JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2012; **00**:1–42
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Purity Analysis for JavaScript through Abstract Interpretation

Jens Nicolay*[1], Quentin Stiévenart[1], Wolfgang De Meuter[1], Coen De Roover[1]

[1] *Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium*
{jnicolay,qstieven,wdmeuter,cderoove}@vub.ac.be

SUMMARY

We present a static analysis for determining whether and to what extent functions in JavaScript programs are pure. To this end, the analysis classifies functions as either pure functions, observers, or procedures. A function is pure if none of its executions generate or depend upon externally observable side-effects. A function is an observer as soon as one of its executions depends on an external side-effect but none of its executions generate observable side-effects. Otherwise, the function is classified as a procedure. Function executions and associated callers are found by traversing all reachable function execution contexts on the call stack at the point where an effect occurs.
Our approach is based on a flow analysis that, in addition to computing traditional control and value flow, keeps track of read and write effects. To increase the precision of our purity analysis, we combine it with an intraprocedural analysis that determines freshness of variables and objects.
We formalize the core aspects of our technique, and discuss its implementation and results on common JavaScript benchmarks. Results show that our approach is capable of determining function purity in the presence of higher-order functions, dynamic property expressions, and prototypal inheritance. When compared to existing purity analyses, we find that our approach is as precise or more precise than the existing analyses.

Received . . .

KEY WORDS: purity analysis, side-effect analysis, freshness analysis, abstract interpretation, JavaScript

1. INTRODUCTION

Theoretically, the only observable effect of a function is turning input arguments into a resulting value. However, in imperative programming languages such as JavaScript, a callable entity (function, constructor, method . . .) can do more than that, such as reading and assigning variables, and loading and storing object properties. Anything a JavaScript function does besides producing a value, is called a side effect of that function. When a function modifies a resource reachable by a caller, the function *generates* an observable side effect. Side effects external to the function also play a role. When a function accesses a program resource other than one of its input arguments, and this resource is modified in between function executions, then that function *depends* on an external side effect.

This paper presents a static analysis that determines the purity of a function based on the side effects that function generates or depends upon during its executions. Our analysis classifies functions as either pure, observer, or procedure. A function is *pure* if none of its executions generate or depend upon externally observable side effects. A function is an *observer* as soon as one of its

*Correspondence to: Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium executions depends on an external side effect, but none of its executions generate observable side effects. We call all other functions *procedures*. Our analysis determines the observable extent of a side effect in terms of all functions that are in the process of computing a result when the effect occurs by traversing the stack [25]. We develop a formal definition of observability and purity in Sections 3 and 4.

There exist different definitions of purity, of which the strictest one corresponds to our classification of *pure* [13]. A weaker form of purity only requires that a function does not generate observable side effects [33], so that functions classified as *observers* by our analysis are considered to be pure as well. A function that is not pure is *impure*. Our notion of purity allows *unobservable* side effects, therefore allowing functions that allocate, access, and modify memory locations to still be pure. Additionally, we consider that pure functions may return locally allocated objects.

Research in different areas has demonstrated that purity aids program understanding, specification, optimization, testing, debugging, and maintenance [3, 13]. Therefore, detection, verification, and enforcement of purity is useful for software engineering purposes. Purity facilitates establishing security and confidentiality aspects of software applications. Pure functions are more secure than impure ones, or at least simplify security analysis, because they interfere less with the rest of the program [5, 9]. Purity also potentially reduces the number of bugs and makes it easier to reproduce them [13]. Pure functions can be safely called from assertions and contracts [4, 18], they allow for more and better program optimizations [7, 31], and they can speed up concurrency analyses by eliminating non-interfering interleavings. Purity analysis and any side-effect analysis it is based on also have immediate applications in program optimization, as they enable the detection of referentially transparent expressions and the application of memoization and parallelization in programs. We discuss these applications in more detail in Section 10. Proving the absence of side effects has many more advantages and applications that are discussed extensively in related work on purity and side-effect analysis (Section 5).

1.1. Challenges

Developing a static purity analysis for JavaScript is challenging in several ways. Some of these challenges are the result of general language properties and features such as object support and higher-order functions that JavaScript shares with other languages, and which makes the purity analysis we develop in this paper more broadly applicable than just JavaScript. Other challenges, such as constructor calls and other semantic features and corner cases, are specific to JavaScript.

1.1.1. Variable vs. object property effects A purity analysis for JavaScript must distinguish between effects on variables and effects on object properties. Variables denote a memory location containing a specific value. When a variable has an object or any other compound data structure as value, then two resources are involved on which effects may occur: the variable itself, and the object it has as a value. Consequently, effects on these resources have to be distinguished as well.

```
1  function f()
2  {
3    var o;
4    o = {x:3};   // variable write
5    o.x = 42     // property write
6  }
```

While it is possible to consider the entire accessed or modified object as the target of a property effect, our analysis takes the property name into account. Therefore, on line 5 in the above example the generated effect is not an “object write effect” on object `o` but a “property write effect” of property `x` on object `o`. This not only provides more detail when reporting about the presence of effects, but also improves precision when reasoning over interference and dependence based on side effects (e.g., Sections 1.1.5 and 10.3).

1.1.2. Precise control and value flow information Our purity analysis examines the observability of an effect in the context of every function execution that is active [16, 25]. The set of active execution contexts can be obtained by traversing the call stack. In the example program below, the property write effect on line 5 results in functions `h` (directly) and `g` (indirectly) mutating object `o` that exists in the caller state at their respective call sites (lines 6 and 4). Therefore the property write effect is observable to callers of `g` and `h`, and the analysis considers both functions to be procedures. Function `f` on the other hand *is* pure, because object `o` is not visible by callers of `f`, and therefore the property write effect on `o` is not observable to callers.

```
1  function f()                 // pure
2  {
3    var o={};
4    function g(p) { h(p) }    // procedure
5    function h(q) { q.x=4 }   // procedure
6    g(o)
7  }
8  f()
```

1.1.3. Higher-order functions While in the above example control flow is straightforward to determine, JavaScript features higher-order functions that complicate control flow and which any purity analysis must handle. In the following example, function `g` is passed as an argument to function `f`.

```
1  var z=0;
2  function g(p) { z=z+1; p.x=z}    // procedure
3  function f(h) { var o={}; h(o)}  // procedure
4  f(g)
```

It is clear that function `g` is a procedure because it writes to global property `z` and mutates an object through its parameter. However, a purity analysis must determine that the application of function `h` in the body of `f` on line 3 executes function `g`. It therefore must classify function `f` also as a procedure, because `f` indirectly mutates `z`.

