Type Analysis for JavaScript

Simon Holm Jensen[1,⋆], Anders Møller[1,⋆,†], and Peter Thiemann[2]

[1] Aarhus University, Denmark
{simonhj,amoeller}@cs.au.dk
[2] Universität Freiburg, Germany
thiemann@informatik.uni-freiburg.de

Abstract. JavaScript is the main scripting language for Web browsers, and it is essential to modern Web applications. Programmers have started using it for writing complex applications, but there is still little tool support available during development.
We present a static program analysis infrastructure that can infer detailed and sound type information for JavaScript programs using abstract interpretation. The analysis is designed to support the full language as defined in the ECMAScript standard, including its peculiar object model and all built-in functions. The analysis results can be used to detect common programming errors – or rather, prove their absence, and for producing type information for program comprehension.
Preliminary experiments conducted on real-life JavaScript code indicate that the approach is promising regarding analysis precision on small and medium size programs, which constitute the majority of JavaScript applications. With potential for further improvement, we propose the analysis as a foundation for building tools that can aid JavaScript programmers.

1 Introduction

In 1995, Netscape announced JavaScript as an "easy-to-use object scripting language designed for creating live online applications that link together objects and resources on both clients and servers" [25]. Since then, it has become the de facto standard for client-side scripting in Web browsers but many other applications also include a JavaScript engine. This prevalence has lead developers to write large programs in a language which has been conceived for scripting, but not for programming in the large. Hence, tool support is badly needed to help debug and maintain these programs.

The development of sound programming tools that go beyond checking mere syntactic properties requires some sort of program analysis. In particular, type analysis is crucial to catch representation errors, which e.g. confuse numbers with strings or booleans with functions, early in the development process. Type analysis is a valuable tool to a programmer because it rules out this class of programming errors entirely.

⋆ Supported by The Danish Research Council for Technology and Production, grant no. 274-07-0488. †Corresponding author.

Applying type analysis to JavaScript is a subtle business because, like most other scripting languages, JavaScript has a weak, dynamic typing discipline which resolves many representation mismatches by silent type conversions. As JavaScript supports objects, first-class functions, and exceptions, tracking the flow of data and control is nontrivial. Moreover, JavaScript's peculiarities present a number of challenges that set it apart from most other programming languages:

- JavaScript is an object-based language that uses prototype objects to model inheritance. As virtually all predefined operations are accessed via prototype objects, it is imperative that the analysis models these objects precisely.
- Objects are mappings from strings (property names) to values. In general, properties can be added and removed during execution and property names may be dynamically computed.
- Undefined results, such as accessing a non-existing property of an object, are represented by a particular value `undefined`, but there is a subtle distinction between an object that lacks a property and an object that has the property set to `undefined`.
- Values are freely converted from one type to another type with few exceptions. In fact, there are only a few cases where no automatic conversion applies: the values `null` and `undefined` cannot be converted to objects and only function values can be invoked as functions. Some of the automatic conversions are non-intuitive and programmers should be aware of them.
- The language distinguishes primitive values and wrapped primitive values, which behave subtly different in certain circumstances.
- Variables can be created by simple assignments without explicit declarations, but an attempt to read an absent variable results in a runtime error. JavaScript's `with` statement breaks ordinary lexical scoping rules, so even resolving variable names is a nontrivial task.
- Object properties can have attributes, like ReadOnly. These attributes cannot be changed by programs but they must be taken into account by the analysis to maintain soundness and precision.
- Functions can be created and called with variable numbers of parameters.
- Function objects serve as first-class functions, methods, and constructors with subtly different behavior. An analysis must keep these uses apart and detect initialization patterns.
- With the `eval` function, a dynamically constructed string can be interpreted as a program fragment and executed in the current scope.
- The language includes features that prescribe certain structures (the global object, activation objects, argument objects) in the implementation of the runtime system. These structures must be modeled in an analysis to obtain sufficient precision.

This paper reports on the design and implementation of a program analyzer for the full JavaScript language. In principle, the design is an application of abstract interpretation using the monotone framework [9, 21]. However, the challenges explained above result in a complicated lattice structure that forms the basis of our analysis. Starting from a simple type lattice, the lattice has evolved in a number of steps driven by an observed lack of precision on small test cases. As the lattice includes precise singleton values, the analyzer duplicates a large amount of the functionality of a JavaScript interpreter including the implementation of predefined functions. Operating efficiently on the elements of the lattice is another non-trivial challenge.

The analyzer is targeted at hand-written programs consisting of a few thousand lines of code. We conjecture that most existing JavaScript programs fit into this category.

One key requirement of the analysis is *soundness*. Although several recent bug finding tools for other languages sacrifice soundness to obtain fewer false positives [5, 12], soundness enables our analysis to guarantee the absence of certain errors. Moreover, the analysis is *fully automatic*. It neither requires program annotations nor formal specifications.

While some programming errors result in exceptions being thrown, other errors are masked by dynamic type conversion and `undefined` values. Some of these conversions appear unintuitive in isolation but make sense in certain circumstances and some programmers may deliberately exploit such behavior, so there is no clear-cut definition of what constitutes an "error". Nevertheless, we choose to draw the programmer's attention to such potential errors. These situations include

1. invoking a non-function value (e.g. `undefined`) as a function,
2. reading an absent variable,
3. accessing a property of `null` or `undefined`,
4. reading an absent property of an object,
5. writing to variables or object properties that are never read,
6. implicitly converting a primitive value to an object (as an example, the primitive value `false` may be converted into a `Boolean` object, and later converting that back to a primitive value results in `true`, which surprises many JavaScript programmers),
7. implicitly converting `undefined` to a number (which yields `NaN` that often triggers undesired behavior in arithmetic operations),
8. calling a function object both as a function and as a constructor (i.e. perhaps forgetting `new`) or passing function parameters with varying types (e.g. at one place passing a number and another place passing a string or no value),
9. calling a built-in function with an invalid number of parameters (which may result in runtime errors, unlike the situation for user defined functions) or with a parameter of an unexpected type (e.g. the second parameter to the `apply` function must be an array).

