a new conversion takes places and a new object is created, the Singleton 42 object of the analysis result. The latter does not have an `append` property, as the programmer might think. As a result, the property access returns the `undefined` value, and the attempt to call this value as a function raises a type error. The execution of the program ends and the `g` variable remains with its initial `undefined` value.

```
1 var s = ”hello!”;
2 s.append = function (s1) { return this+s1; };
3 var g = s.append(”hi!”);
```

Listing 3.19: Example of use: error

```
> ./jsanalyzer -v exampleError.js
Estimated Types
Script variables:
When exception ({objects: {TypeErrorInstance,}})
    s:{strings: ”hello!”}
    g:{undefined}
objects:
Singleton 41 ->
    properties:
    append->[value: {objects: {Anonymous Function Literal (line 2),}}]
    length->[value: {numbers: 6}, readonly, not_configurable, not_enumerable]
    prototype: {String.prototype,}
    default value: {strings: ”hello!”}
    default index: {}
    default other: {}
    scope: none
Singleton 42 ->
    properties:
    length->[value: {numbers: 6}, readonly, not_configurable, not_enumerable]
    prototype: {String.prototype,}
    default value: {strings: ”hello!”}
    default index: {}
    default other: {}
    scope: none
```

Listing 3.20: Analysis result: error

3.4.3 Objects and Prototypes Example

The Listing 3.21 shows an example of simulating classes and inheritance in JavaScript. As stated before, the constructor functions, when used with the `new` keyword, not only initialize the newly created object, but also they set its prototype object to be the value of their `prototype` property. Therefore, one can use the `prototype` property object of a constructor function, in order to store functions, which can be used as methods by the objects created with the constructor function. This is the case with the function literal assignments of lines 6 and 21. Also, by setting the `prototype` property of a constructor function to be an object created with another constructor function, one can simulate inheritance, since through the prototype chain, the properties of the objects created with the latter constructor are available to the objects created with the former constructor. In our example, the property `move` that is available to objects created with the constructor `Shape` is also available to objects created with the constructor `Circle`, because the prototype object of the latter is a Shape object (see line 17), the prototype object of which contains the `move` property.

The analysis result for this example is shown in Listing 3.22. The `for/in` loop visits all the available properties of the `c` object. This object is constructed with the `Circle` constructor. The `p` variable accumulates all the possible property names, while the `ans` variable is a union of the values of these properties. The constructor objects and their `prototype` property objects are also shown. We can see that a prototype chain has been formed: the `c` object (Singleton 52) has the `Circle.prototype` object (Singleton 49) as its prototype object, and that has the `Shape.prototype` (Singleton 2) object as its prototype object. As always, the prototype chain ends at the `Object.prototype` object.

```
function Shape(x,y) {
  this.x = x;
  this.y = y;
}

Shape.prototype.move = function (dx,dy) {
  this.x += dx; this.y += dy;
};

function Circle(x,y,r) {
  this.s = Shape;
  this.s(x,y);
  delete this.s;
  this.radius = r;
}

Circle.prototype = new Shape;
delete Circle.prototype.x;
delete Circle.prototype.y;
Circle.prototype.constructor = Circle;
Circle.prototype.area = function () {
  return 3.14*this.radius*this.radius;
};

var c = new Circle(7,8,12);
var ans;

c.move(2,2);
for (var p in c)
  ans = c[p];
```