1.1.4. Closures JavaScript has closures and, unlike purity analyses for more traditional object-oriented languages without closures, a purity analysis for JavaScript must handle free variables.

```
1  function f()                   // pure
2  {
3    function g()                 // pure
4    {
5      var z = 10;
6      function h() { z = 20 };   // procedure
7      h()
8    }
9    g()
10 }
11 f()
```

In the example above, when `z` is assigned on line 6, the call stack consists of function executions of `h`, `g`, and `f`. Function `h` is a procedure because it mutates `z`, which is a free variable of `h` and therefore exists in the caller state in this example. Function `g` is pure, because `z` is local to `g`. Function `f` is also pure, because `z` is not part of its scope.

1.1.5. Finer-grained classification scheme Our analysis classifies functions that depend on external side effects as observers. To determine which functions are observers, the analysis must detect resources that are read by functions and that are modified in between two function executions.

```
var o = {x:123};
var p = Object.create(o);
function f()               // observer
{
  return p.x
}
f();
o.x = 456;
f()
```

In the above example, object `o` defines a property `x` (line 1) and object `p` inherits this property through its prototype `o` (line 2). Function `f` is applied twice, with a mutation of property `o.x` in between the two executions. Function `f` is an observer because its resulting value depends on the value of an object property that is mutated in between two executions of `f`.

This example also illustrates the usefulness of taking the property name into account when object property effects are involved. If the property read by `f` on line 5 would be different from the property mutated on line 8, then `f` would not depend on an external side effect and be considered pure in this example.

1.1.6. Semantic corner cases In JavaScript it is not always straightforward to determine the correct side effects. Because of the semantics of the language, variable `x` in the example program below is actually a property of the global object.

```
1  var x;
2  this.x = 10; // property write
3  function f()
4  {
5    var y;
6    x = 10;     // property write
7    y = 20;     // variable write
8  }
```

Assigning to `x` on line 6 must generate a property write effect on the global object, identical to the effect generated on line 2, and not a variable write effect.

Another example of non-obvious effects happens when assigning an array index that is equal or greater than the current length of the array. This generates an additional write effect on the `length` property of that array.

1.1.7. Constructor vs. function calls A final challenge that we mention concerns the fact that functions in JavaScript are also constructors when invoked through `new`.

```
1  function f() { this.x = 10 }  // procedure
2  new f();                      // no observable side effects
3  f();                          // generates observable write effect
```

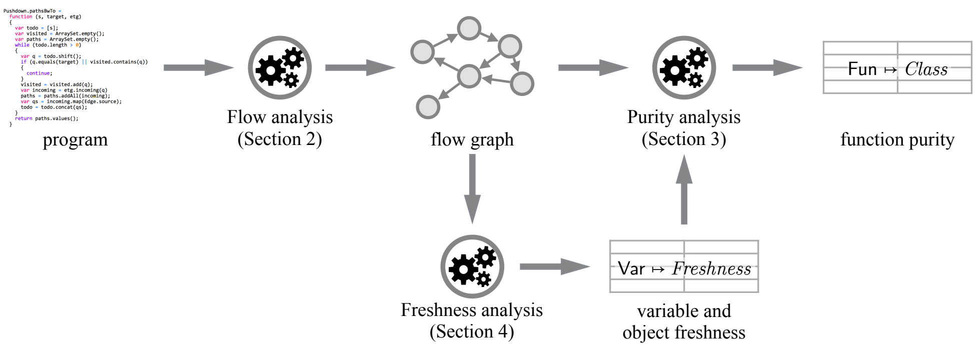
When function `f` in the above example is invoked as a constructor (line 2), then this results in a pure function execution because `this` in the body of `f` is bound to a freshly allocated object. The regular function application on the next line results in an impure execution of `f` because `this` is bound to the global object during execution. As a result, function `f` is a procedure.

1.2. Overview of Our Approach

Our approach for designing a static purity analysis, depicted in Figure 1, is based on an abstract interpretation [8] of the program that integrates control flow, value flow, and effects. Abstract interpretation is a good match for static analysis, because it provides a theory to develop static analyses that are correct and provide meaningful results about the runtime behavior of programs.

We define a core imperative language, JS_0, that represents an interesting and non-trivial subset of standard JavaScript semantics (Section 2). The semantics of this input language is expressed as an abstract state machine that is instrumented to register read and write effects on resources. Resources in our semantics are variables and objects, which are allocated in a store at specific addresses. Starting from an initial evaluation state, all possible successor states are explored. The result of state exploration is a flow graph, which is a representation of the runtime behavior of the input program in terms of control flow, value flow, and effects.

Our purity analysis then examines the flow graph to determine whether a given function can be classified as pure, observer, or procedure by looking at all effects during and in between function executions (Section 3). For every effect, the analysis traverses the call stack to obtain all active function executions in the program state associated with the effect. When in the context of a function execution an observable write effect is detected, then the applied function is labeled a procedure. For dependence on external effects, the purity analysis looks for read–write–read sequences in the



```
Pushdown.pathsBwTo =
  function (s, target, etg)
  {
    var todo = [s];
    var visited = ArraySet.empty();
    var paths = ArraySet.empty();
    while (todo.length > 0)
    {
      var q = todo.shift();
      if (q.equals(target) || visited.contains(q))
      {
        continue;
      }
      visited = visited.add(q);
      var incoming = etg.incoming(q)
      paths = paths.addAll(incoming);
      var qs = incoming.map(Edge.source);
      todo = todo.concat(qs);
    }
    return paths.values();
  }
```

Figure 1. Overview of our approach.

program involving the same resource, with the initial and subsequent read occurring in executions of the same function.