The first three on this list cause runtime errors (exceptions) if the operation in concern is ever executed, so these warnings have a higher priority than the others. In many situations, the analysis can report a warning as a definite error rather than a potential error. For example, the analysis may detect that a property read operation will always result in `undefined` because the given property is never present, in which case that specific warning gets high priority. As the analysis is sound, the absence of errors and warnings guarantees that the operations concerned will not fail. The analysis can also detect dead code.

The following tiny but convoluted program shows one way of using JavaScript's prototype mechanism to model inheritance:

```
function Person(n) {
  this.setName(n);
  Person.prototype.count++;
}
Person.prototype.count = 0;
Person.prototype.setName = function(n) { this.name = n; }
function Student(n,s) {
  this.b = Person;
  this.b(n);
  delete this.b;
  this.studentid = s.toString();
}
Student.prototype = new Person;
```

The code defines two "classes" with constructors `Person` and `Student`. `Person` has a static field `count` and a method `setName`. `Student` inherits `count` and `setName` and defines an additional `studentid` field. The definition and deletion of `b` in `Student` invokes the super class constructor `Person`. A small test case illustrates its behavior:

```
var t = 100026.0;
var x = new Student("Joe Average", t++);
var y = new Student("John Doe", t);
y.setName("John Q. Doe");
assert(x.name === "Joe Average");
assert(y.name === "John Q. Doe");
assert(y.studentid === "100027");
assert(x.count == 3);
```

Even for a tiny program like this, many things could go wrong – keeping the different errors discussed above in mind – but our analysis is able to prove that none of the errors can occur here. Due to the forgiving nature of JavaScript, errors may surface only as mysterious `undefined` values. Simple errors, like misspelling `prototype` or `name` in just a single place or writing `toString` instead of `toString()`, are detected by the static type analysis instead of causing failure at runtime. The warning messages being produced by the analysis can help the programmer not only to detect errors early but also to pinpoint their cause.

Contributions

This work is the first step towards a full-blown JavaScript program analyzer, which can be incorporated into an IDE to supply on-the-fly error detection as well as support for auto-completion and documentation hints. It focuses on JavaScript version 1.5, corresponding to ECMAScript 3rd edition [11], which is currently the most widely used variant of the language and which is a subset of the upcoming revision of the JavaScript language.

In summary, the contributions of this paper are the following:

– We define a type analysis for JavaScript based on abstract interpretation [9]. Its main contribution is the design of an intricate lattice structure that fits with the peculiarities of the language. We design the analysis building on existing techniques, in particular recency abstraction [3].
– We describe our prototype implementation of the analysis, which covers the full JavaScript language as specified in the ECMAScript standard [11], and we report on preliminary experiments on real-life benchmark programs and measure the effectiveness of the various analysis techniques being used.
– We identify opportunities for further improvements of precision and speed of the analysis, and we discuss the potential for additional applications of the analysis technique.

Additional information about the project is available online at

`http://www.brics.dk/TAJS`

2 Related Work

The present work builds on a large body of work and experience in abstract interpretation and draws inspiration from work on soft typing and dynamic typing. The main novelty consists of the way it combines known techniques, leading to the construction of the first full-scale implementation of a high precision program analyzer for JavaScript. It thus forms the basis to further investigate the applicability of techniques in this new domain.

Dolby [10] explains the need for program analysis for scripting languages to support the interactive completion and error spotting facilities of an IDE. He sketches the design of the WALA framework [13], which is an adaptable program analysis framework suitable for a range of languages, including Java, JavaScript, Python, and PHP. While our first prototype was built on parts of the WALA framework, we found that the idiosyncrasies of the JavaScript language required more radical changes than were anticipated in WALA's design.

Eclipse includes JSDT [7], which mainly focuses on providing instantaneous documentation and provides many shortcuts for common programming and documentation patterns as well as some refactoring operations. It also features some unspecified kind of prototype-aware flow analysis to predict object types and thus enable primitive completion of property names. JSEclipse [1] is another Eclipse plugin, which includes built-in knowledge about some popular JavaScript frameworks and uses the Rhino JavaScript engine to run parts of the code to improve support for code completion. Neither of these plugins can generate warnings for unintended conversions or other errors discussed above.

Program analysis for scripting languages has evolved from earlier work on type analysis for dynamically typed languages like Scheme and Smalltalk [6, 31, 16]. These works have clarified the need for a type structure involving union types and recursive types. They issue warnings and insert dynamic tests in programs that cannot be type checked. MrSpidey [14] is a flow-based implementation of these ideas with visual feedback about the location of the checks in a programming environment. In contrast, our analysis only reports warnings because the usefulness of checks is not clear in a weakly typed setting.

Thiemann's typing framework for JavaScript programs [30] has inspired the design of the abstract domain for the present work. That work concentrates on the design and soundness proof, but does not present a typing algorithm. In later work, Heidegger and Thiemann [17] propose a recency-based type system for a core language of JavaScript, present its soundness proof, sketch an inference algorithm, and argue the usefulness of this concept.

