Dynamic Determinacy Analysis

Max Schäfer [*]
Nanyang Technological University
schaefer@ntu.edu.sg

Manu Sridharan   Julian Dolby
IBM T.J. Watson Research Center
{msridhar, dolby}@us.ibm.com

Frank Tip
University of Waterloo
ftip@uwaterloo.ca

Abstract

We present an analysis for identifying *determinate* variables and expressions that always have the same value at a given program point. This information can be exploited by client analyses and tools to, e.g., identify dead code or specialize uses of dynamic language constructs such as `eval`, replacing them with equivalent static constructs. Our analysis is completely dynamic and only needs to observe a single execution of the program, yet the determinacy facts it infers hold for any execution. We present a formal soundness proof of the analysis for a simple imperative language, and a prototype implementation that handles full JavaScript. Finally, we report on two case studies that explored how static analysis for JavaScript could leverage the information gathered by dynamic determinacy analysis. We found that in some cases scalability of static pointer analysis was improved dramatically, and that many uses of runtime code generation could be eliminated.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

Keywords static analysis; dynamic analysis; JavaScript

1. Introduction

Most modern programming languages offer support for reflective programming. For instance, Java programs can inspect an object's class at runtime to discover or access its fields and methods, and they can dynamically load classes by name, or even create new classes from scratch. Modern dynamic languages such as Ruby or JavaScript are even more liberal and permit almost arbitrary reflective changes to a running program's state. JavaScript, for example, provides `for-in` loops to discover an object's properties, and dynamic property accesses to read, write or even delete properties referenced by a computed name. Finally, the notorious `eval` function can execute arbitrary text as code, which can access or even declare local variables of the enclosing scope.

While cherished by many programmers for their conciseness and expressiveness, these features make any form of sound program analysis very hard. As an extreme example, a program could evaluate arbitrary user input as program code, thus defeating any attempts to obtain useful analysis results. In practice, however, programmers tend to use reflective features in a more disciplined fashion. For instance, Bodden et al. [5] found that Java programs using reflective class loading tend to always load the same classes, so by observing the classes loaded on some test runs, an analysis can gain a fairly complete picture of the program's reflective behavior. Similarly, Furr et al. [13] report that while dynamic features in Ruby are used pervasively, most uses are "highly constrained" and can be replaced with static alternatives, and Jensen et al. [17] show similar results for uses of `eval` in JavaScript.

This paper proposes a general approach for soundly identifying such constrained uses of dynamic language features. The basic concept behind our approach is *determinacy*.[1] Roughly speaking, a variable `x` is *determinate* at a program point p if `x` must have the same value, say v, whenever program execution reaches p. In that case, we will also say that x *determinately has value* v at p. We allow the program point p to be qualified by a calling context c to account for cases where `x` is only guaranteed to have value v if p is reached while executing inside calling context c. By extension, an expression is determinate at at program point p if its value is computed only from variables that are determinate at p.

Determinacy information can be used by a client to reason about reflective code. For instance, if for some use of `eval` the argument is known to determinately have some string value s, then a static analysis can analyze the code represented by s and maintain soundness. Type checkers, program transformation tools, and static optimizers could also benefit from this kind of specialization.

Determinacy information is not only useful for dealing with dynamic features. For example, in JavaScript it is not uncommon for highly polymorphic functions to behave very differently depending on the type and number of arguments they are passed. Figure 1 shows a short, highly simplified excerpt from the popular jQuery library,[2] which defines a utility function `$` that performs different

[1] Determinacy should not be confused with the notion of determinism in concurrent programming.

[2] See `http://www.jquery.com`

```
function $(selector) {
  if(typeof selector === "string") {
    if(isHTML(selector))
      // parse as HTML and return DOM tree
    else
      // interpret as CSS query
  } else if(typeof selector === "function") {
    // install as event handler for document ready
  } else {
    return [selector];
  }
}
```

Figure 1. Example of polymorphic function in JavaScript

[*] This paper was written while the author was with IBM Research.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.
Copyright © 2013 ACM 978-1-4503-2014-6/13/06. . . $10.00

tasks depending on its argument. If the argument is a string (line 2), it is either parsed as HTML and the corresponding DOM fragment is returned (line 3), or it is interpreted as a CSS selector and all matching elements are returned (line 5). If the argument is a function (line 7), it is installed as an event handler to be executed once the HTML document has finished loading. Otherwise (line 9), the argument is wrapped in a single-element array and returned.

Individual call sites of `$` tend to be monomorphic, i.e., they only exercise one particular functionality of the function, so the conditions of the `if` statements become determinate for that site. For instance, under a call `$(function() { ... })`, the conditional expression on line 2 must be false, whereas line 7 must evaluate to `true`. A flow-insensitive static analysis can use this information to identify code that is unreachable for this particular invocation of `$`, thereby gaining a degree of flow sensitivity. Similarly, a partial evaluator could use determinacy information to complement binding time analysis for specializing a program with respect to determinate input, and an optimizer could use it to detect dead code.

Statically computing determinacy information, however, is very challenging, especially for dynamic languages like JavaScript, where even a call graph is hard to construct without an effective approach to dealing with dynamic language features [30], the very problem determinacy analysis is supposed to help with.

In this paper, we propose a *dynamic determinacy analysis* that analyzes one or more concrete executions to infer determinacy facts that are guaranteed to hold in *any* execution. Our analysis is similar in spirit to dynamic information flow analysis [32], in the sense that we consider indeterminate program inputs to be "tainted" and track the flow of this taint. Any value that is not tainted must then be determinate. As in dynamic information flow, our analysis must consider how indeterminate inputs can influence other values both directly (by participating in computations) and indirectly (e.g., by being used as an `if` condition).

Since our determinacy analysis is dynamic, it can only observe one execution at a time and has to carefully account for the possible effects of code that was not exercised in this execution, but may be exercised in another. Determinacy information itself can help here: if the conditional expression of an `if` statement is determinate, for instance, the dynamic analysis only has to explore one branch, since the other branch is definitely not run in any execution. To handle indeterminate conditionals, we introduce the technique of *counterfactual execution*, where we explore both the branch that would ordinarily be executed, *and* the other branch that would not normally be run. In this way, we can more precisely account for the behavior of other executions and infer more determinacy facts.

We have formalized our analysis for a simple imperative language with records and higher-order functions and proved its correctness: variables marked as determinate by the analysis at a certain program point will in fact have the predicted value any time an execution reaches that point. We have implemented a prototype determinacy analysis that handles all of JavaScript, and obtained promising results on two case studies where we combined the determinacy analysis with a static analysis: first, we extended the static points-to analysis for JavaScript implemented in WALA [30] to make use of determinacy facts, dramatically increasing precision and scalability in some cases; second, we further extended the static analysis to use determinacy facts for eliminating uses of `eval`.

In summary, this paper makes the following main contributions:

- We introduce a **dynamic determinacy analysis**, which analyzes one or more concrete executions to identify variables and expressions that have the same value at a given program point in any execution. (Section 2).
- We give a **formalization** and a **correctness proof** of our determinacy analysis for a simple imperative language. (Section 3).

```
 1 (function() {
 2   function checkf(p) {
 3     // ⟦p.f<32⟧_{16→4} = true; ⟦p.f<32⟧_{25→4} = ?
 4     if(p.f < 32)
 5       setg(p, 42);
 6   }
 7
 8   function setg(r, v) {
 9     r.g = v;
10   } // ⟦r.g⟧_{18→5→10} = 42
11
12   var x = { f : 23 },
13       y = { f : Math.random()*100 };
14   // ⟦x.f⟧_{14} = 23, ⟦y.f⟧_{14} = ?
15
16   checkf(x);
17   // ⟦x.f⟧_{17} = 23, ⟦x.g⟧_{17} = 42
18   checkf(y);
19   // ⟦y.g⟧_{19} = ?
20
21   (y.f > 50 ? checkf : setg)(x, 72);
22   // ⟦x.g⟧_{22} = ?
23
24   var z = { f: x.g - 16, h: true };
25   checkf(z);
26 })();
```

Figure 2. Example program for illustrating dynamic determinacy analysis. Some key determinacy facts are given in comments.

- We present a **prototype implementation** of the analysis for JavaScript. (Section 4).
- We report on **two case studies**, demonstrating that the information gathered by our dynamic determinacy analysis can significantly improve the precision of static pointer analysis and identify many uses of `eval` that can be eliminated. (Section 5).

Finally, Section 6 surveys related work, and Section 7 concludes.

2. Overview

This section provides a high-level overview of determinacy and its applications. First, we introduce some terminology and informally explain how determinacy facts can be computed using dynamic analysis, using a small JavaScript example that was carefully contrived to demonstrate the key issues involved. Then, we illustrate the usefulness of determinacy analysis by showing how determinacy facts can be used to improve static pointer analysis for JavaScript and eliminate common uses of the `eval` function.