Effects are observable to a caller when the address of an accessed or modified resource is mapped in the caller store that is in effect at function entry. Although checking addresses of resources for determining observability is attractive because it is a close fit with the store semantics of the abstract machine, we find that this technique is problematic in a static analysis setting: because an analysis must complete in finite space and time, the set of addresses is made finite, which decreases the precision of address-based purity analysis. We therefore attempt to recover some of this precision by defining a lexical and intraprocedural freshness analysis on top of a flow graph (Section 4). A variable that is created in the function scope of an executing function is fresh in that execution context, while an object reference is fresh in a function execution if the referenced object was allocated during that execution. The purity analysis can use freshness of variables and objects to mask certain read and write effects, thereby increasing the precision of the purity analysis.

After explaining our approach in detail, we give an overview of related work (Section 5). We then discuss our implementation (Section 6), work out an example of each of the analyses presented in this paper (Section 7), and describe the setup and outcome of our experiments (Section 8). The remainder of the paper presents limitations (Section 9) and applications (Section 10) of our purity analysis.

The contributions presented in this paper are the following:

- We present an abstract machine for a core JavaScript-like language that tracks read and write effects generated by accessing and modifying variables and object properties during interpretation.
- We introduce a purity analysis over a flow graph annotated with effects.
- We define a lexical freshness analysis for variables and objects to improve the precision of our purity analysis.
- We implement our purity analysis for a substantial subset of JavaScript and experiment with it on several JavaScript benchmarks. We find that the analysis is capable of determining function purity with sufficient precision to be useful to client applications.

This paper is an extended version of "Detecting Function Purity in JavaScript" [27], published at the 2015 Working Conference on Source Code Analysis and Manipulation (SCAM'15). It differs from the earlier version in the following key aspects:

- The abstract machine for our core JavaScript-like language now tracks read effects in addition to write effects.
- Tracking read effects enables our purity analysis to consider "observers" as an additional purity class in its classification of functions. Consequently, the reported results are more fine-grained than in our earlier work, which in turn allows us to map these results onto different definitions of purity that exist in literature.
- We simplified the formulation of the freshness analysis incorporated by our purity analysis, and explicitly limit it to the most recent function application (the running execution context).

This simplification improves running times, yet our experiments showed that this incurred no precision loss.

- We increased the number of benchmarks to which we apply our analysis from 7 to 10. We also compare our experimental results to three existing approaches for determining function purity.

2. FLOW ANALYSIS

To reason about runtime program properties, a static analysis models the runtime behavior of a program and queries this model for the properties of interest. In this section we introduce a core JavaScript language, JS_0, and a flow analysis that models the execution of a JS_0 program as a flow graph from which information about control flow, value flow, and effects can be extracted. JS_0 is a higher-order, functional, and object-oriented language. Because function call and return is the dominant control-flow pattern in higher-order, functional programs, the flow analysis models call/return as a pushdown analysis [37, 20] instead of a more classic but less precise finite-state analysis.

Notation and Conventions We use $\uplus$ to denote *disjoint union*: if $X = Y \uplus Z$, then $Y = X \setminus Z$. The notation $X = x : X'$ deconstructs a *sequence* X into its first element x and the rest X'. We write $\langle\rangle$ for the empty sequence. The *power domain* of set X is denoted as $\mathcal{P}(X)$. The *empty function* is denoted as $[]$, and for all inputs returns the bottom element $\bot$ of its range. The notation $f[x \mapsto y]$ denotes *function extension* and yields a function f' such that:

$$f'(z) = \begin{cases} y & \text{if } z = x, \\ f(z) & \text{else.} \end{cases}$$

We write the *function restriction* (or narrowing) of a function f to domain X as $f|X$, such that $(f|X)(x) = f(x)$ if $x \in X$ and $(f|X)(x) = \bot$ else. *Function joining* happens in a pointwise fashion. If $\sqcup$ is the join operator for the range of the function, then $[x \mapsto y_1] \sqcup [x \mapsto y_2] = [x \mapsto y_1 \sqcup y_2]$. In particular, $\bigsqcup\{[x_0 \mapsto y_0], \ldots, [x_n \mapsto y_n]\} = [x_0 \mapsto y_0] \sqcup \ldots \sqcup [x_n \mapsto y_n]$. When a function is defined using several (numbered) *cases*, then it is assumed that the cases are considered in order and do not fall through.

2.1. *Input Language:* JS_0

To simplify the formalization of our analysis, we work on a core functional language JS_0 with assignment and mutable objects. This input language, depicted in Figure 2, most notably features objects as maps, higher-order functions, prototypal inheritance, and assignment. Although JS_0 is a small language, its set of features is sufficiently representative and challenging for performing purity analysis. In Section 6.1 we describe a number of extensions to JS_0 such as primitives and conditionals. Our implementation (Section 6), which we used to validate our approach, supports an even larger subset of traditional features of JavaScript such as variable declarations, iteration, non-local return flow, and other typical features of JavaScript, including type coercions and parts of the standard built-in functions and objects. We assume that every element in JS_0 is uniquely labeled so that different occurrences of the same expression can be distinguished.

2.2. *Semantics of* JS_0

The small-step semantics of the input language is expressed as an abstract machine [12] that transitions between states.

The resulting machine is a variation on the CESIK$^\star\Xi$ abstract machine described in Johnson and Van Horn [20]. This machine actually is an *abstract* abstract machine because it operates on abstract values, although it can be parameterized to express concrete semantics.

$$
\begin{array}{rcll}
e \in \mathsf{Exp} & ::= & s & \text{[simple expr]} \\
& | & f & \text{[function]} \\
& | & v\,\texttt{(}s\texttt{)} & \text{[function call]} \\
& | & s_0\,\texttt{.}\,v\,\texttt{(}s_1\texttt{)} & \text{[method call]} \\
& | & \texttt{new}\ v\,\texttt{(}s\texttt{)} & \text{[new expr]} \\
& | & v\texttt{=}e & \text{[assignment]} \\
& | & s\,\texttt{.}\,v & \text{[property load]} \\
& | & s\,\texttt{.}\,v\texttt{=}e & \text{[property store]} \\
& | & \texttt{return}\ s & \text{[return]} \\
s \in \mathsf{Simple} & ::= & v & \text{[reference]} \\
& | & \texttt{this} & \text{[this expr]} \\
f \in \mathsf{Fun} & ::= & \texttt{function (}v\texttt{)}\,\{\texttt{var}\ v_h\texttt{;}\ e\} & \\
v \in \mathsf{Var} & = & \text{a set of identifiers} &
\end{array}
$$