Anderson and others [2] present a type system with an inference algorithm for a primitive subset of JavaScript based on a notion of definite presence and potential absence of properties in objects. Their system does not model type change and the transition between presence and absence of a property is harder to predict than in a recency-based system.

Furr and others [15] have developed a typed dialect of Ruby, a scripting language with features very similar to JavaScript. Their approach requires the programmer to supply type annotations to library functions. Then they employ standard constraint solving techniques to infer types of user-defined functions. There is support for universal types and intersection types (to model overloading), but these types can only be declared, not inferred. They aim for simplicity in favor of precision also to keep the type language manageable, whereas our design aims for precision. Their paper contains a good overview of further, more pragmatic approaches to typing for scripting languages like Ruby and Python.

Similar techniques have been applied to the Erlang language by Marlow and Wadler [24] as well as by Nyström [27]. These ideas have been extended and implemented in a practical tool by Lindahl and Sagonas [23]. Their work builds on success typings, a notion which seems closely related to abstract interpretation.

One program analysis that has been developed particularly for JavaScript is points-to analysis [20]. The goal of that analysis is not program understanding, but enabling program optimization. The paper demonstrates that the results from the analysis enable partial redundancy elimination. The analysis is flow and context insensitive and it is limited to a small first-order core language. In contrast, our analysis framework deals with the entire language and performs points-to analysis as part of the type analysis. As our analysis is flow and context sensitive, it yields more precise results than the dedicated points-to analysis.

Balakrishnan and Reps [3] were first to propose the notion of recency in abstract interpretation. They use it to create a sound points-to analysis with sufficient precision to resolve the majority of virtual method calls in compiled C++ code. Like ourselves, they note that context sensitivity is indispensable in the presence of recency abstraction. However, the rest of their framework is substantially different as it is targeted to analyzing binary code. Its value representation is based on a stride domain and the interprocedural part uses a standard k-limited call-chain abstraction.

Shape analysis [28] is yet more powerful than recency abstraction. For example, it can recover strongly updatable abstractions for list elements from a summary description of a list data structure. This capability is beyond recency abstraction. However, the superior precision of shape analysis requires a much more resource-intensive implementation.

Finally, our analysis uses abstract garbage collection. This notion has been investigated in depth in a polyvariant setting by Might and Shivers [26], who attribute its origin to Jagannathan and others [19]. They, as well as Balakrishnan and Reps [3], also propose abstract counting which is not integrated in our work as the pay-off is not yet clear.

3 Flow Graphs for JavaScript

The analysis represents a JavaScript program as a flow graph, in which each node contains an instruction and each edge represents potential control flow between instructions in the program. The graph has a designated program entry node corresponding to the first instruction of the global code in the program. Instructions refer to *temporary variables*, which have no counterpart in JavaScript, but which are introduced by the analyzer when breaking down composite expressions and statements to instructions. The nodes can have different kinds:

declare-variable$[x]$: declares a program variable named x with value `undefined`.

read-variable$[x, v]$: reads the value of a program variable named x into a temporary variable v.

write-variable$[v, x]$: writes the value of a temporary variable v into a program variable named x.

constant$[c, v]$: assigns a constant value c to the temporary variable v.

read-property$[v_1, v_2, v_3]$: performs an object property lookup, where v_1 holds the base object, v_2 holds the property name, and v_3 gets the resulting value.

write-property$[v_1, v_2, v_3]$: performs an object property write, where v_1 holds the base object, v_2 holds the property name, and v_3 holds the value to be written.

delete-property$[v_1, v_2, v_3]$: deletes an object property, where v_1 holds the base object, v_2 holds the property name, and v_3 gets the resulting value.

if$[v]$: represents conditional flow for e.g. `if` and `while` statements.

entry$[f, x_1, \ldots, x_n]$, exit, and exit-exc: used for marking the unique entry and exit (normal/exceptional) of a function body. Here, f is the (optional) function name, and $x_1, \ldots, x_n$ are formal parameters.

call$[w, v_0, \ldots, v_n]$, construct$[w, v_0, \ldots, v_n]$, and after-call$[v]$: A function call is represented by a pair of a call node and an after-call node. For a call node, w holds the function value and $v_0, \ldots, v_n$ hold the values of `this` and the parameters. An after-call node is returned to after the call and contains a single variable for the returned value. The construct nodes are similar to call nodes and are used for `new` expressions.

return$[v]$: a function return.

throw$[v]$ and catch$[x]$: represent `throw` statements and entries of `catch` blocks.

<*op*>$[v_1, v_2]$ and <*op*>$[v_1, v_2, v_3]$: represent unary and binary operators, where the result is stored in v_2 or v_3, respectively.

This instruction set is reminiscent of the bytecode language used in some interpreters [18] but tailored to program analysis. Due to the limited space, we here omit the instructions related to `for-in` and `with` blocks and settle for this informal description of the central instructions. They closely correspond to the ECMAScript specification – for example, read-property is essentially the [[Get]] operation from the specification.

We distinguish between different kinds of edges. *Ordinary* edges correspond to intra-procedural control flow. These edges may be labeled to distinguish branches at if nodes. Each node that may raise an exception has an *exception* edge to a catch node or an exit-exc node. Finally, *call* and *return* edges describe flow from call or construct nodes to entry nodes and from exit nodes to after-call nodes.

All nodes as well as ordinary edges and exception edges are created before the fixpoint iteration starts, whereas the call and return edges are added on the fly when data flow is discovered, as explained in Section 4.