2.1 Determinacy

To illustrate key challenges for determinacy analysis, consider the example program in Figure 2. It consists of two functions `checkf` (lines 2–6) and `setg` (lines 8–10), and a sequence of statements (lines 12–25). These functions and statements are wrapped into an anonymous function call to provide lexical scope.

Running the example program. Program execution starts with the call on line 26, which invokes the anonymous function defined on lines 1–26. Executing its body, we first encounter the function definitions for `checkf` and `setg`. Then, when line 12 is reached, variables `x` and `y` are initialized to object literals, both containing a single property `f`. While property `x.f` is set to the constant value `23`, property `y.f` is initialized to `Math.random()*100`, a random floating point number between `0` and `100`; let us assume that it evaluates to `31.4` in our concrete execution.

Execution then proceeds by calling `checkf` twice, on lines 16 and 18, passing `x` and `y` as the argument, respectively. If the condition on line 4 evaluates to true, `checkf` calls `setg`, which assigns the value `42` to the `g` property of the argument. In our example execution, the condition evaluates to true for both calls, so `x.g` and `y.g` both hold the value `42` after line 18. Then, line 21 invokes a function whose name is determined by the conditional expression `(y.f > 50 ? checkf : setg)`. Since `y.f` holds the value `31.4` at this point, `setg` is called, with arguments `x` and `72`, causing the value `72` to be assigned into `x.g`. Next, on line 24, a third local variable `z` is initialized to an object literal with properties `f` and `h` containing values `56` and `true`, respectively. Finally, when `checkf` is called with `z` as an argument on line 25, the condition on line 4 evaluates to false so that `setg` is not called.

Determinacy facts. The goal of determinacy analysis is to identify situations where a variable or expression must hold the same value at a given location in *any* execution of the program.

For example, `x.f` is determinate on line 14: it must always have the value 23 here because its initialization does not depend in any way on program input, or on values in the environment. On the other hand, `y.f` is *indeterminate* on line 14, because the value returned by `Math.random` may vary in different executions. We express these observations using the following *determinacy facts*:

$$[\![\texttt{x.f}]\!]_{14} = \texttt{23}$$
$$[\![\texttt{y.f}]\!]_{14} = \textbf{?}$$

Likewise, variables `x` and `y` are both determinate at line 14.[3]

Sometimes, determinacy facts only hold under a given calling context. For instance, for the call to `checkf` on line 16, the condition `p.f < 32` on line 4 will evaluate to true in any program execution, but this is not the case for the calls to `checkf` on lines 18 and 25. We express this as a *qualified determinacy fact*

$$[\![\texttt{p.f < 32}]\!]_{16\to4} = \texttt{true}$$

stating that the condition `p.f < 32` is determinately true on line 4 if this line is reached from the call at line 16. Determinacy facts inferred by our dynamic analysis are always qualified with a complete call stack reaching all the way back to the program's entrypoint.

Dynamic determinacy analysis. To infer determinacy facts, we instrument the program to compute not only the values of variables, properties and other expressions, but also whether they are determinate at the current program point. This is done using techniques reminiscent of dynamic information flow analysis, where instead of high and low security levels we track indeterminacy.

Program inputs (including, for JavaScript, the DOM) are considered indeterminate, as are the results of certain functions like `Math.random`. Constants, on the other hand, are determinate, so the analysis marks `x` and `x.f` as determinate on line 12.

Starting from these sources, indeterminacy (like high-security annotations in information flow) propagates both directly and indirectly: a variable becomes indeterminate when it is assigned an expression involving other indeterminate variables, but also when it is modified in code that is control dependent on an indeterminate condition and hence not necessarily run in every execution.

Direct propagation of indeterminacy is easy to track by computing a compound expression's determinacy from its constituent expressions: e.g., `Math.random()*100` is indeterminate, so `y.f` is indeterminate at line 13. To track indirect flow of indeterminacy, the dynamic analysis has to carefully reason about conditionals.

Conditionals and counterfactual execution. Conditionals where the condition is itself determinate are easy to handle, since any other execution will take the same branch as the one we are observing. This case arises in the call to `checkf` on line 16, where the condition `p.f < 32` is determinately true, so by simply continuing execution we can infer the fact $[\![\texttt{x.g}]\!]_{17} = \texttt{42}$.

Indeterminate conditions require a more careful treatment. For the second call to `checkf` (line 18), the condition `p.f < 32` on line 4 is indeterminate, but happens to be true. In another execution it may well be false, causing the branch not to be executed. To account for this possibility, we execute the branch as usual, but record any variable or property write. After the branch has finished executing, we mark all written variables and properties (in this case only `y.g`) as indeterminate, giving $[\![\texttt{y.g}]\!]_{19} = \textbf{?}$.

By marking variables indeterminate only *after* the branch has finished executing, we can infer more determinacy facts inside it. Note that both arguments to `setg` are determinate at line 5 when `checkf` is invoked from line 18. So, we infer $[\![\texttt{r.g}]\!]_{18\to5\to10} = \texttt{42}$, even though `y.g` is later marked indeterminate. Of course, `x.f` and `x.g` stay determinate, as they are not written to in the branch.

The third, and most difficult, case arises with conditions that are indeterminate and happen to be false, as is the case for `p.f < 32` in the third call to `checkf` on line 25. Here, we have to account for the fact that the condition may evaluate to true in another execution, causing the branch to be taken.

A simple approach would be to statically examine the code to determine which local variables it could write to (none in this case) and mark them indeterminate. However, this does not account for possible heap writes by functions called within the branch. Since a dynamic analysis does not have access to a static call graph, a conservative approach is to "flush" the heap, marking every property of every object as indeterminate.

This is clearly very imprecise, so our analysis takes another approach, which we refer to as *counterfactual execution*. Even though the condition is false, we still (counterfactually) execute the branch, recording any determinacy facts that may arise. After it has finished executing, we (i) undo the effect of any writes to variables or properties (since the branch was not originally meant to execute), and (ii) mark any written variables and properties as indeterminate (since other executions may not perform these writes). In our case, this leads to `z.g` being marked indeterminate after the conditional, while, for instance, `z.h` is still determinate.

Indeterminate calls. Since JavaScript functions are first-class, the target of a function call must itself be computed as a value, which may be indeterminate. An example is the call on line 21, where the callee is computed by the indeterminate expression `y.f > 50 ? checkf : setg`. In the concrete execution we are considering, it evaluates to `setg`, which is then invoked to set `x.g` to `72`. However, the value of the callee expression is indeterminate since `y.f` is indeterminate at line 21, so another execution may well invoke `checkf` instead, which would result in `x.g` being `42`.

To account for this, we conservatively flush the heap, marking all properties as indeterminate after every indeterminate call. In our example, this means that `x.f`, `x.g`, `y.f` and `y.g` are indeterminate at line 22. While this is overly conservative (`x.f` is, in fact, still determinate), it is the best thing we can do, since we do not track alternative values for indeterminate expressions. Note that `x` and `y` need not be made indeterminate, since they are local variables and cannot possibly be written by any called function.

2.2 Improving pointer analysis

We now show how determinacy facts can be used to improve static pointer analysis for JavaScript. Static analysis of JavaScript is very challenging [18, 27, 30], due to pervasive use of reflective constructs in JavaScript programs and the absence of of static types. Determinacy facts can be used to rewrite uses of reflection into more verbose code that is easier to analyze, as we illustrate below.

[3] While `y.f` is indeterminate, `y` itself must hold the object literal assigned to it at line 13.

```
 1 function Rectangle(w, h) {
 2   this.width = w;
 3   this.height = h;
 4 }
 5
 6 Rectangle.prototype.toString = function() {
 7   return "["+this.width+"x"+this.height+"]";
 8 };
 9
10 String.prototype.cap = function() {
11   return this[0].toUpperCase()+this.substr(1);
12 };
13
14 function defAccessors(prop) {
15   Rectangle.prototype["get" + prop.cap()] =
16     function() { return this[prop]; };
17
18   Rectangle.prototype["set" + prop.cap()] =
19     function(v) { this[prop] = v; };
20 }
21
22 var props = ["width", "height"];
23 for (var i=0; i < props.length; i++)
24   defAccessors(props[i]);
25
26 var r = new Rectangle(20, 30);
27 r.setWidth(r.getWidth()+20);
28 alert(r.toString());  // [40x30]
```

Figure 3. Example program illustrating the use of determinacy facts for improving a static pointer analysis

Example. Consider the example program in Figure 3, which is loosely modeled after code seen in jQuery and other commonly-used JavaScript frameworks. The program defines a `Rectangle` constructor function on lines 1–4, which initializes properties `width` and `height` to the values passed in parameters `w` and `h`, respectively. A `toString()` method is defined on the `Rectangle.prototype` object (lines 6–8), thereby making it available on all objects created by `Rectangle`. Similarly, a utility method `cap` for performing string capitalization is defined on `String.prototype` (line 10), which means that it can be invoked on any string object.