Figure 2. Input language JS$_0$.

$$
\begin{array}{rcll}
\varsigma \in \mathit{State} & ::= & \mathbf{ev}(e,\rho,\sigma,\iota,\kappa,\Xi) & \text{[eval state]} \\
& | & \mathbf{ko}(d,\sigma,\iota,\kappa,\Xi) & \text{[kont state]} \\
\rho \in \mathit{Env} & = & \mathsf{Var} \rightharpoonup \mathit{Addr} & \text{[environment]} \\
\sigma \in \mathit{Store} & = & \mathit{Addr} \rightharpoonup (D + \mathit{Obj}) & \text{[store]} \\
d \in D & = & \mathcal{P}(\mathit{Addr} + \mathit{Prim}) & \text{[value]} \\
\mathit{Prim} & = & \{\mathbf{undef}\} & \text{[primitive value]} \\
\omega \in \mathit{Obj} & = & (\mathsf{Var} \rightharpoonup D) \times (\text{“proto”} \mapsto D) \times (\text{“call”} \mapsto \mathcal{P}(\mathit{Callable})) & \text{[object]} \\
c \in \mathit{Callable} & ::= & (f,\rho) & \text{[callable]} \\
\iota \in \mathit{LKont} & = & \mathit{Frame}^* & \text{[frame]} \\
\phi \in \mathit{Frame} & ::= & \mathbf{as}(v,\rho) & \text{[assignment frame]} \\
& | & \mathbf{st}(s,v,\rho) & \text{[property store frame]} \\
\kappa \in \mathit{Kont} & ::= & \epsilon \mid \tau & \text{[meta-continuation]} \\
\tau \in \mathit{Ctx} & ::= & (e,c,d_{\text{arg}},a_{\text{this}},\sigma) & \text{[execution context]} \\
\Xi \in \mathit{KStore} & = & \mathit{Kont} \rightharpoonup \mathcal{P}(\mathit{LKont} \times \mathit{Kont}) & \text{[stack store]} \\
a \in \mathit{Addr} & & \text{is a set of addresses} & \text{[address]} \\
\mathit{eff} \in \mathit{Eff} & ::= & \mathbf{Wv}(a,v) & \text{[variable write effect]} \\
& | & \mathbf{Wp}(a,v) & \text{[property write effect]} \\
& | & \mathbf{Rv}(a,v) & \text{[variable read effect]} \\
& | & \mathbf{Rp}(a,v) & \text{[property read effect]} \\
E \in \mathcal{P}(\mathit{Eff}) & & \text{is a set of effects} & \text{[effects]}
\end{array}
$$

Figure 3. State-space of the abstract machine semantics.

Figure 3 shows the abstract state-space of the machine. In the remainder of this section we detail the components and the operation of the abstract machine.

2.2.1. States A machine state is either an evaluation state ($\mathbf{ev}$), or a continuation state ($\mathbf{ko}$). In an evaluation state, the machine evaluates an expression e in environment ρ. In a continuation state, the machine is ready to continue evaluation with a value d it has just computed.

2.2.2. Environment and Store An environment ρ maps variables to addresses. Addresses represent a location in the store σ, and the store maps addresses to values. Looking up the value of a variable therefore consists of first looking up that variable's address, and then referencing this address in the store.

2.2.3. Values Values in our semantics are addresses (a) and the `undefined` value ($\mathbf{undef}$). An address is a pointer to an object. Objects (ω) are represented as maps from properties to values. Properties "call" and "prototype" are two special properties that are distinct from all other properties, and are used to implement function objects and object prototypes, respectively.

2.2.4. Stacks The stack is modeled as a local continuation (ι) delimited by a meta-continuation (κ). The local continuation is a (possibly empty) list of frames, and acts like a "regular" stack, on top of which frames can be pushed and from which topmost frames can be popped. The local stack in $\mathtt{JS}_0$ contains pending variable assignments and property stores in the local function.

The meta-continuation is either empty (ϵ), or a stack address (τ) pointing to underlying stacks stored in a stack store (Ξ). The meta-continuation serves two purposes:

1. Meta-continuations serve as *stack addresses* pointing to underlying caller stacks, which are stored in a stack store (Ξ).
2. Meta-continuations play the role of *execution contexts* that are generated at call sites (except for the root context ϵ that is created at the start of program evaluation). The meta-continuation stored inside a program state is the execution context on top of the stack and is called the *running execution context*.

The result of indirection introduced by meta-continuations is an abstraction of a linear stack into a graph. Nodes in this graph are combinations of a local continuation with a meta-continuation, while edges indicate reachability as encoded in the stack store. Traversing the stack from top to bottom in a program state happens by first traversing the local continuation ι of that state (if not empty), and then looking up the underlying stacks by following outgoing edges. This process repeats until the root context ϵ is encountered. A single stack address can have more than one underlying stack, represented by multiple outgoing edges in the stack graph, indicating a loss of control-flow precision. A stack address may also directly or transitively be reachable from itself, forming loops in the stack graph. Traversing the stack therefore always involves keeping a set of visited stack nodes to avoid infinite recursion.

The graph representing a (potentially unbounded) stack is guaranteed to be finite when stack frames and stack addresses are selected from finite sets. For our abstract machine semantics we use the AAC stack address allocator [20], generating stack addresses that contain five components:

1. the call expression (e) that is involved,
2. the callable (c) that is invoked,
3. the argument (d_{arg}) to the function application,
4. the `this` pointer (a_{this}) in the context of the call,
5. and the caller store (σ) that is in effect at function entry.

It can be verified by inspection of the state space (Figure 3) that this stack allocation scheme is finite when the set of syntactic variables (Var) and the set of value store addresses (Addr) is finite.

These five components also completely determine the outcome of a function execution. Therefore, this stack allocation scheme results in maximally attainable call/return precision under any given abstraction of the components that make up stack addresses. Under concrete semantics, full call/return precision is preserved and the graph representing the stack of a terminating program is a linked list.

2.2.5. Effects Four types of effects are tracked: reading and writing of variables ($\mathbf{Rv}$, $\mathbf{Wv}$), and reading and writing of object properties ($\mathbf{Rp}$, $\mathbf{Wp}$). They are generated by transitions between states.