4 The Analysis Lattice and Transfer Functions

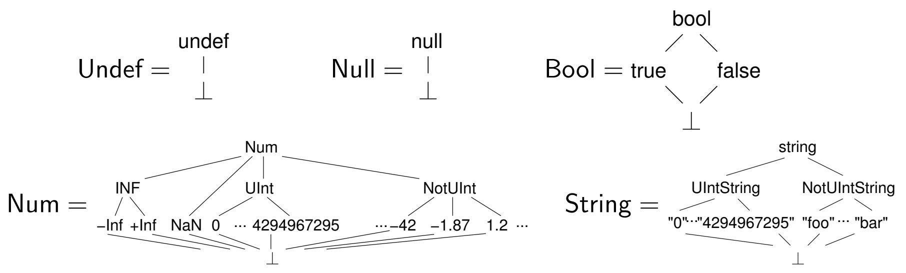
The classical approach of abstract interpretation [9] and the monotone framework [21] requires a lattice of abstract states. Our lattice structure is similar to a lattice used for constant propagation with JavaScript's type structure on top. Numbers and strings are further refined to recognize array indices. For objects, the analysis performs a context-sensitive flow analysis that discovers points-to information.

For a given flow graph, we let N denote the set of nodes, T is the set of temporary variables, and L is the set of *object labels* corresponding to the possible allocation sites (including construct nodes, constant nodes for function declarations, and objects defined in the standard library).

Abstract values are described by the lattice Value:

$$\mathsf{Value} = \mathsf{Undef} \times \mathsf{Null} \times \mathsf{Bool} \times \mathsf{Num} \times \mathsf{String} \times \mathcal{P}(L)$$

The components of Value describe the different types of values.



For example, the abstract value $(\bot, \mathsf{null}, \bot, \bot, \mathtt{baz}, \emptyset)$ describes a concrete value that is either `null` or the string "`baz`", and $(\mathsf{undef}, \bot, \bot, \bot, \bot, \{\ell_{42}, \ell_{87}\})$ describes a value that is `undefined` or an object originating from ℓ_{42} or ℓ_{87}.

Objects are modeled as follows:

$$\mathsf{Obj} = (P \hookrightarrow \mathsf{Value} \times \mathsf{Absent} \times \mathsf{Attributes} \times \mathsf{Modified}) \times \mathcal{P}(\mathsf{ScopeChain})$$

Here, P is the infinite set of property names (i.e. all strings). The partial map provides an abstract value for every possible property name. There are four special property names: [[Prototype]], [[Value]], *default_index*, and *default_other*. The former two correspond to the internal properties used by ECMAScript; *default_index* and *default_other* are always in the domain of the map and provide an abstract value for all property names that are not in the domain of the map (hence the map is effectively total): *default_index* covers property names that match UIntString (array indices), and *default_other* covers all other strings. This distinction is crucial when analyzing programs involving array operations. Section 4.3 explains the ScopeChain component, which models the special internal property [[Scope]].

Each value stored in an object has additional components. Absent models potentially absent properties, Modified is related to interprocedural analysis as explained in Section 4.3, and Attributes models the property attributes ReadOnly, DontDelete, and DontEnum.

$$\mathsf{Absent} = \begin{tikzcd}[row sep=small] \mathsf{absent} \arrow[d, no head] \\ \bot \end{tikzcd} \qquad \mathsf{Modified} = \begin{tikzcd}[row sep=small] \mathsf{modified} \arrow[d, no head] \\ \bot \end{tikzcd}$$

$$\mathsf{Attributes} = \mathsf{ReadOnly} \times \mathsf{DontDelete} \times \mathsf{DontEnum}$$

$$\mathsf{ReadOnly} = \begin{tikzcd}[row sep=small, column sep=tiny] & \top \arrow[dl, no head] \arrow[dr, no head] & \\ \mathsf{RO} \arrow[dr, no head] & & \mathsf{notRO} \arrow[dl, no head] \\ & \bot & \end{tikzcd} \quad \mathsf{DontDelete} = \begin{tikzcd}[row sep=small, column sep=tiny] & \top \arrow[dl, no head] \arrow[dr, no head] & \\ \mathsf{DD} \arrow[dr, no head] & & \mathsf{notDD} \arrow[dl, no head] \\ & \bot & \end{tikzcd} \quad \mathsf{DontEnum} = \begin{tikzcd}[row sep=small, column sep=tiny] & \top \arrow[dl, no head] \arrow[dr, no head] & \\ \mathsf{DE} \arrow[dr, no head] & & \mathsf{notDE} \arrow[dl, no head] \\ & \bot & \end{tikzcd}$$

An abstract state consists of an abstract store, which is a partial map from object labels to abstract objects, together with an abstract stack:

$$\mathsf{State} = (L \hookrightarrow \mathsf{Obj}) \times \mathsf{Stack} \times \mathcal{P}(L) \times \mathcal{P}(L)$$

The last two object label sets in State are explained in Section 4.3.

The stack is modeled as follows:

$$\begin{aligned} \mathsf{Stack} &= (T \rightarrow \mathsf{Value}) \times \mathcal{P}(\mathsf{ExecutionContext}) \times \mathcal{P}(L) \\ \mathsf{ExecutionContext} &= \mathsf{ScopeChain} \times L \times L \\ \mathsf{ScopeChain} &= L^* \end{aligned}$$

The first component of Stack provides values for the temporary variables. The $\mathcal{P}$(ExecutionContext) component models the top-most execution context[3] and the $\mathcal{P}(L)$ component contains object labels of all references in the stack. An execution context contains a scope chain, which is here a sequence of object

[3] The ECMAScript standard [11] calls a stack frame an *execution context* and also defines the terms *scope chain* and *variable object*.

labels, together with two additional object labels that identify the variable object and the `this` object.