We now use a standard trick employing dynamic property accesses to concisely define getter and setter methods for the `width` and `height` properties. We iterate over the array `["width", "height"]` using the for loop on lines 23–24, invoking the `defAccessors` function twice, first with `prop` set to `"width"`, then to `"height"`. Function `defAccessors` (lines 14–20) defines a getter function (line 15–16) and a setter function (line 18–19), and stores them into `Rectangle.prototype`. The names of these accessor methods are computed dynamically, capitalizing the property name using `cap` and prepending `"get"` and `"set"`, respectively. Thus, the getter methods end up in properties `getWidth` and `getHeight`, and similar for the setters. Lines 26–28 illustrate how these getter and setter methods can be used: Executing these lines will bring up an alert box that reads `[40x30]`.

Applying static pointer analysis. Now let us consider how a standard static pointer analysis algorithm such as 0-CFA [29] would analyze this program. Since such algorithms typically do not track string values, they would conservatively treat the writes on lines 15 and 18 as possibly writing to *any* property of `Rectangle.prototype`. From this, they would conclude that the call to `getWidth` on line 27 could invoke *either* of these functions, and that the call to `toString` on line 28 could invoke the getter, the setter, or the actual `toString` method, which is very imprecise. Even a recent, more advanced analysis that reasons about correlated dynamic property accesses [30] is unable to determine the result of applying `cap` to the property name, yielding the same result.

Static string analyses [8] can reason abstractly about string operations, but are expensive and rely on a call graph. If, as in this example, such reasoning is required to construct a precise call graph to begin with, an iterative scheme interleaving call graph construction and string analysis would have to be devised. By contrast, a context-sensitive static analysis can be adapted to leverage determinacy facts in a fairly straightforward manner, as we now show.

Using determinacy facts. Our determinacy analysis produces facts that can help the static pointer analysis handle complex string operations. For example:

- The analysis shows that the `prop` argument to `defAccessors` is determinate in the first iteration of the loop at lines 23–24:

  $$[\![\,\texttt{prop}\,]\!]_{24^0 \rightarrow 15} = \texttt{"width"}$$

  Here, 24^0 denotes the first time execution reaches line 24.
- Within the `defAccessors` call, the result of the `prop.cap()` call is also determinate:

  $$[\![\,\texttt{this[0].toUpperCase()+...}\,]\!]_{24^0 \rightarrow 15 \rightarrow 11} = \texttt{"Width"}$$

  which makes the result of the string concatenation determinate:

  $$[\![\,\texttt{"get" + prop.cap()}\,]\!]_{24^0 \rightarrow 15} = \texttt{"getWidth"}$$

Similar facts hold for the second loop iteration, and since `props.length` is determinate at line 23, the determinacy analysis can also show that the loop executes at most two times. Given these determinacy facts, the pointer analysis can use loop unrolling and context sensitivity to precisely handle the property writes at lines 15 and 18. First, the analysis unrolls the loop at lines 23–24 twice, using the iteration bound given by the determinacy analysis. Then, via context sensitivity, each call site of `defAccessors` in the unrolled loop is treated as invoking a function specialized by the corresponding determinacy facts. For example, the first call site, corresponding to the loop iteration with `props[i] == "width"`, invokes the following specialized function:

```
function defAccessors(prop) {
  Rectangle.prototype.getWidth =
    function() { return this.width; };
  Rectangle.prototype.setWidth =
    function(v) { this.width = v; };
}
```

For this specialized code, the pointer analysis can easily prove that only the getter function is written into property `getWidth`, and that only the setter function is written into property `setWidth`, thus enabling precise resolution of the function call at line 27.

Besides replacing dynamic property accesses with static ones, determinacy facts can also be used to eliminate conditionals where the condition is determinate, helping to analyze cases like that shown in Figure 1. In Section 5.1, we shall show how our implementation of these techniques achieved large speedups when analyzing certain versions of jQuery.

2.3 Eliminating `eval`

JavaScript's `eval` function transforms text into executable code at run time. Although strongly discouraged by leading practitioners [11], it is pervasively used in practice [26], leaving static analyses with a stark choice: if `eval` is not handled, soundness is compromised; if it is handled conservatively, analysis results will become extremely imprecise, and thus useless for most applications.

Recent research suggests that many uses of `eval` could, in fact, be replaced by equivalent `eval`-free code [17, 23]. For example,

```
 1 ivymap = window.ivymap || {};
 2 function showIvyViaJs(locationId) {
 3   var _f = undefined;
 4   var _fconv = "ivymap[\'"+locationId+"\']";
 5   try {
 6     _f = eval(_fconv);
 7     if (_f!=undefined) {
 8       _f();
 9     }
10   } catch(e) {
11   }
12 }
13
14 showIvyViaJs('pc.sy.banner.tcck.');
15 showIvyViaJs('pc.sy.banner.duilian.');
```

Figure 4. Program (taken from [17]) for which determinacy analysis can show that all calls to `eval` have a determinate argument.

consider the code in Figure 4, taken from Jensen et al. [17], who extracted it from a real-world website.

Clearly, the strings passed to `eval` are determinate in both invocations of `showIvyViaJS`:

$$[\![\texttt{\_fconv}]\!]_{14\to 6} = \texttt{"ivymap['pc.sy.banner.tcck.']"}$$
$$[\![\texttt{\_fconv}]\!]_{15\to 6} = \texttt{"ivymap['pc.sy.banner.duilian.']"}$$

Using the same approach as in the previous subsection, a static analysis can specialize each call to `showIvyViaJS`, replacing calls to `eval` with the code obtained by (statically) parsing the string values found by the determinacy analysis, thus eliminating the calls to `eval` altogether. Section 5.2 presents encouraging results from an initial experiment using this approach, which showed that `eval` elimination via determinacy facts can complement existing purely-static approaches like that of TAJS [17].

3. Formalization

In this section, we formalize the dynamic determinacy analysis from Section 2 on a simple imperative language μJS with records, dynamic property accesses and first-order functions. μJS contains most of the features that make determinacy analysis for JavaScript challenging. We exclude prototypes, constructors and other features usually modeled in JavaScript-based core calculi [15], since they are orthogonal to determinacy analysis. For simplicity, we also exclude unstructured control flow.

We start out by formalizing the syntax and (concrete) semantics of μJS. Then, the analysis is formalized as an instrumented semantics over values with determinacy annotations $\cdot^!$ and $\cdot^?$. Finally, we show soundness of the analysis: if a variable has a value $v^!$ at some program point p in *some* instrumented execution, it has value v at the same point in *every* concrete execution.

3.1 Syntax and semantics of μJS

Figure 5 presents the syntax of μJS, which is mostly a subset of that of JavaScript, except that we abbreviate the keyword **function** as **fun**. Instead of the complex statement and expression syntax of JavaScript, μJS only allows simple statements reminiscent of three address code. In particular, conditionals have only a single branch and their conditional expression must be a variable.

Values in μJS are either primitive values, closures, or heap-allocated records. Primitive values are taken from an unspecified set $\mathrm{PrimVal}$ containing a special value **undefined**. We assume that all primitive values are silently coerced to strings or Boolean values where necessary. When viewed as a Boolean value, addresses and records evaluate to `true`. We also assume a set PrimOp of binary operators on primitive values. Closures are represented as a function value plus an environment, as usual. Record literals are always empty, but property load and store statements can be used to add more properties; we do not model property deletion. Note that the name of the property accessed by these statements is itself a variable, and hence can be computed at runtime, just as in JavaScript. There is no designated prototype property, since μJS does not model JavaScript's prototype system. For convenience, we treat records r as total functions from names to values: if $r = \{\overline{x\colon v}\}$, then $r(x_i) = v_i$, and $r(y) =$ **undefined** for $y \notin \overline{x}$.