Listing 3.21: Example of use: objects and prototypes

```
> ./jsanalyzer -v exampleObjects.js
Estimated Types
Script variables:
Normally
    c:{objects: {Singleton 52,}}
    ans:{numbers: {9,10,12,}, undefined, objects: {
        Anonymous Function Literal (line 6),Function Circle (line 10),
        Anonymous Function Literal (line 21),}}
    p:{strings: {"area","constructor","move","radius","x","y",}, undefined}
objects:
Function Shape (line 1) ->
    properties:
    length->[value: {numbers: 2}, readonly, not_configurable, not_enumerable]
    prototype->[value: {objects: {Singleton 2,}}, not_configurable]
    prototype: {Function.prototype,}
    default value: {}
    default index: {}
    default other: {}
    scope: {[global,],}
Singleton 2 ->
    properties:
    constructor->[value: {objects: {Function Shape (line 1),}}, not_enumerable]
    move->[value: {objects: {Anonymous Function Literal (line 6),}}]
    prototype: {Object.prototype,}
    default value: {}
    default index: {}
    default other: {}
    scope: none
Function Circle (line 10) ->
    properties:
    length->[value: {numbers: 3}, readonly, not_configurable, not_enumerable]
    prototype->[value: {objects: {Singleton 49,}}, not_configurable]
    prototype: {Function.prototype,}
    default value: {}
    default index: {}
    default other: {}
    scope: {[global,],}
Singleton 49 ->
    properties:
    area->[value: {objects: {Anonymous Function Literal (line 21),}}]
    constructor->[value: {objects: {Function Circle (line 10),}}]
    prototype: {Singleton 2,}
    default value: {}
    default index: {}
    default other: {}
    scope: none
Singleton 52 ->
    properties:
    radius->[value: {numbers: 12}]
    x->[value: {numbers: 9}]
    y->[value: {numbers: 10}]
    prototype: {Singleton 49,}
    default value: {}
    default index: {}
    default other: {}
    scope: none
  ...
```

Listing 3.22: Analysis result: objects and prototypes

Chapter 4

Related and Future Work

4.1 Related Work

As far as JavaScript is concerned, Thiemann and Heidegger have presented a type system that uses recency types and demotions [12, 2]. Jensen, Thiemann and Møller have designed the dataflow analysis, which is used by JsAnalyzer [6]. They have also proposed a method of lazy propagation for improving the performance of the interprocedural dataflow analysis [7]. Jang *et al.* have described a points-to analysis for JavaScript programs, which is based on a context-insensitive analysis and the collection and solution of constraints [5]. Finally, Richards *et al.* have published an empirical study about the actual usage of JavaScript's dynamic features, useful for the design and the evaluation of static analyses for the language [10].

Ruby is another dynamically typed scripting programming language, that has been attempted to be analyzed statically. In their work, Furr *et al.*, have designed a static type system and an inference algorithm for the language [1]. Their tool, Diamondback Ruby, can infer types for ruby programs or check annotated types provided by the user.

4.2 Future Work

A lot of future work can be done in the direction of improving the analysis method and its results. In its current form the analysis lacks support for some special categories of objects, like Date and RegExp objects. It also does not handle the language features for dynamically generated code. These features are widely used, thus an effective method to blend them in the analysis will significantly add to its power. Finally, some optimizations, like lazy propagation, can be implemented to improve the analysis' performance.

Furthermore, JsAnalyzer is a static analysis framework that can be used to assist various purposes. To begin with, a dataflow analysis is needed in almost every method for error dedection. In particular, this work is aimed to be used in a static analysis tool that provides sound error information, which is extremely useful for languages with dynamic type systems, like JavaScript. Also, the flow and type information produced by the analysis can be used in JavaScript interpreters for optimization purposes.

Bibliography

[1] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[2] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 200–224, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification.* ECMA (European Association for Standardizing Information and Communication Systems), third edition, 1999. `http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf`.

[4] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification.* ECMA (European Association for Standardizing Information and Communication Systems), fifth edition, 2011. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`.

[5] D. Jang and K.-M. Choe. Points-to analysis for javascript. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1930–1937, New York, NY, USA, 2009. ACM.

[6] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *Proceedings of the 17th international conference on Static analysis*, SAS'10, pages 320–339, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] OCaml Programming Language. `http://caml.inria.fr/ocaml/`.

[9] Rhino. `http://www.mozilla.org/rhino/`.

[10] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[11] SpiderMonkey JavaScript Engine. `http://developer.mozilla.org/en/SpiderMonkey`.

[12] P. Thiemann. Towards a type system for analyzing javascript programs. In *Proceedings of the 14th European conference on Programming Languages and Systems*, ESOP'05, pages 408–422, Berlin, Heidelberg, 2005. Springer-Verlag.

[13] v8 JavaScript Engine. `http://code.google.com/p/v8/`.

[14] W3C DOM. `http://www.w3.org/DOM/`.