Finally, we define the analysis lattice, which assigns a set of abstract states to each node (corresponding to the program points *before* the nodes):

$$\mathsf{AnalysisLattice} = V \times N \ \rightarrow \ \mathsf{State}$$

V is the set of version names of abstract states for implementing context sensitivity. As a simple heuristic, we currently keep two abstract states separate if they have different values for `this`, which we model by $V = \mathcal{P}(L)$.

The lattice order is defined as follows: For the components of Value, the Hasse diagrams define the lattice order for each component. All maps and products are ordered pointwise, and power sets are ordered by subset inclusion – except the last $\mathcal{P}(L)$ component of State, which uses $\supseteq$ instead of $\subseteq$ (see Section 4.3).

These definitions are the culmination of tedious twiddling and experimentation. Note, for example, that for two abstract stores σ_1 and σ_2 where $\sigma_1(\ell)$ is undefined and $\sigma_2(\ell)$ is defined (i.e. the object ℓ is absent in the former and present in the latter), the join simply takes the content of ℓ from σ_2, i.e. $(\sigma_1 \sqcup \sigma_2)(\ell) = \sigma_2(\ell)$, as desired. Also, for every abstract store σ and every ℓ where $\sigma(\ell) = (\omega, s)$ is defined, we have absent set in $\omega(\mathit{default\_index})$ and in $\omega(\mathit{default\_other})$ to reflect the fact that in every object, *some* properties are absent. Thereby, joining two stores where an object ℓ is present in both but some property p is only present in one (and mapped to the bottom Value in the other) results in a store where ℓ is present and p is marked as absent (meaning that it is *maybe* absent).

The analysis proceeds by fixpoint iteration, as in the classical monotone framework, using the transfer functions described in Section 4.1. The initial abstract state for the program entry node consists of 161 abstract objects (mostly function objects) defined in the standard library.

We omit a formal description of the abstraction/concretization relation between the ECMAScript specification and this abstract interpretation lattice. However, we note that during fixpoint iteration, an abstract state never has dangling references (i.e. in every abstract state σ, every object label ℓ that appears anywhere within σ is always in the domain of the store component of σ). With this invariant in place, it should be clear how every abstract state describes a set of concrete states.

The detailed models of object structures represented in an abstract state allows us to perform *abstract garbage collection* [26]. An object ℓ can safely be removed from the store unless ℓ is reachable from the abstract call stack. This technique may improve both performance and precision (see Section 5).

Section 5 contains an illustration of the single abstract state appearing at the final node of the example program after the fixpoint is reached.

4.1 Transfer Functions

For each kind of node n in the flow graph, a monotone transfer function maps an abstract state before n to a abstract state after n. In addition, we provide a transfer function for each predefined function in the ECMAScript standard library. Some edges (in particular, call and return edges) also carry transfer functions. As usual, the before state of node n is the join of the after states of all predecessors of n.

The transfer function for read-property$[v_{obj}, v_{prop}, v_{target}]$ serves as an illustrative example. If v_{obj} is not an object, it gets converted into one. If v_{obj} abstracts many objects, then the result is the join of reading all of them. The read operation for a single abstract object descends the prototype chain and joins the results of looking up the property until the property was definitely present in a prototype. If v_{prop} is not a specific string, then the *default_index* and *default_other* fields of the object and its prototypes are also considered. Finally, the temporary variable v_{target} is overwritten with the result; all temporaries can be strongly updated. As this example indicates, it is essential that the analysis models all aspects of the JavaScript execution model, including prototype chains and type coercions.

A special case is the transfer function for the built-in functions `eval` and `Function` that dynamically construct new program code. The analyzer cannot model such a dynamic extension of the program because the fixpoint solver requires N and L to be fixed. Hence, the analyzer issues a warning if these functions are used. This approach is likely satisfactory as these functions are mostly used in stylized ways, e.g. for JSON data, according to a study of existing JavaScript code [22].

4.2 Recency Abstraction

A common pattern in JavaScript code is creating an object with a constructor function that adds properties to the object using write-property operations. In general, an abstract object may describe multiple concrete objects, so such operations must be modeled with weak updates of the relevant abstract objects. Subsequent read-property operations then read potentially absent properties, which quickly leads to a proliferation of `undefined` values, resulting in poor analysis precision. Fortunately, a solution exists which fits perfectly with our analysis framework: *recency abstraction* [3].

In essence, each allocation site ℓ (in particular, those identified by the construct instructions) is described by *two* object labels: $\ell^@$ (called the *singleton*) always describes exactly one concrete object (if present in the domain of the store), and ℓ^* (the *summary*) describes an unknown number of concrete objects. Typically, $\ell^@$ refers to the *most recently allocated* object from ℓ (hence the name of the technique), and ℓ^* refers to older objects – however the addition of interprocedural analysis (Section 4.3) changes this slightly.

In an intra-procedural setting, this mechanism is straightforward to incorporate. Informally, the transfer function for a node n of type construct$[v]$ joins the $n^@$ object into the n^* object, redirects all pointers from $n^@$ to n^*, sets $n^@$ to an empty object, and assigns $n^@$ to v. Henceforth, v refers to a singleton abstract object, which permits strong updates.

The effect of incorporating recency abstraction on the analysis precision is substantial, as shown in Section 5.