$l \in \mathrm{Literal}$	$::=$	pv	primitive value
	$\mid$	$\textbf{fun}(x)\ \{\ \textbf{var}\ \overline{y};\ \overline{s};\ \textbf{return}\ z;\ \}$	function value
	$\mid$	$\{\}$	empty record
$s \in \mathrm{Stmt}$	$::=$	$x = l$	literal load
	$\mid$	$x = y$	variable copy
	$\mid$	$x = y[z]$	property load
	$\mid$	$x[y] = z$	property store
	$\mid$	$x = y \diamond z$	primitive operator
	$\mid$	$x = f(y)$	function call
	$\mid$	$\textbf{if}(x)\{\overline{s}\}$	conditional
	$\mid$	$\textbf{while}(x)\{\overline{s}\}$	loop

$x, y, z, f \in \mathrm{Name};\ \diamond \in \mathrm{PrimOp};\ pv \in \mathrm{PrimVal};\ a \in \mathrm{Address}$

$v \in \mathrm{Value}$	$::=$	$pv \mid a \mid (\textbf{fun}(x)\ \{b\}, \rho)$
$r \in \mathrm{Rec}$	$::=$	$\{\overline{x\colon v}\}$
$\rho \in \mathrm{Env}$	$:=$	$\mathrm{Name} \rightharpoonup \mathrm{Value}$
$h \in \mathrm{Heap}$	$:=$	$\mathrm{Address} \rightharpoonup \mathrm{Rec}$

Figure 5. Syntax and semantic domains of μJS; b abbreviates function bodies

$e \in \mathrm{Event}$	$::=$	$x = v$	variable write
	$\mid$	$a[x] = v$	heap write
	$\mid$	$\textbf{if}(v)\{t\}$	conditional
	$\mid$	$\textbf{fun}(v)\{t\}_\rho$	function call
$t \in \mathrm{Trace}$	$:=$	$\overline{e}$	

Figure 6. Concrete execution traces

Our semantics for μJS is big-step. While a small-step semantics would allow more direct reasoning about individual program points, it makes it hard to match up control flow forks and joins, which play a crucial role in computing determinacy information. To be able to speak about intermediate states, we use a trace-based semantics [22] with an evaluation judgement of the form $(h, \rho, s) \downarrow (h', \rho', t)$: h and ρ are the initial heap and environment, s is the statement to evaluate; h' and ρ' are the resulting heap and environment, and t is a *trace* recording all assignments, conditionals and function calls executed during evaluation of s.

Figure 6 gives the grammar of trace events and traces; the environment ρ in a function call event denotes the closure environment of the called function. Figure 8 lists the evaluation rules, which are mostly standard, except that they each add an event to the trace. We assume that the semantics of built-in primitive operators $\diamond$ is given by a partial function $[\![\diamond]\!]$. If this function is undefined on the provided arguments, execution gets stuck, as is the case when accessing undefined local variables or trying to invoke a non-function. In full JavaScript, these situations give rise to elaborate implicit conversions or exceptions, which we do not model.

Local variables have to be declared before use (hence the precondition $x \in \mathrm{dom}(\rho)$ in most rules), and we do not model globals.

$$
\begin{array}{lll}
d \in D & := & \{!, ?\} \\
\hat{v} \in \widehat{\text{Value}} & ::= & pv^d \mid a^d \mid (\textbf{fun}(x)\ \{b\}, \hat{\rho})^d \\
\hat{r} \in \widehat{\text{Rec}} & ::= & \{\overline{x\colon \hat{v}}\} \mid \{\overline{x\colon \hat{v}}, \ldots\} \\
\hat{h} \in \widehat{\text{Heap}} & := & \text{Address} \rightharpoonup \widehat{\text{Rec}} \\
\hat{\rho} \in \widehat{\text{Env}} & := & \text{Name} \rightharpoonup \widehat{\text{Value}} \\
\hat{e} \in \widehat{\text{Event}} & ::= & x = \hat{v} \mid a^d[x^{d'}] = \hat{v} \mid \textbf{if}(\hat{v})\{\hat{t}\} \\
 & \mid & (\textbf{fun}(\hat{v})\{\hat{t}\})^d_{\hat{\rho}} \\
\hat{t} \in \widehat{\text{Trace}} & := & \overline{\hat{e}}
\end{array}
$$

Figure 7. Instrumented semantic domains and traces

While local variables can be modified, their semantics is quite idiosyncratic: updates are only visible inside the updating function, so updating a closure variable will not have the desired effect. A more standard semantics can be achieved by modeling local variables that are accessed in a nested functions as immutable references to records with a single property, and accessing that property instead.

Given a trace t, we can reconstruct what effects the execution that produced it had on the heap and the environment. Its *variable domain* $\text{vd}(t)$ is the set of all variables written during the execution; it consists of all variables x such that t or one of its sub-traces (excluding sub-traces inside function calls[4]) contains an event $x = v$. The *property domain* $\text{pd}(t)$ contains all address-property name pairs $\langle a, p\rangle$ such that property p of the record at a was modified during the execution, which is the case iff t or one of its sub-traces (*including* sub-traces inside function calls) contains an event $a[p] = v$. We also define the variable domain of a sequence of statements, $\text{vd}(\overline{s})$, to contain all variables x such that one of the statements in $\overline{s}$ (without nested functions) is an assignment to x.

Clearly, if $\langle h, \rho, s\rangle \downarrow \langle h', \rho', t\rangle$, then $\forall x \notin \text{vd}(t).\rho(x) = \rho'(x)$, and similarly $\forall (a, p) \notin \text{pd}(t).h(a)(p) = h'(a)(p)$.[5]

3.2 Instrumented semantics

We formalize determinacy analysis via an instrumented semantics of μJS where variables and properties are bound to annotated values of the form $v^!$ and $v^?$, representing determinate and indeterminate values, respectively.

Instrumented values, records, heaps, environments and traces are defined in Figure 7. An instrumented value of the form $v^!$ represents the single concrete value v, whereas $v^?$ represents any concrete value. For records, a *closed record* of the form $\{\overline{x\colon \hat{v}}\}$ represents all records that have precisely the properties $\overline{x}$ with values represented by the corresponding instrumented values $\overline{\hat{v}}$; an *open record* of the form $\{\overline{x\colon \hat{v}, \ldots}\}$, on the other hand, represents all records that have *at least* the properties $\overline{x}$ (again with the same restrictions on their values), but may have more.

Like their concrete counterparts, we also interpret instrumented records as functions; looking up a non-existent property on a closed record yields **undefined**$^!$, and on an open record **undefined**$^?$.

For an instrumented value $\hat{v}$, we define $\hat{v}^?$ as $\hat{v}$ with all $\cdot^!$ annotations replaced by $\cdot^?$. For records $\hat{r}$, $\hat{r}^?$ additionally turns $\hat{r}$ into an open record. For heaps, we let $\hat{h}^?$ be the heap resulting from $\hat{h}$ by replacing every record $\hat{r}$ with $\hat{r}^?$, which corresponds to a heap flush. We also define $\hat{v}^! := \hat{v}$, $\hat{r}^! := \hat{r}$, $\hat{h}^! := \hat{h}$ i.e., the extant determinacy annotations are retained.

[4] Function calls need not be considered, since callees cannot update their caller's local variables.

[5] These equalities should be considered to hold if either both sides are undefined, or both are defined and equal.

$$
(\textsc{LdLit})\ \frac{x \in \text{dom}(\rho)}{\langle h, \rho, x = pv\rangle \downarrow \langle h, \rho[x \mapsto pv], x = pv\rangle}
$$

$$
(\textsc{LdClos})\ \frac{F \equiv \textbf{fun}(y)\ \{\ \textbf{var}\ \overline{y};\ \overline{s};\ \textbf{return}\ z;\ \} \quad x \in \text{dom}(\rho)}{\langle h, \rho, x = F\rangle \downarrow \langle h, \rho[x \mapsto (F, \rho)], x = (F, \rho)\rangle}
$$

$$
(\textsc{LdRec})\ \frac{x \in \text{dom}(\rho) \quad a \notin \text{dom}(\rho)}{\langle h, \rho, x = \{\}\rangle \downarrow \langle h[a \mapsto \{\}], \rho[x \mapsto a], x = a\rangle}
$$

$$
(\textsc{Assign})\ \frac{x \in \text{dom}(\rho) \quad \rho(y) = v}{\langle h, \rho, x = y\rangle \downarrow \langle h, \rho[x \mapsto v], x = v\rangle}
$$

$$
(\textsc{Ld})\ \frac{x \in \text{dom}(\rho) \quad \rho(y) = a \quad \rho(z) = z' \quad h(a) = r \quad r(z') = v}{\langle h, \rho, x = y[z]\rangle \downarrow \langle h, \rho[x \mapsto v], x = v\rangle}
$$

$$
(\textsc{Sto})\ \frac{\rho(x) = a \quad \rho(y) = y' \quad h(a) = r \quad \rho(z) = v}{\langle h, \rho, x[y] = z\rangle \downarrow \langle h[a \mapsto r[y' \mapsto v]], \rho, a[y'] = v\rangle}
$$