2.2.6. Program Injection The injection function $\mathcal{I} : \mathsf{Exp} \to \mathit{State}$ turns an expression into an initial evaluation state with empty environment, initial store, empty local continuation, and the root context as meta-continuation.

$$\begin{aligned}\mathcal{I}(e) &= \mathbf{ev}(e, [], \sigma_0, \langle\rangle, \epsilon, [])\\ \text{where } \kappa_0 &= (e, \bot, \bot, a_0, \sigma_0)\\ \sigma_0 &= [a_0 \mapsto []]\end{aligned}$$

The initial store σ_0 maps the global object at address a_0, which we assume to be globally available.

2.2.7. Address Allocation Address allocation is a parameter of the semantics that can be used to control the context-sensitivity of the resulting analysis. Any address allocation scheme is sound [24], although to simplify the semantics we assume that addresses for objects (in Obj) are distinct from addresses for all other values (in D). We assume the presence of allocation functions $\mathit{allocVar}$ for allocating variables, $\mathit{allocCtr}$ for allocating constructor objects, $\mathit{allocFun}$ for allocating function objects, and $\mathit{allocProto}$ for allocating prototypes of function objects.

To express concrete semantics, we can take $\mathit{Addr} = \mathbb{N}$ and

$$\mathit{allocX}(e, \rho, \sigma, \iota, \kappa) = 1 + \max(\mathrm{Dom}(\sigma)),$$

where allocX is one of the allocation functions and Dom returns the domain of a function.

For abstract semantics, a monovariant allocation scheme that generates addresses based on allocation sites (0CFA) would be $\mathit{Addr} = \mathsf{Exp}$ with

$$\mathit{allocVar}(e, \rho, \sigma, \iota, \kappa) = e$$

and similar definitions for the other allocators.

2.2.8. Simple Expressions Function $\mathit{evalSimple} : \mathsf{Simple} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} \mapsto D \times \mathcal{P}(\mathit{Eff})$ evaluates simple expressions: references and `this` expression. It not only returns the resulting value, but also the set of generated effects. The following three cases define $\mathit{evalSimple}$.

1. To evaluate a variable reference v, first the address of the variable is looked up in the lexical environment. If the variable is available in the environment, then the value associated with its address a in the store is returned and a variable read effect $\mathbf{Rv}(a, v)$ is generated.

$$\begin{aligned}\mathit{evalSimple}(v, \rho, \sigma, \kappa) &= (\sigma(a), \{\mathbf{Rv}(a, v)\})\\ &\text{if } v \in \mathrm{Dom}(\rho)\\ &\text{where } a = \rho(v)\end{aligned}$$

2. If a variable v is not available in the environment, a property lookup is performed on the global object at address a_0, and a property read effect $\mathbf{Rp}(a_0, v)$ is generated.

$$\begin{aligned}\mathit{evalSimple}(v, \rho, \sigma, \kappa) &= (\omega(v), \{\mathbf{Rp}(a_0, v)\})\\ &\text{where } \omega = \sigma(a_0)\end{aligned}$$

3. The value for a `this` expression is retrieved from the running execution context, and no effects are generated.

$$\mathit{evalSimple}([\![\mathtt{this}]\!], \rho, \sigma, (e, c, d_{\text{arg}}, a_{\text{this}}, \sigma)) = (\{a_{\text{this}}\}, \varnothing)$$

2.2.9. Property Lookup Relation *lookupProp* looks up a property by traversing the prototype chain of an object. If the property is not found in the chain, `undefined` is returned.

$$\begin{aligned}
&\mathit{lookupProp}(v, a, \sigma)\\
&\qquad = \begin{cases} (\omega(v), \{\mathbf{Rp}(a, v)\}) & \text{if } v \in \mathrm{Dom}(\omega)\\ (\{\mathbf{undef}\}, \varnothing) & \text{if } \omega(\text{“proto”}) = \varnothing\\ \mathit{lookupProp}(v, a', \sigma) & \text{else}\end{cases}\\
&\quad \text{where } \omega = \sigma(a)\\
&\qquad\quad a' \in \omega(\text{“proto”})
\end{aligned}$$

2.2.10. Function Call Function *evalCall* applies a callable (or closure) (f, ρ) to an argument d_{arg} in a certain program state. It extends the function's static environment ρ and the store σ by binding parameters to their argument values and local variables to `undefined`. To simplify the semantics, we assume a single parameter and a single local variable declaration that is hoisted to the beginning of the function scope (see Section 6.1.5 for extensions).

Parameter κ is the execution context of the caller, while parameter κ' represents the execution context for the call itself. The stack store Ξ is extended by allocating the caller stack (ι, κ) at stack address κ'. Evaluation of the body of the function happens in the static environment and store extended with the binding of the argument (ρ' and σ'), with an empty local continuation $\langle\rangle$, and the execution context for the call κ'.

$$\begin{aligned}
\mathit{evalCall}((f, \rho), d_{\text{arg}}, \sigma, \iota, \kappa, \Xi, \kappa') &= \mathbf{ev}(e, \rho', \sigma', \langle\rangle, \kappa', \Xi')\\
\text{where } f &= [\![\texttt{function (}v\texttt{)}\,\{\texttt{var } v_h\texttt{; } e\}]\!]\\
\rho' &= \rho[v \mapsto a]\\
\sigma' &= \sigma \sqcup [a \mapsto d_{\text{arg}}, a_h \mapsto \{\mathbf{undef}\}]\\
a &= \mathit{allocVar}(v, \rho, \sigma, \iota, \kappa)\\
a_h &= \mathit{allocVar}(v_h, \rho, \sigma, \iota, \kappa)\\
\Xi' &= \Xi \sqcup [\kappa' \mapsto \{(\iota, \kappa)\}]
\end{aligned}$$

2.2.11. Transition Relation Using the auxiliary functions from previous sections, we can now define the transition relation $\mapsto$ of our abstract machine. In order to determine function purity, we must be able to reason about effects that occur as a result of reading and modifying variables and object properties during evaluation. We therefore make read and write effects explicit by modeling them on the transition relation that transitions between states.

$$(\mapsto) \sqsubseteq \mathit{State} \times \mathit{State} \times \mathcal{P}(\mathit{Eff})$$

Rules for transitions from evaluation states (**ev**) correspond with the different syntactic cases, while rules for transitions from continuation states (**ko**) correspond with the different kinds of continuations. We list the different cases for $\mapsto$.