4.3 Interprocedural Analysis

Function calls have a remarkably complicated semantics in JavaScript, but each step can be modeled precisely with our lattice definition. The transfer function for a call node n, $\mathsf{call}[w, v_0, \ldots]$, extracts all function objects from w and then, as a side-effect, adds call edges to the entry nodes of these functions and return edges from their exit nodes back to the after-call node n' of n. To handle exception flow, return edges are also added from the exit-exc nodes to n'_{exc}, where n' has an exception edge to n'_{exc}. The call edge transfer function models parameter passing. It also models the new execution context being pushed onto the call stack. The base object, v_0, is used for setting `this` and the scope chain of the new execution context (which is why we need $\mathcal{P}(\mathsf{ScopeChain})$ in Obj).

A classical challenge in interprocedural analysis is to avoid flow through infeasible paths when a function is called from several sites [29]. Ignoring this effect may lead to a considerable loss of precision. We use the Modified component of Obj to keep track of object properties that may have been modified since the current function was entered. For an abstract state σ_m at an exit node m with a return edge to an after-call node n', which belongs to a call node n, the edge transfer function checks whether the definitely non-modified parts of σ_m are inconsistent with σ_n, in which case it can safely discard the flow. (A given object property that is non-modified in σ_m is *consistent* with σ_n if its abstract value according to σ_n is less than or equal to its value according to σ_m.) If consistent, the transfer function replaces all non-modified parts of σ_m by the corresponding potentially more precise information from σ_n, together with the abstract stack. When propagating this flow along return edges, we must take into account the use of recency abstraction to "undo" the shuffling of singleton and summary objects. To this end, two sets of object labels are part of State to keep track of those object labels that are definitely/maybe summarized since entering the current function.

4.4 Termination of the Analysis

The usual termination requirement that the lattice should have finite height does not apply here, now even for a fixed program. We informally argue that the analysis nevertheless always terminates by the following observations: (1) The length of the ScopeChain object label sequences is always bounded by the lexical nesting depth of the program being analyzed. (2) The number of abstract states maintained for each node is solely determined by the choice of context sensitivity criteria. The simple heuristic proposed in Section 4 ensure the sizes of these sets to be bounded for any program. (3) The partial map in Obj has a potentially unbounded domain. However, at any point during fixpoint iteration a property name p can only occur in the domain if it was put in by a write-variable or write-property instruction. The property name for such an instruction comes from a temporary variable whose value is drawn from Value and coerced to String. In case that value is not a constant string, the use of *default_index* and *default_other* ensures that the domain is unmodified, and there are clearly only finitely many nodes that contain such an instruction. Together, these observations ensure that a fixpoint will be reached for any input program. The theoretical worst case complexity is obviously high, because of the complex analysis lattice. Nevertheless, our tool analyzes sizable programs within minutes, as shown in the next section.

5 Experiments

Our prototype is implemented on top of the JavaScript parser from Rhino [4] with around 17,000 lines of Java code. For testing that the prototype behaves as expected on the full JavaScript language, we have collected a corpus of more than 150 programs. These test programs are mostly in the range 5–50 lines of code and include 28 example programs[4] from Anderson et al. [2].

For the Anderson programs, our analysis detects all errors without spurious warnings and provides type information consistent with that of Anderson [2]. Our own programs were written to exercise various parts of the system and to provoke certain error messages, so it is not surprising that the analysis handles these well.

Running the analysis on the example program from Section 1 results in two warnings. First, the analysis correctly detects that the expression `s.toString()` involves a coercion from a primitive type to an object (which was deliberate by the programmer, in this case). Second, the analysis is able to prove that `y.studentid` is a string after the call to `y.setName`, but not that the string is a particular string, which results in a warning at the second `assert` statement. The reason is that `setName` is called twice on the same object with different strings (once through the constructor and once directly). A stronger heuristic for context sensitivity might resolve this issue.

Figure 1 shows the abstract state for the final program point of the example program, as obtained by running the prototype implementation. Each box describes an abstract object. For this simple program, each of them is a singleton (see Section 4.2). Edges correspond to references. For obvious reasons, only the used parts of the standard library are included in the illustration. The activation objects that are used during execution of the function calls have been removed by the abstract garbage collection. GLOBAL describes the global object, which also acts as execution context for the top-level code. OBJECT_PROTOTYPE and FUNCTION_PROTO model the prototype objects of the central built-in objects `Object` and `Function`, respectively. F_Person, F_Student, and F_0 correspond to the three functions defined in the program, and F_Person_PROTO, F_Student_PROTO, and F_0_PROTO are their prototype objects. Finally, L0 and L1 describe the two `Student` objects being created. The special property names [[Prototype]], [[Scope]], and [[Value]] are the so-called internal properties. For an