$$
(\textsc{PrimOp})\ \frac{x \in \text{dom}(\rho)\ \rho(y) = pv_1\ \rho(z) = pv_2\ pv_1[\![\diamond]\!]pv_2 = pv_3}{\langle h, \rho, x = y \diamond z\rangle \downarrow \langle h, \rho[x \mapsto pv_3], x = pv_3\rangle}
$$

$$
(\textsc{Inv})\ \frac{\begin{array}{c} x \in \text{dom}(\rho) \quad \rho(f) = (\textbf{fun}(z)\ \{\textbf{var}\ \overline{x'};\overline{s};\ \textbf{return}\ y';\}, \rho') \quad \rho(y) = v \\ \langle h, \rho'[z \mapsto v, \overline{x' \mapsto \textbf{undefined}}], \overline{s}\rangle \overline{\downarrow} \langle h', \rho'', t\rangle \quad \rho''(y') = v' \end{array}}{\langle h, \rho, x = f(y)\rangle \downarrow \langle h', \rho[x \mapsto v'], (\textbf{fun}(v)\{t\}_{\rho'}; x = v')\rangle}
$$

$$
(\textsc{If}_1)\ \frac{\rho(x) = v \quad v = \textbf{true} \quad \langle h, \rho, \overline{s}\rangle \overline{\downarrow} \langle h', \rho', t\rangle}{\langle h, \rho, \textbf{if}(x)\{\overline{s}\}\rangle \downarrow \langle h', \rho', \textbf{if}(v)\{t\}\rangle}
$$

$$
(\textsc{If}_2)\ \frac{\rho(x) = v \quad v = \textbf{false}}{\langle h, \rho, \textbf{if}(x)\{\overline{s}\}\rangle \downarrow \langle h, \rho, \textbf{if}(v)\{\}\rangle}
$$

$$
(\textsc{While})\ \frac{\langle h, \rho, \textbf{if}(x)\{\overline{s};\ \textbf{while}(x)\{\overline{s}\}\}\rangle \downarrow \langle h', \rho', t\rangle}{\langle h, \rho, \textbf{while}(x)\{\overline{s}\}\rangle \downarrow \langle h', \rho', t\rangle}
$$

$$
(\textsc{Seq})\ \frac{\langle h_i, \rho_i, s_i\rangle \downarrow \langle h_{i+1}, \rho_{i+1}, e_i\rangle}{\langle h_0, \rho_0, \overline{s}\rangle \overline{\downarrow} \langle h_n, \rho_n, \overline{e}\rangle}
$$

Figure 8. Concrete semantics of μJS

An instrumented trace looks like a concrete trace, except that it contains instrumented values instead of concrete ones. For a function call event $(\textbf{fun}(\hat{v})\{\hat{t}\})^d_{\hat{\rho}}$, for instance, $\hat{v}$ is the argument value, $\hat{\rho}$ is the closure environment of the callee, and $\hat{t}$ is the trace resulting from the call. The annotation d indicates whether the callee is determinate or not. The variable and property domains $\text{vd}(\hat{t})$ and $\text{pd}(\hat{t})$ are defined exactly as for concrete traces, simply disregarding determinacy annotations.

As described in Section 2, after executing a branch guarded by an indeterminate but true condition, the analysis marks every variable written in the branch as indeterminate. Similarly, after counterfactually executing a branch, every written variable is reset to its previous value *and* marked indeterminate.

To model this in our semantics, we use the notation $\hat{\rho}'[V := \hat{\rho}^d]$, where $\hat{\rho}, \hat{\rho}'$ are instrumented environments, V is a set of variable names, and d is a determinacy flag. Roughly, this means that $\hat{\rho}'$ is updated so that every variable $x \in V$ is reset to the value it had in $\hat{\rho}$, and marked as indeterminate if $d = ?$. More precisely, we define

$$
\hat{\rho}'[V := \hat{\rho}^d](x) := \begin{cases} \hat{\rho}(x)^d & \text{if } x \in \text{dom}(\hat{\rho}) \cap V \\ \hat{\rho}'(x) & \text{otherwise} \end{cases}
$$

Thus, for a trace $\hat{t}$, $\hat{\rho}'[\mathrm{vd}(\hat{t}) := \hat{\rho}'^{?}]$ marks every variable x written by $\hat{h}$ as indeterminate, and $\hat{\rho}'[\mathrm{vd}(\hat{t}) := \hat{\rho}^{?}]$ additionally resets its value to $\hat{\rho}(x)$.

For records $\hat{r}, \hat{r}'$, we define $\hat{r}'[V := \hat{r}^{d}]$ as for environments. For heaps $\hat{h}, \hat{h}'$, a set $A \subseteq \text{Address} \times \text{Name}$ of address-property name pairs and a determinacy flag d, we define

$$\hat{h}'[A := \hat{h}^{d}](a) := \begin{cases} \hat{h}'(a)[A_a := \hat{h}(a)^{d}] & \text{if } a \in \mathrm{dom}(\hat{h}') \cap \\ & \qquad \mathrm{dom}(\hat{h}) \\ \hat{h}'(a) & \text{otherwise} \end{cases}$$

where $A_a := \{p \mid (a, p) \in A\}$.

Given these technical preliminaries, we can now define the evaluation judgement $\langle \hat{h}, \hat{\rho}, s\rangle \hat{\downarrow}^{n} \langle \hat{h}', \hat{\rho}', \hat{t}\rangle$ of the instrumented semantics as shown in Figure 9. The index n is used to bound the nesting depth of counter-factual executions, to be discussed shortly.

We omit ($\widehat{\text{LdClos}}$), ($\widehat{\text{LdRec}}$), ($\widehat{\text{Assign}}$), ($\widehat{\text{While}}$) and ($\widehat{\text{Seq}}$), which are straightforward adaptations of their concrete counterparts. In ($\widehat{\text{PrimOp}}$), the determinacy flags d_1 and d_2 of both operands are applied to the result, so it will be indeterminate if at least one of the operands is. Similarly in ($\widehat{\text{Ld}}$), the result of a property load is only determinate if both the address and the property name are. Rule ($\widehat{\text{Sto}}$) applies the determinacy flag d of the address to be accessed to the whole heap: if $d = ?$, the heap is flushed. Similarly, the flag d' of the property name is applied to the record being accessed: if $d' = ?$, all properties are made indeterminate and it becomes an open record, since any of the existing properties may be written, or a new one added.

The most interesting rules are the ones concerning conditionals. ($\widehat{\text{If}_1}$) handles the case where the condition is (determinate or indeterminate) true by simply executing the branch, and afterwards marking all written properties and variables as indeterminate if the condition is indeterminate. ($\widehat{\text{If}_2\text{-Det}}$) applies when the condition is determinate false; in this case, the conditional is a no-op.

($\widehat{\text{Cntr}}$) initiates counterfactual execution if the condition is indeterminate false, executing the branch while increasing the counterfactuality level to $n + 1$. After the branch has finished executing, ($\widehat{\text{Cntr}}$) reverts to the heap and environment before the counterfactual (i.e., assignments performed by the counterfactually executed branch are not visible), and marks any variable or property possibly written by counterfactual code as indeterminate.

In general, counterfactually executing branches may lead to non-termination, so we introduce a cut-off: when the number of nested counterfactual executions is about to exceed some fixed number k, the pre-condition $n < k$ prevents the ($\widehat{\text{Cntr}}$) from applying; instead, rule ($\widehat{\text{CntrAbort}}$) shortcuts evaluation and conservatively flushes the heap and marks any variable that may possible be assigned anywhere in the branch indeterminate.

3.3 Soundness

To show soundness, we will now prove that the instrumented trace generated by an execution under the instrumented semantics correctly predicts any concrete trace produced by an execution under the normal semantics: where the instrumented trace has $v^{!}$, a concrete trace must have v; where the instrumented trace has $v^{?}$, we cannot predict the corresponding value in the concrete trace.