1. A simple expression is evaluated by delegating to *evalSimple*.

$$\begin{aligned}
\mathbf{ev}(s, \rho, \sigma, \iota, \kappa, \Xi) &\mapsto (\mathbf{ko}(d, \sigma, \iota, \kappa, \Xi), E)\\
\text{where } (d, E) &= \mathit{evalSimple}(s, \rho, \sigma, \kappa)
\end{aligned}$$

2. Evaluating a function expression yields a reference to a function object (ω_f) that is allocated in the store. Following JavaScript semantics, a function object has a fresh object assigned to

its `prototype` property.

$$\begin{aligned}
\mathbf{ev}(\llbracket \overbrace{\texttt{function (}v\texttt{)} \{\texttt{var } v_h\texttt{; } e\}}^{f} \rrbracket, \rho, \sigma, \iota, \kappa, \Xi) \mapsto (\mathbf{ko}(\{a\}, \sigma', \iota, \kappa, \Xi), \varnothing)& \\
\text{where } a &= \mathit{allocFun}(f, \rho, \sigma, \iota, \kappa) \\
a' &= \mathit{allocProto}(f, \rho, \sigma, \iota, \kappa) \\
\sigma' &= \sigma \sqcup [a \mapsto \{\omega_f\}, a' \mapsto \{\omega_{\text{proto}}\}] \\
\omega_f &= [\text{“call”} \mapsto \{(f, \rho)\}, \\
&\qquad \text{“proto”} \mapsto \varnothing, \\
&\qquad \texttt{prototype} \mapsto \{a'\}] \\
\omega_{\text{proto}} &= [\text{“proto”} \mapsto \varnothing]
\end{aligned}$$

3. A function call is evaluated by first evaluating operator and argument, and then applying the *evalCall* helper function with a reference to the global object (a_0) as `this` value.

$$\begin{aligned}
\mathbf{ev}(\llbracket \overbrace{v\ \texttt{(}s\texttt{)}}^{e} \rrbracket, \rho, \sigma, \iota, \kappa, \Xi) \mapsto (\mathit{evalCall}(c, d_{\text{arg}}, \sigma, \iota, \kappa, \Xi, \kappa'), E)& \\
\text{where } (d_f, E_f) &= \mathit{evalSimple}(v, \rho, \sigma, \kappa) \\
(d_{\text{arg}}, E_{\text{arg}}) &= \mathit{evalSimple}(s, \rho, \sigma, \kappa) \\
a_f &\in d_f \\
\omega_f &= \sigma(a_f) \\
c &\in \omega_f(\text{“call”}) \\
\kappa' &= (e, c, d_{\text{arg}}, a_0, \sigma) \\
E &= E_f \cup E_{\text{arg}}
\end{aligned}$$

4. For a method call additionally the method is looked up in the receiver, and the receiver is set as value for `this` in the new execution context.

$$\begin{aligned}
\mathbf{ev}(\llbracket \overbrace{s_0\ \texttt{.}\ v\ \texttt{(}s_1\texttt{)}}^{e} \rrbracket, \rho, \sigma, \iota, \kappa, \Xi) \mapsto (\mathit{evalCall}(c, d_{\text{arg}}, \sigma, \iota, \kappa, \Xi, \kappa'), E)& \\
\text{where } (d_{\text{this}}, E_{\text{this}}) &= \mathit{evalSimple}(s_0, \rho, \sigma, \kappa) \\
(d_{\text{arg}}, E_{\text{arg}}) &= \mathit{evalSimple}(s_1, \rho, \sigma, \kappa) \\
a_{\text{this}} &\in d_{\text{this}} \\
(d_f, E_f) &\in \mathit{lookupProp}(v, a_{\text{this}}, \sigma) \\
a_f &\in d_f \\
\omega_f &= \sigma(a_f) \\
c &\in \omega_f(\text{“call”}) \\
\kappa' &= (e, c, d_{\text{arg}}, a_{\text{this}}, \sigma) \\
E &= E_{\text{this}} \cup E_{\text{arg}} \cup E_f
\end{aligned}$$

5. A constructor call allocates a new object on the heap, and sets a reference to this object as value for `this` in the new execution context. The internal prototype of the new object is the value of the `prototype` property of the invoked constructor. The caller store in the context

is the store *without* the newly created object.

$$\begin{aligned}
\mathbf{ev}(\llbracket\overbrace{\texttt{new } v\,(s)}^{e}\rrbracket, \rho, \sigma, \iota, \kappa, \Xi) &\mapsto (\mathit{evalCall}(c, d_{\text{arg}}, \sigma', \iota, \kappa, \Xi, \kappa'), E)\\
\text{where } (d_f, E_f) &= \mathit{evalSimple}(v, \rho, \sigma, \kappa)\\
(d_{\text{arg}}, E_{\text{arg}}) &= \mathit{evalSimple}(s, \rho, \sigma, \kappa)\\
a_f &\in d_f\\
\omega_f &= \sigma(a_f)\\
c &\in \omega_f(\text{“call”})\\
a_{\text{this}} &= \mathit{allocCtr}(e, \rho, \sigma, \iota, \kappa)\\
\omega &= [\text{“proto”} \mapsto \omega_f(\texttt{prototype})]\\
\sigma' &= \sigma \sqcup [a_{\text{this}} \mapsto \{\omega\}]\\
\kappa' &= (e, c, d_{\text{arg}}, a_{\text{this}}, \sigma)\\
E &= E_f \cup E_{\text{arg}}
\end{aligned}$$

6. Variable assignment pushes a continuation to assign the value of the right hand side to the variable. No effects are generated during this step.

$$\begin{aligned}
\mathbf{ev}(\llbracket v{=}e\rrbracket, \rho, \sigma, \iota, \kappa, \Xi) &\mapsto (\mathbf{ev}(e, \rho, \sigma, \phi : \iota, \kappa, \Xi), \varnothing)\\
\text{where } \phi &= \mathbf{as}(v, \rho)
\end{aligned}$$