[4] `http://www.doc.ic.ac.uk/~cla97/js0impl/`

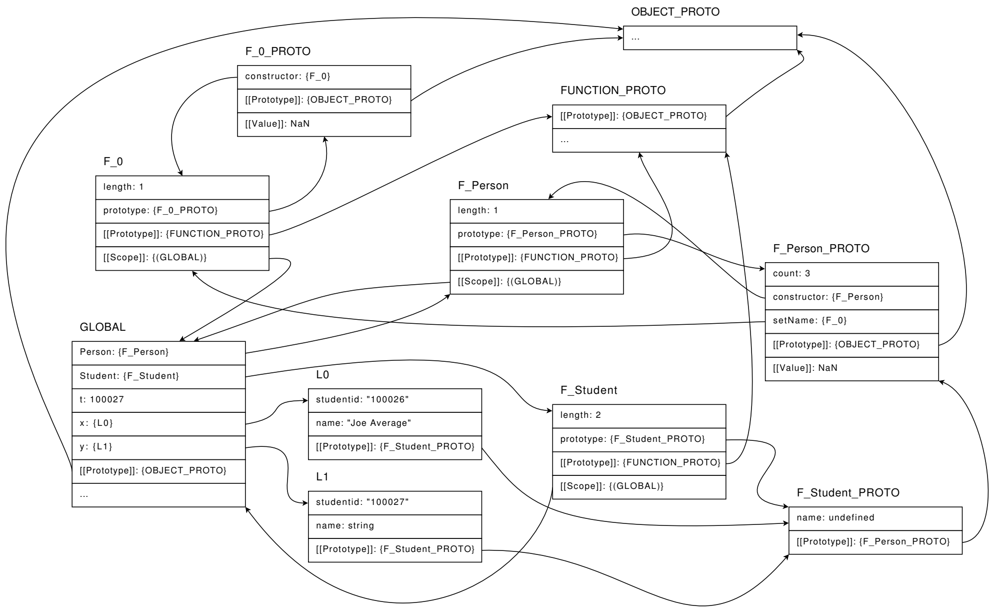


Fig. 1. Abstract state for the final program point of the example program.

example prototype chain, consider the object referred to by the variable `x` using the global object as variable object. Its prototype chain consists of L0, followed by F_Student_PROTO and F_Person_PROTO, which reflects the sequence of objects relevant for resolving the expression `x.count`. As the illustration shows, even small JavaScript programs give rise to complex object structures, which our analysis lattice captures in sufficient detail.

The tool also outputs a call graph for the program in form of the call edges that are produced during fixpoint iteration, which can be useful for program comprehension.

The Google V8 benchmark suite[5] is our main testbed to evaluate the precision of the analysis on real code. It consists of four complex, standalone JavaScript programs. Although developed for testing performance of JavaScript interpreters, they are also highly demanding subjects for a static type analysis. In addition, we use the four most complex SunSpider benchmarks[6].

Clearly we do not expect to find bugs in such thoroughly tested programs, so instead we measure precision by counting the number of operations where the analysis does *not* produce a warning (for different categories), i.e. is capable of proving that the error cannot occur at that point.

For the `richards.js` benchmark (which simulates the task dispatcher of an operating system), the analysis shows for 95% of the 58 call/construct nodes

[5] `http://v8.googlecode.com/svn/data/benchmarks/v1/`

[6] `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`

that the value being invoked is always a function (i.e. category 1 from Section 1). Moreover, it detects one location where an absent variable is read (category 2). (In this case, the absent variable is used for feature detection in browsers.) This situation *definitely* occurs if that line is ever executed, and there are no spurious warnings for this category. Next, it shows for 93% of the 259 read/write/delete-property operations that they never attempt to coerce `null` or `undefined` into an object (category 3). For 87% of the 156 read-property operations where the property name is a constant string, the property is guaranteed to be present. As a bonus, the analysis correctly reports 6 functions to be dead, i.e. unreachable from program entry. We have not yet implemented checkers for the remaining categories of errors discussed in the introduction. In most cases, the false positives appear to be caused by the lack of path sensitivity.

The numbers for the `benchpress.js` benchmark (which is a collection of smaller benchmarks running in a test harness) are also encouraging: The analysis reports that 100% of the 119 call/construct operations always succeed without coercion errors, 0 warnings are reported about reading absent variables, 89% of the 113 read/write/delete-property operations have no coercion errors, and for 100% of the 48 read-property operations that have constant property names, the property being read is always present.

The third benchmark, `delta-blue.js` (a constraint solving algorithm), is larger and apparently more challenging for type analysis: 78% of the 182 call and construct instructions are guaranteed to succeed, 8 absent variables are correctly detected (all of them are functions that are defined in browser APIs, which we do not model), 82% of 492 read/write/delete-property instructions are proved safe, and 61% of 365 read-property with constant names are shown to be safe. For this benchmark, many of the false positives would likely be eliminated by better context sensitivity heuristics.

The results for the first three V8 benchmarks and the four SunSpider benchmarks are summarized in Figure 2. For each of the categories discussed above, the table shows the ratio between precise answers obtained and the number of nodes of the relevant kind.

	lines	call / construct	variable read	property access	fixed-property read
`richards.js`	529	95%	100%	93%	87%
`benchpress.js`	463	100%	100%	89%	100%
`delta-blue.js`	853	78%	100%	82%	61%
`3d-cube.js`	342	100%	100%	92%	100%
`3d-raytrace.js`	446	99%	100%	94%	94%
`crypto-md5.js`	291	100%	100%	100%	100%
`access-nbody.js`	174	100%	100%	93%	100%

Fig. 2. Analysis precision.

The fourth (and largest) V8 benchmark, `cryptobench.js`, presently causes our prototype to run out of memory (with a limit of 512MB). For the other benchmarks, analysis time is less than 10 seconds, except `3d-raytrace.js` and `delta-blue.js` which require 30 seconds and 6 minutes, respectively. Although analysis speed and memory consumption have not been key objectives for this prototype, we naturally pursue this matter further. Most likely, the work list ordering used by the fixpoint solver can be improved.