Obviously, however, this can only hold if instrumented and concrete execution start in compatible states. To formalize this notion, we inductively define a modeling relation $\cdot \models_{\mu} \cdot$ relating instrumented values, environments, records, heaps and traces to their concrete counterparts (Figure 10). The mapping $\mu\colon \text{Address} \to \text{Address}$ maps addresses of the concrete execution to corresponding addresses of the instrumented execution, since the choice of a in (LdRec) is not constrained enough to require both executions to choose precisely the same addresses.

$$(\widehat{\text{LdLit}})\ \frac{x \in \mathrm{dom}(\hat{\rho})}{\langle \hat{h}, \hat{\rho}, x = pv\rangle \hat{\downarrow}^{n} \langle \hat{h}, \hat{\rho}[x \mapsto pv^{!}], x = pv^{!}\rangle}$$

$$(\widehat{\text{Ld}})\ \frac{x \in \mathrm{dom}(\hat{\rho}) \quad \hat{\rho}(y) = a^{d} \quad \hat{\rho}(z) = z'^{d'} \quad \hat{h}(a) = \hat{r} \quad \hat{r}(z') = \hat{v}}{\langle \hat{h}, \hat{\rho}, x = y[z]\rangle \hat{\downarrow}^{n} \langle \hat{h}, \hat{\rho}[x \mapsto (\hat{v}^{d})^{d'}], x = (\hat{v}^{d})^{d'}\rangle}$$

$$(\widehat{\text{Sto}})\ \frac{\hat{\rho}(x) = a^{d} \quad \hat{\rho}(y) = y'^{d'} \quad \hat{h}(a) = \hat{r} \quad \hat{\rho}(z) = \hat{v}}{\langle \hat{h}, \hat{\rho}, x[y] = z\rangle \hat{\downarrow}^{n} \langle (\hat{h}[a \mapsto (\hat{r}[y' \mapsto \hat{v}])^{d'}])^{d}, \hat{\rho}, a^{d}[y'^{d'}] = \hat{v}\rangle}$$

$$(\widehat{\text{PrimOp}})\ \frac{x \in \mathrm{dom}(\hat{\rho}) \quad \hat{\rho}(y) = pv_1^{d_1} \quad \hat{\rho}(z) = pv_2^{d_2} \quad pv_1 [\![\diamond]\!] pv_2 = pv_3}{\langle \hat{h}, \hat{\rho}, x = y \diamond z\rangle \downarrow \langle \hat{h}, \hat{\rho}[x \mapsto (pv_3^{d_1})^{d_2}], x = (pv_3^{d_1})^{d_2}\rangle}$$

$$(\widehat{\text{Inv}})\ \frac{\begin{array}{c} x \in \mathrm{dom}(\hat{\rho}) \quad \hat{\rho}(f) = (\textbf{fun}(z)\ \{\textbf{var}\ \overline{x'};\ \overline{s};\ \textbf{return}\ z';\}, \hat{\rho}')^{d} \quad \hat{\rho}(y) = \hat{v} \\ \langle \hat{h}, \hat{\rho}'[z \mapsto \hat{v}, \overline{x' \mapsto \textbf{undefined}^{!}}], \overline{s}\rangle\, \overline{\hat{\downarrow}^{n}} \langle \hat{h}', \hat{\rho}'', \hat{t}\rangle \quad \hat{\rho}''(z') = \hat{v}' \end{array}}{\langle \hat{h}, \hat{\rho}, x = f(y)\rangle \hat{\downarrow}^{n} \langle \hat{h}'^{d}, \hat{\rho}[x \mapsto \hat{v}'^{d}], (\textbf{fun}(\hat{v})\ \{\hat{t}\}^{d}_{\hat{\rho}'}; x = \hat{v}'^{d})\rangle}$$

$$(\widehat{\text{If}_1})\ \frac{\hat{\rho}(x) = v^{d} \quad v = \textbf{true} \quad \langle \hat{h}, \hat{\rho}, \overline{s}\rangle\, \overline{\hat{\downarrow}^{n}} \langle \hat{h}', \hat{\rho}', \hat{t}\rangle}{\langle \hat{h}, \hat{\rho}, \textbf{if}(x)\{\overline{s}\}\rangle \hat{\downarrow}^{n} \langle \hat{h}'[\mathrm{pd}(\hat{t}) := \hat{h}'^{d}], \hat{\rho}'[\mathrm{vd}(\hat{t}) := \hat{\rho}'^{d}], \textbf{if}(v^{d})\{\hat{t}\}\rangle}$$

$$(\widehat{\text{If}_2\text{-Det}})\ \frac{\hat{\rho}(x) = v^{!} \quad v = \textbf{false}}{\langle \hat{h}, \hat{\rho}, \textbf{if}(x)\{\overline{s}\}\rangle \downarrow \langle \hat{h}, \hat{\rho}, \textbf{if}(v^{!})\{\}\rangle}$$

$$(\widehat{\text{Cntr}})\ \frac{\hat{\rho}(x) = v^{?} \quad v = \textbf{false} \quad n < k \quad \langle \hat{h}, \hat{\rho}, \overline{s}\rangle\, \overline{\hat{\downarrow}^{n+1}} \langle \hat{h}', \hat{\rho}', \hat{t}\rangle}{\langle \hat{h}, \hat{\rho}, \textbf{if}(x)\{\overline{s}\}\rangle \hat{\downarrow}^{n} \langle \hat{h}'[\mathrm{pd}(\hat{t}) := \hat{h}^{?}], \hat{\rho}'[\mathrm{vd}(\hat{t}) := \hat{\rho}^{?}], \textbf{if}(v^{?})\{\hat{t}\}\rangle}$$

$$(\widehat{\text{CntrAbort}})\ \frac{\hat{\rho}(x) = v^{?} \quad v = \textbf{false} \quad n \geq k}{\langle \hat{h}, \hat{\rho}, \textbf{if}(x)\{\overline{s}\}\rangle \hat{\downarrow}^{n} \langle \hat{h}^{?}, \hat{\rho}[\mathrm{vd}(\overline{s}) := \hat{\rho}^{?}], \textbf{if}(v^{?})\{\}\rangle}$$

Figure 9. Instrumented semantics for determinacy analysis.

$$v^{?} \models_{\mu} v' \qquad pv^{!} \models_{\mu} pv \qquad \mu(a)^{!} \models_{\mu} a \qquad \frac{F \equiv \textbf{fun}(y)\ \{b\} \quad \hat{\rho} \models_{\mu} \rho}{(F, \hat{\rho})^{!} \models_{\mu} (F, \rho)}$$

$$\frac{\forall x \in \mathrm{dom}(\rho). \hat{\rho}(x) \models_{\mu} \rho(x)}{\hat{\rho} \models_{\mu} \rho} \qquad \frac{\forall a \in \mathrm{dom}(h). \hat{h}(\mu(a)) \models_{\mu} h(a)}{\hat{h} \models_{\mu} h}$$

$$\frac{\forall x. \hat{r}(x) \models_{\mu} r(x)}{\hat{r} \models_{\mu} r} \qquad \frac{\hat{v} \models_{\mu} v}{x = \hat{v} \models_{\mu} x = v} \qquad \frac{a^{d} \models_{\mu} a' \quad p^{d'} \models_{\mu} p' \quad \hat{v} \models_{\mu} v}{a^{d}[p^{d'}] = \hat{v} \models_{\mu} a'[p'] = v}$$

$$\frac{\hat{v} \models_{\mu} v \quad \hat{t} \models_{\mu} t}{\textbf{if}(\hat{v})\{\hat{t}\} \models_{\mu} \textbf{if}(v)\{t\}} \qquad \frac{v' = \textbf{false}}{\textbf{if}(v^{?})\{\hat{t}\} \models_{\mu} \textbf{if}(v')\{\}}$$

$$\frac{\hat{v} \models_{\mu} v}{(\textbf{fun}(\hat{v})\{\hat{t}\})^{?}_{\hat{\rho}} \models_{\mu} \textbf{fun}(v)\{t\}_{\rho}} \qquad \frac{\hat{t} \models_{\mu} t \quad \hat{v} \models_{\mu} v \quad \hat{\rho} \models_{\mu} \rho}{(\textbf{fun}(\hat{v})\{\hat{t}\})^{!}_{\hat{\rho}} \models_{\mu} \textbf{fun}(v)\{t\}}$$

Figure 10. Modeling relations between instrumented values, environments, records, heaps, traces and their concrete counterparts

The rules for values, environments, records and heaps are fairly straightforward. For traces, note that the instrumented and concrete traces of the branches of a conditional need not match if the condition variable is indeterminate false. Similarly, for a function call event we do not require the traces of the called function to correspond if the called function is not determinate.

We can now state the soundness of the instrumented semantics:

Theorem 1. *If* $\langle \hat{h}, \hat{\rho}, s\rangle \hat{\downarrow}^n \langle \hat{h}', \hat{\rho}', \hat{t}\rangle$ *and* $\langle h, \rho, s\rangle \downarrow \langle h', \rho', t\rangle$ *where* $\hat{h} \models_\mu h$ *and* $\hat{\rho} \models_\mu \rho$ *for some* μ *that is injective on* $\mathrm{dom}(h)$, *then* $\hat{h}' \models_{\mu'} h'$, $\hat{\rho}' \models_{\mu'} \rho'$, *and* $\hat{t} \models_{\mu'} t$ *for some* μ' *that is injective on* $\mathrm{dom}(h')$, *and agrees with* μ *and* $\mathrm{dom}(h)$.