7. Loading a property involves evaluating the receiver, and looking up the property in that receiver.

$$\begin{aligned}
\mathbf{ev}(\llbracket s\,\texttt{.}\,v\rrbracket, \rho, \sigma, \iota, \kappa, \Xi) &\mapsto (\mathbf{ko}(d, \sigma, \iota, \kappa, \Xi), E)\\
\text{where } (d_r, E_r) &= \mathit{evalSimple}(s, \rho, \sigma, \kappa)\\
a &\in d_r\\
(d, E') &\in \mathit{lookupProp}(v, a, \sigma)\\
E &= E_r \cup E'
\end{aligned}$$

8. Like assignment, storing a property requires evaluating the right hand side and pushing a continuation to perform the actual property update.

$$\begin{aligned}
\mathbf{ev}(\llbracket s\,\texttt{.}\,v{=}e\rrbracket, \rho, \sigma, \iota, \kappa, \Xi) &\mapsto (\mathbf{ev}(e, \rho, \sigma, \phi : \iota, \kappa, \Xi), \varnothing)\\
\text{where } \phi &= \mathbf{st}(s, v, \rho)
\end{aligned}$$

9. Function return computes a return value and clears the local continuation.

$$\begin{aligned}
\mathbf{ev}(\llbracket \texttt{return } s\rrbracket, \rho, \sigma, \iota, \kappa, \Xi) &\mapsto (\mathbf{ko}(d, \rho, \sigma, \langle\rangle, \kappa, \Xi), E)\\
\text{where } (d, E) &= \mathit{evalSimple}(s, \rho, \sigma, \kappa)
\end{aligned}$$

10. When the machine has to continue with an assignment frame on top of the stack, it assigns the value computed for the right hand side to the variable on the left, if the variable is in scope. It then continues with this value, generating a variable write effect.

$$\begin{aligned}
\mathbf{ko}(d, \sigma, \mathbf{as}(v, \rho) : \iota, \kappa, \Xi) &\mapsto (\mathbf{ko}(d, \sigma', \iota, \kappa, \Xi), E)\\
&\quad \text{if } v \in \mathrm{Dom}(\rho)\\
\text{where } a &= \rho(v)\\
\sigma' &= \sigma \sqcup [a \mapsto d]\\
E &= \{\mathbf{Wv}(a, v)\}
\end{aligned}$$

11. If the variable is not found in the environment, the machine performs a property update on the global object, generating a property write effect.

$$\begin{aligned}\mathbf{ko}(d, \sigma, \mathbf{as}(v, \rho) : \iota, \kappa, \Xi) &\mapsto (\mathbf{ko}(d, \sigma', \iota, \kappa, \Xi), E)\\ \text{where } \omega &= \sigma(a_0)[v \mapsto d]\\ \sigma' &= \sigma \sqcup [a_0 \mapsto \omega]\\ E &= \{\mathbf{Wp}(a_0, v)\}\end{aligned}$$

12. Storing a property always happens directly on the receiver and does not require traversing prototype links. It generates a property write effect.

$$\begin{aligned}\mathbf{ko}(d, \sigma, \mathbf{st}(s, v, \rho) : \iota, \kappa, \Xi) &\mapsto (\mathbf{ko}(d, \sigma', \iota, \kappa, \Xi), E)\\ \text{where } (d_r, E_r) &= \mathit{evalSimple}(s, \rho, \sigma, \kappa)\\ a &\in d_r\\ \omega &= \sigma(a)[v \mapsto d]\\ \sigma' &= \sigma \sqcup [a \mapsto \omega]\\ E &= E_r \cup \{\mathbf{Wp}(a, v)\}\end{aligned}$$

13. When the machine reaches a state with an empty local continuation, the machine dereferences the stack address to obtain an underlying stack. If no stacks are found in the stack store, then the machine has reached a program exit and halts, and the current value is the result value of the program. Else, the machine has reached a function exit, which is the consequence of either an explicit `return`, or of an implicit return by reaching the end of a function body. When exiting a constructor call, a reference to the newly created object is returned.

$$\begin{aligned}\mathbf{ko}(d, \sigma, \langle\rangle, \kappa, \Xi) &\mapsto (\mathbf{ko}(d', \sigma, \iota', \kappa', \Xi), \varnothing)\\ \text{where } (\iota', \kappa') &\in \Xi(\kappa)\\ d' &= \{a_{\text{this}}\}\\ ([\![\texttt{new}\ v\ \texttt{(}s\texttt{)}\,]\!], \_, \_, a_{\text{this}}, \_) &= \kappa\end{aligned}$$

14. When exiting a regular function call, the current value is returned.

$$\begin{aligned}\mathbf{ko}(d, \sigma, \langle\rangle, \kappa, \Xi) &\mapsto (\mathbf{ko}(d, \sigma, \iota', \kappa', \Xi), \varnothing)\\ \text{where } (\iota', \kappa') &\in \Xi(\kappa)\end{aligned}$$

2.2.12. Flow Graph We determine function purity by reasoning about effects that happen during program evaluation. We therefore construct a *flow graph* representing program evaluation, in which nodes are reachable states, and edges are transitions between states that are labeled with the effects that occur on transition. Let $\hookrightarrow$ be transition relation $\mapsto$ with the effects removed:

$$\varsigma \hookrightarrow \varsigma' \iff \varsigma \mapsto (\varsigma', E).$$

Let ς_0 be the initial state for expression e:

$$\varsigma_0 = \mathcal{I}(e).$$

Evaluation of an expression e can be expressed as computing the transitive closure of $\hookrightarrow$ after injection.

$$\mathcal{E}(e) = \{\varsigma \mid \varsigma_0 \hookrightarrow^* \varsigma'\}$$

The definition of flow graph G_e for expression e then is as follows:

$$\varsigma \xrightarrow{E} \varsigma' \in G_e \iff \varsigma \in \mathcal{E}(e) \text{ and } \varsigma \mapsto (\varsigma', E)$$

Static analysis requires a finite flow graph for every possible program. We can guarantee finiteness by plugging in finite sets for Var and Addr into the state-space of the analysis (Figure 3). When analyzing a finite program (finite Var) using a monovariant allocation policy (finite Addr), then the entire state space is finite as well and $\hookrightarrow$, which is monotonic, has a least fixpoint.