We can disable various features in the analysis to obtain a rough measure of their effect. Disabling abstract garbage collection has little consequence on the precision of the analysis on these programs, however it is cheap to apply and it generally reduces memory consumption. Using recency abstraction is crucial: With this technique disabled, the analysis of `richards.js` can only guarantee that a constant property is present in 2 of the 156 read-property nodes (i.e. less than 2%, compared to 87% before) and the number of warnings about potential dereferences of null or undefined rises from 19 to 90. These numbers confirm our hypothesis that recency abstraction is essential to the precision of the analysis. The Modified component of State is important for some benchmarks; for example, the number of warnings about dereferences of null or undefined in `3d-raytrace.js` rises from 21 to 61 if disabling this component. Finally, we observe that context sensitivity has a significant effect on e.g. `delta-blue.js`.

6 Conclusion

Scripting languages are a sweet-spot for applying static analysis techniques: There is yet little tool support for catching errors before code deployment and the programs are often relatively small. Our type analyzer is the first sound and detailed tool of this kind for real JavaScript code. The use of the monotone framework with an elaborate lattice structure, combined with recency abstraction, results in an analysis with good precision on demanding benchmarks.

We envision an IDE for JavaScript programming with features known from strongly typed languages, such as highlighting of type-related errors and support for precise content assists and safe refactorings. This goal requires further work, especially to improve the analysis speed. Our primary objectives for the prototype have been soundness and precision, so there are plenty of opportunities for improving performance. For example, we currently use a naive work list heuristic and the representation of abstract states employs little sharing.

In further experiments, we want to investigate if there is a need for even higher precision. For example, the String component could be replaced by regular languages obtained using a variant of string analysis [8]. It may also be fruitful to tune the context sensitivity heuristic or incorporate simple path sensitivity.

Another area is the consideration of the DOM, which is heavily used by most JavaScript programs. Our work provides a basis for modeling the different DOM implementations provided by the main browsers and hence for catching browser specific programming errors. Additionally, it paves the way for analyzing code that uses libraries (Dojo, Prototype, Yahoo! UI, FBJS, jQuery, etc.). With these further challenges ahead, the work presented here constitutes a starting point for developing precise and efficient program analysis techniques and tools that can detect errors (recall the list from Section 1) and provide type information for JavaScript programs used in modern Web applications.

Acknowledgments We thank Julian Dolby and Stephen Fink for contributing the WALA framework to the research community, which helped us in the early phases of the project. Our work also benefited from inspiring discussions about JavaScript with Lars Bak and the Google Aarhus team.

References

1. Adobe. JSEclipse. `http://labs.adobe.com/technologies/jseclipse/`.
2. Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proc. 19th European Conference on Object-Oriented Programming, ECOOP '05*, volume 3586 of *LNCS*. Springer-Verlag, July 2005.
3. Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In *Proc. 13th International Static Analysis Symposium, SAS '06*, volume 4134 of *LNCS*. Springer-Verlag, August 2006.
4. Norris Boyd et al. Rhino: JavaScript for Java. `http://www.mozilla.org/rhino/`.
5. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, June 2000. John Wiley & Sons.
6. Robert Cartwright and Mike Fagan. Soft typing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '91*, June 1991.
7. Bradley Childs. JavaScript development toolkit (JSDT) features. `http://live.eclipse.org/node/569`, July 2008.
8. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.
9. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '77*, pages 238–252, 1977.
10. Julian Dolby. Using static analysis for IDE's for dynamic languages, 2005. The Eclipse Languages Symposium.
11. ECMA. ECMAScript Language Specification, 3rd edition. ECMA-262.
12. Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *4th Symposium on Operating System Design and Implementation, OSDI '00*. USENIX, October 2000.
13. Stephen Fink and Julian Dolby. WALA – The T.J. Watson Libraries for Analysis. `http://wala.sourceforge.net/`.
14. Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '96*, pages 23–32, 1996.

15. Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proc. 24th Annual ACM Symposium on Applied Computing, SAC '09, Object Oriented Programming Languages and Systems Track*, March 2009.
16. Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *Proc. 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '90*, pages 136–150, 1990.
17. Phillip Heidegger and Peter Thiemann. Recency types for dynamically-typed object-based languages. In *Proc. International Workshops on Foundations of Object-Oriented Languages, FOOL '09*, January 2009.
18. Apple Inc. Squirrelfish bytecodes. `http://webkit.org/specs/squirrelfish-3bytecode.html`.
19. Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98*, pages 329–341, 1998.
20. Dongseok Jang and Kwang-Moo Choe. Points-to analysis for JavaScript. In *Proc. 24th Annual ACM Symposium on Applied Computing, SAC '09, Programming Language Track*, March 2009.
21. John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977. Springer-Verlag.
22. Rasmus Kromann-Larsen and Rune Simonsen. Statisk analyse af JavaScript: Indledende arbejde. Master's thesis, Department of Computer Science, University of Aarhus, 2007. (In Danish).
23. Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *Proc. 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '06*, pages 167–178, 2006.
24. Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *Proc. 2nd ACM SIGPLAN International Conference on Functional Programming, ICFP '97*, pages 136–149, 1997.
25. Sun Microsystems and Netscape Inc. Netscape and Sun announce Javascript(TM), the open, cross-platform object scripting language for enterprise networks and the internet. `http://sunsite.nus.sg/hotjava/pr951204-03.html`, 1995.
26. Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *Proc. 11th ACM SIGPLAN International Conference on Functional Programming, ICFP '06*, 2006.
27. Sven-Olof Nyström. A soft-typing system for Erlang. In *Proc. 2nd ACM SIGPLAN Erlang Workshop, ERLANG '03*, pages 56–71, 2003.
28. Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
29. Micha Sharir and Amir Pnueli. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.
30. Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *Proc. Programming Languages and Systems, 14th European Symposium on Programming, ESOP '05*, April 2005.
31. Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, 1997.