A proof of this theorem and its mechanization in Coq are available online at `http://github.com/xiemaisi/determinacy`.

To relate this result to our intuitive description of a determinacy analysis, note that every event in a (concrete or instrumented) trace corresponds to a statement in the program being executed under a certain call stack. The soundness result essentially shows that the instrumented heap and environment at some position p in an instrumented trace correctly model all concrete heaps and environments that may be encountered by a concrete execution at p, provided its initial heap and environment are correctly modeled by the initial instrumented heap and environment.

4. Implementation

We have implemented a prototype dynamic determinacy analysis for JavaScript, which instruments a program to track determinacy flags as in the instrumented semantics of the previous section. Instrumented programs can run in a browser or atop Node.js, using ZombieJS for DOM emulation.[6]

For instrumentation, the program is first translated into a form similar to μJS with a small number of additional statement forms. Almost all of the ECMAScript 5.1 standard is handled, except implicit conversions using `toString` and `valueOf`; getters and setters are partially supported. To handle unstructured control flow, the instrumented program adjusts determinacy information at every control flow merge point, not just after `if`s.

To implement heap flushes, we keep a global *epoch counter*. Every property has a recency annotation, and is only considered determinate if this annotation equals the current epoch. Thus, incrementing the epoch counter flushes the heap.

Native functions from the standard library cannot be instrumented. For some of them, we provide hand-written models that conservatively approximate their effects on determinacy information; e.g., most string handling functions do not affect the determinacy of heap objects. Calling a native function without a model, however, yields an indeterminate result and causes a heap flush.

For DOM functions, we assume that they can only modify DOM data structures, so calling them does not affect the determinacy of other heap locations. Return values of DOM functions, however, are always considered indeterminate, as is any value that is read from a DOM data structure. This is very conservative, and could be improved by providing a more detailed model of the DOM. Since DOM events can fire in any order, we perform a heap flush immediately upon entering an event handler. In practice, this means that few useful determinacy facts can be derived for them.

If counterfactual execution encounters a call to a native function that is not known to be side effect-free, we immediately abort the counterfactual execution and flush the heap. Exceptions during counterfactual execution are handled similarly.

Finally, note that `eval` is easy to handle in a dynamic analysis: calls to `eval` are instrumented to recursively instrument any code loaded at runtime, flushing the heap if the code is not determinate.

Currently, our instrumentation is not optimized, so instrumented code is expected to run slower. In typical applications, however, the main focus of the analysis is on the program's initialization phase, which for most JavaScript programs tends to be fairly short.

[6] See `http://nodejs.org` and `http://zombie.labnotes.org`.

jQuery Version	**Baseline**	**Spec**		**Spec+DetDOM**	
1.0	✗	✓	(82)	✓	(2)
1.1	✗	✗	(107)	✓	(4)
1.2	✓	✓	(>1000)	✓	(0)
1.3	✗	✗	(>1000)	✗	(>1000)

Table 1. Comparison of pointer analysis scalability on several jQuery versions; the number of heap flushes is given in parentheses.

5. Case Studies

This section reports on two case studies combining determinacy analysis with static analysis to improve pointer analysis scalability and to eliminate calls to `eval` as suggested in Section 2.

5.1 Improving pointer analysis

We enhanced the points-to analysis of WALA [30] to exploit determinacy facts computed by our dynamic analysis. This is done by creating clones of functions based on the full call stacks present in determinacy facts to enable specialization, extending the baseline context sensitivity policy used in [30], as discussed in Section 2. We perform three kinds of specializations: (i) removing branches guarded by determinately false conditions; (ii) making dynamic property accesses with determinate property names static; (iii) unrolling loops with a determinate maximum number of iterations if this enables other specializations.

We compared our specializing analysis (**Spec**) to the analysis of [30] (**Baseline**) on several versions of jQuery, as shown in Table 1. For each version, we analyzed an HTML file that includes the jQuery library itself with no additional client code; this is not trivial due to jQuery's complex initialization code [30]. In all cases, the static analysis either completed in less than 5 seconds, indicated by ✓ in the table,[7] or it did not complete within a 10-minute timeout, indicated by ✗. We also give the number of heap flushes performed by the dynamic analysis in parentheses; note that we stop the dynamic analysis after 1000 heap flushes, since at this point it is unlikely to detect new determinacy facts.

Since the DOM model used by the dynamic analysis is very limited, we also ran the dynamic analysis with the additional (unsound) assumption that all properties of DOM objects are determinate, and that operations on the DOM return determinate values (column **Spec+DetDOM**). This avoids many heap flushes and yields more determinacy facts, effectively specializing the code to one particular browser and one HTML document. While unsound in general, this configuration is an indicator of possible improvements to be gained from a richer DOM model.

The results show that determinacy facts can dramatically enhance the scalability of a static pointer analysis: while **Baseline** fails to analyze jQuery 1.0 in 10 minutes, **Spec** analyzes it in a matter of seconds. All specializations listed above were necessary to make this version analyzable; in particular, one loop had to be unrolled 21 times to enable specialization of two critical property writes (similar in style to Figure 3) to make the pointer analysis terminate. The additional context sensitivity needed to make use of determinacy facts is moderate: up to four levels of calling context are required, but only for call sites where a determinacy fact is available; this additional effort pays for itself, since the specialized functions are much easier to analyze than their unspecialized counterparts.

For jQuery 1.1, the determinacy facts were insufficient to make our specializing analysis terminate. Further investigation suggested

[7] As measured on a Lenovo ThinkPad W520 with a 2.20 GHz Intel Core i7-2720QM processor and 8GB RAM running Linux 3.2.0-32, using the OpenJDK 64-Bit Server VM, version 1.7.0_09, with a 5GB maximum heap.

that our imprecise DOM model was to blame, and indeed assuming a determinate DOM makes this version analyzable as well. Again, adding determinacy facts required no more than four levels of context sensitivity and made the analysis dramatically faster.

Interestingly, jQuery 1.2 is easy to analyze for all analysis configurations, even the baseline. This is due to a change that made complex initialization code execute lazily; without client code, this code is dead and need not be analyzed.

jQuery 1.3 (and later versions) are not yet analyzable, even using a determinate DOM. We suspect this is due to more complex event handlers being installed during initialization. Some handlers are not exercised by the dynamic analysis, but even for the ones that are we need to perform a heap flush on entry (see Section 4). Nonetheless, it is clear from our experiments that determinacy analysis has the potential for significantly improving the scalability of pointer analysis of complex JavaScript frameworks such as jQuery.

5.2 Eliminating calls to `eval`

We further enhanced our analysis to also specialize calls to `eval` where `eval` is the only call target and its argument string is determinate. We ran this analysis on the benchmark suite used by Jensen et al. [17]. As they note, there are some difficulties in eliminating uses of `eval` even with constant arguments. Many of these issues do not arise in our case, since we perform specialization on an intermediate representation with fully resolved name bindings.

Jensen et al. report that their *unevalizer* tool could eliminate all uses of `eval` in 19 of 28 benchmarks. Our dynamic analysis needs to be able to run the program, so we had to disregard 3 benchmarks that are missing required code, and one that cannot be run in ZombieJS. On 14 out of the remaining 24 programs, our analysis could specialize all uses of `eval`. Interestingly, this includes six programs that *unevalizer* cannot handle, including the code shown in Figure 4. In this program, the code to be evaluated results from a string concatenation, but their analysis requires the concatenation to be a syntactic part of the `eval` argument expression, which is not the case here. Other cases involve `for-in` loops: if the set of properties to iterate over is determinate, our analysis assumes that the iteration order is also determinate. This is true for all major JavaScript implementations, but *unevalizer* does not assume this.

In the remaining 10 programs, our analysis fails to specialize away at least one use of `eval`. This is either because the evaluated string is genuinely indeterminate (1 case); the dynamic analysis does not cover a use that WALA considers reachable (4 cases); or incomplete DOM modeling causes heap flushes, making the callee of a use of `eval` indeterminate (1 case). In the remaining 4 cases, `eval` occurs inside a loop for which the dynamic analysis cannot derive a determinate upper bound, and which hence cannot be specialized. In only one of these cases is the loop bound truly indeterminate; in the other cases, the imprecision is again caused by heap flushes due to missing DOM modeling.

We also ran our analysis with the potentially unsound determinate DOM assumption described in Section 5.1. This enabled better inference of determinate loop bounds and detection of unreachable code, allowing it to handle 20 benchmarks.

In conclusion, we found that determinacy analysis can be an effective tool for eliminating calls to `eval`.

5.3 Discussion

The case studies above show that determinacy facts can be useful for client analyses like flow-insensitive static analyses. In principle, the additional context sensitivity needed to take advantage of the facts could be very expensive, but it seems that, in practice, scalability problems are often due to a *lack* of determinacy facts.

A more sophisticated DOM model could help finding more determinacy facts, as could a more precise treatment of event handlers. However, soundly reasoning about all possible interleavings of event handlers is a challenging task that we leave to future work.

6. Related Work

Constant Propagation Like determinacy analysis, constant propagation aims to discover expressions that have the same value in every execution. Unlike in determinacy analysis, this value is usually required to be independent of calling context (though context-qualified constants have also been considered [24]) and primitive (i.e., not an object or array). Constant propagation typically relies on a sound static call graph, which is hard to obtain in dynamic languages like JavaScript. Our approach does not require a static call graph, but can only produce fully qualified determinacy facts.

Partial Evaluation Partial evaluation [19] seeks to identify and evaluate program parts that only depend on designated *static inputs*, yielding a *residual program* that can be run on the remaining inputs. *Online* partial evaluation generates the residual program in one pass, whereas *offline* evaluation relies on static analysis to specialize the program. Usually, it first computes a call graph, and then performs a *binding time analysis* to determine expressions whose value only depends on static inputs. A *monovariant* binding time analysis assigns a single classification to every expression, whereas a *polyvariant* analysis [10] takes contexts into account.

Our determinacy analysis can be viewed as a polyvariant binding time analysis where static program inputs are considered determinate. Typical binding time analyses are completely static and hence do not need to know the concrete values of any inputs. Online partial evaluators, on the other hand, operate on given concrete values for the static inputs. By contrast, our analysis is completely dynamic, and hence requires concrete values for *all* inputs.

The main advantage of our approach for JavaScript is that it does not rely on a static analysis. Unlike an online partial evaluator, we do not need heuristics for ensuring termination: our analysis can be run as long as desired and aborted at any point without endangering the soundness of the results.

Symbolic Execution Our counterfactual execution can be viewed as a very limited form of symbolic execution [20]. Unlike symbolic execution, our analysis performs a merge at control-flow join points, rather than maintaining two “forked” executions. This trades precision for performance, since we cannot reason about all possible values of indeterminate expressions.

Information Flow Analysis Our determinacy analysis is similar to a dynamic information flow analysis [3, 32] with indeterminate inputs corresponding to high-security inputs. Since we want to infer determinacy facts even under indeterminate conditionals, we do not mark constants under such conditionals as indeterminate immediately (as an information flow analysis would), but only after the branch has finished executing. Counterfactual execution is similar to faceted execution [4], but rather than creating faceted values at control-flow merge points, we mark the values as indeterminate.

Self-Adjusting Computation The goal of *self-adjusting computation* [1] (a generalization of incremental computation [25]) is to incrementally update the result of a computation given changes to some of its inputs. This is achieved using a dynamic dependence graph tracking computations that depend on changeable inputs. Earlier systems for self-adjusting computation require programs to follow a particular programming style, whereas our analysis works on arbitrary programs. More recent systems [6, 7] automatically transform appropriately annotated programs into the required form, using an inference algorithm based on static information flow. Such a system could be used for static determinacy analysis of typed languages. Conversely, our analysis (which is similar to *dynamic* information flow) may itself be useful for self-adjusting computation.

Trace-Based Abstract Interpretation The instrumented traces in our formalization are similar to abstract derivation trees used in trace-based abstract interpretation [28] in that they describe a class of concrete executions. However, our traces are obtained by dynamic analysis, not static abstract interpretation. Our semantics does not yet cover infinite derivations and infinite traces.

Combinations of Static and Dynamic Analysis Several researchers have proposed using information gathered by a dynamic analysis to specialize uses of dynamic language constructs. Furr et al. [13] present a type inferencer for Ruby that uses information obtained during a profile run to replace dynamic features with statically analyzable alternatives. While their approach is technically similar to ours, their dynamic analysis is not sound, so a client has to account for cases where later runs do not match the profile. Bodden et al. [5] use a similar approach to specialize reflective Java programs, also without soundness guarantees.

Kneuss et al. [21] describe an (unsound) analysis for finding type errors in PHP that uses a flow-sensitive static analysis starting from a program state recorded by a dynamic analysis.

Dufour et al. [12] perform static analysis for Java based on a dynamic call graph, thus essentially analyzing only a single execution; this suffices for their application area of performance analysis. Wei et al. [31] apply the same approach to JavaScript, additionally using dynamic information to eliminate uses of `eval`. They argue that this approach, while unsound, can be useful for applications such as taint analysis where soundness is not always required.

Combining static and dynamic analysis the other way around, an unsound static analysis can be supplemented with additional runtime checks to catch cases not covered by the analysis. This technique has been used to enforce information flow policies [9, 14], and to infer type information for JIT compilation [16].

7. Conclusions

We have presented a purely dynamic analysis for identifying determinate expressions which always have the same value at a given program point. While the analysis only observes one program execution at a time, the facts it derives hold for *all* executions. We have proved the analysis correct for a simple imperative language, implemented it for full JavaScript, and shown how its results can be used by a static analysis to deal with reflective language features.

While our results are promising, more work is needed for the combined static-dynamic analysis to realize its full potential. To generate *more* determinacy facts, we plan to explore automated test generation [2] to improve coverage of the dynamic analysis. Running the determinacy analysis on different inputs yields more facts, which are all sound and hence can be used together. To generate *better* determinacy facts, we will work on inferring determinacy facts with shallower calling contexts. We will investigate whether these improvements are enough to make WALA scale to realistic web applications, or whether more work is needed. Finally, we also plan to apply determinacy analysis to other problems such as partial evaluation and dead code detection, and explore potential uses for counterfactual execution outside the setting of determinacy analysis, for instance in dynamic taint analysis.

References

1. Umut A. Acar. *Self-Adjusting Computation*. Ph.D. thesis, CMU, 2005.
2. Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A Framework for Automated Testing of JavaScript Web Applications. In *ICSE*, 2011.
3. Thomas H. Austin and Cormac Flanagan. Efficient Purely-Dynamic Information Flow Analysis. In *PLAS*, 2009.
4. Thomas H. Austin and Cormac Flanagan. Multiple Facets for Dynamic Information Flow. In *POPL*, 2012.
5. Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *ICSE*, 2011.
6. Yan Chen, Joshua Dunfield, and Umut A. Acar. Type-Directed Automatic Incrementalization. In *PLDI*, 2012.
7. Yan Chen, Joshua Dunfield, Matthew Hammer, and Umut Acar. Implicit Self-Adjusting Computation for Purely Functional Programs. In *ICFP*, 2011.
8. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *SAS*, 2003.
9. Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged Information Flow for JavaScript. In *PLDI*, 2009.
10. Charles Consel. Polyvariant Binding-Time Analysis For Applicative Languages. In *PEPM*, 1993.
11. Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.
12. Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended Analysis for Performance Understanding of Framework-based Applications. In *ISSTA*, 2007.
13. Michael Furr, Jong-hoon An, and Jeffrey S. Foster. Profile-guided Static Typing for Dynamic Scripting Languages. In *OOPSLA*, 2009.
14. Salvatore Guarnieri and V. Benjamin Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, 2009.
15. Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *ECOOP*, 2010.
16. Brian Hackett and Shu-yu Guo. Fast and Precise Hybrid Type Inference for JavaScript. In *PLDI*, 2012.
17. Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the Eval That Men Do. In *ISSTA*, 2012.
18. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *SAS*, 2009.
19. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.
20. James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7), July 1976.
21. Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Runtime Instrumentation for Precise Flow-Sensitive Type Analysis. In *RV*, 2010.
22. Xavier Leroy and Hervé Grall. Coinductive Big-Step Operational Semantics. *Inf. Comput.*, 207(2):284–304, 2009.
23. Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. Eval Begone! Semi-Automated Removal of Eval from JavaScript Programs. In *OOPSLA*, 2012.
24. Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.
25. Thomas W. Reps and Tim Teitelbaum. The Synthesizer Generator. In *SDE*, 1984.
26. Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The Eval That Men Do—A Large-Scale Study of the Use of Eval in JavaScript Applications. In *ECOOP*, 2011.
27. Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *PLDI*, 2010.
28. David A. Schmidt. Trace-Based Abstract Interpretation of Operational Semantics. *Lisp and Symbolic Computation*, 10(3):237–271, 1998.
29. O. Shivers. Control Flow Analysis in Scheme. In *PLDI*, 1988.
30. Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP*, 2012.
31. Shiyi Wei and Barbara G. Ryder. A Practical Blended Analysis for Dynamic Features in JavaScript. TR 12-18, Virginia Tech, 2012.
32. Steve Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.