3. PURITY ANALYSIS

Using the flow graph from the previous section, our analysis determines function purity by examining all effects in the context of all function executions. The analysis traverses all states (Section 3.1), and for every effect in a state it traverses the call stack (Section 3.2) to determine in which execution contexts the effect is observable. The result of the purity analysis is a map P from functions to their effect class.

$$\begin{aligned}
class \in Class &= \{\mathsf{pure}, \mathsf{obs}, \mathsf{proc}\} && \text{[effect class]}\\
P \in Purity &= \mathsf{Fun} \mapsto Class && \text{[function purity]}\\
R \in Read &= Res \mapsto \mathcal{P}(\mathsf{Fun}) && \text{[read table]}\\
O \in Obs &= Res \mapsto \mathcal{P}(\mathsf{Fun}) && \text{[observer table]}\\
res \in Res &= Addr + (Addr \times \mathsf{Var}) && \text{[resource]}
\end{aligned}$$

The effect class ($Class$) is a join semi-lattice in which $\bot \sqsubset \mathsf{pure} \sqsubset \mathsf{obs} \sqsubset \mathsf{proc}$, so that for example $\mathsf{pure} \sqcup \mathsf{obs} = \mathsf{obs}$ and $\mathsf{proc} \sqcup \mathsf{pure} = \mathsf{proc}$. The effect class of a function is proc if an execution of that function generates observable side effects, obs if a function execution depends on external side effects but none generate observable side effects, and pure otherwise.

Our purity analysis navigates the flow graph G_e and updates effect maps P, R and O. Mapping P maps functions onto their effect class. During a function execution, if an observable write effect is generated, then the applied function becomes a procedure (proc) in P.

For dependence on external effects, the analysis looks for read–write–read sequences involving the same resource and with the initial and subsequent read occurring in an execution of the same function. If a function reads an external resource, this effect is tracked in the read table (R). If a resource is modified, then all functions that have a read dependency on that resource are added to the observer table (O) as potential observers of modifications of the resource. If a function reads a resource and is mapped in the observer table as a potential observer for that resource, then that function becomes an actual observer of an external effect, and its effect class at that point is joined with obs in P.

We consider variables and object properties to be resources (Res) on which read and write effects are possible.

The purity analysis that we present in this section is finite if the underlying flow graph is finite, because the purity maps monotonically increase in a finite domain.

3.1. Graph Traversal

We define a function $travGraph_P$ that navigates flow graph G_e and propagates information about effects per procedure. The result of graph traversal is a map from functions to effect classes.

$$travGraph_P : \mathcal{P}(State) \times \mathcal{P}(State) \times Purity \times Read \times Obs \to Purity$$

The first parameter of function $travGraph_P$ is the set of seen states, which is initially empty. The second parameter is the "work list", which actually is a *set* of states that still need to be visited. Graph traversal starts at the initial state ς_0 of the flow graph. The final three parameters are the effect maps of the analysis needed to implement the algorithm sketched at the start of this section. These maps are initially empty. The initial argument for purity analysis therefore is $(\varnothing, \{\varsigma_0\}, [], [], [])$.

The following three cases define $travGraph_P$.

1. If the work list of states W is empty, the analysis terminates and the map of effect class per function P is returned.

$$travGraph_P(S, \varnothing, P, R, O) = P$$

2. If a state in the work list W is already in the set of seen states S, it is removed from the work list and traversal continues.

$$travGraph_P(S \cup \{\varsigma\}, W \uplus \{\varsigma\}, P, R, O) = travGraph_P(S \cup \{\varsigma\}, W, P, R, O)$$

3. The interesting case is when the analysis encounters a state ς that it has not yet seen with the current effect maps. For every effect eff in the set of effects E associated with an outgoing transition from state ς in the flow graph, the analysis delegates to effect handler handle_P. The handler returns potentially updated effects maps, and if any of these maps increases the set of seen states is cleared to make sure that at the end of graph traversal the analysis has visited each state with the maximum attainable effect maps.

$$\begin{aligned}
\mathit{travGraph}_P(S, W \uplus \{\varsigma\}, P, R, O) &= \mathit{travGraph}_P(S', W', P', R', O') \\
\text{where } (P', R', O') &= \bigsqcup\{\mathit{handle}_P(\mathit{eff}, \varsigma, P, R, O) \\
&\qquad \mid \mathit{eff} \in E \wedge (\varsigma \xrightarrow{E} \varsigma') \in G_e\} \\
W' &= W \cup \{\varsigma' \mid (\varsigma \to \varsigma') \in G_e\} \\
S' &= \begin{cases} \varnothing & \text{if } P \neq P' \vee R \neq R' \vee O \neq O' \\ S \cup \{\varsigma\} & \text{else} \end{cases}
\end{aligned}$$

Function handle_P is an intermediate dispatcher that forwards to $\mathit{handleWrite}$ and $\mathit{handleRead}$ after building the resource.

$$\mathit{handle}_P : \mathit{Eff} \times \mathit{State} \times \mathit{Purity} \times \mathit{Read} \times \mathit{Obs} \to \mathit{Purity} \times \mathit{Read} \times \mathit{Obs}$$

The first two parameters of this function are an effect and a state in which the effect occurs. The remaining parameters are the effect maps that will potentially be updated. In case of an effect on an object, function handle_P combines the address of the object and the property name into a resource. Otherwise the effect is on a variable and the corresponding resource is the address of the variable. The four cases for handle_P are as follows.

1. Variable write effect:

$$\mathit{handle}_P(\overbrace{\mathbf{Wv}(a, \_)}^{\mathit{eff}}, \varsigma, P, R, O) = \mathit{handleWrite}(\mathit{eff}, a, \varsigma, P, R, O)$$

2. Property write effect:

$$\mathit{handle}_P(\overbrace{\mathbf{Wp}(a, v)}^{\mathit{eff}}, \varsigma, P, R, O) = \mathit{handleWrite}(\mathit{eff}, (a, v), \varsigma, P, R, O)$$

3. Variable read effect:

$$\mathit{handle}_P(\overbrace{\mathbf{Rv}(a, \_)}^{\mathit{eff}}, \varsigma, P, R, O) = \mathit{handleRead}(\mathit{eff}, a, \varsigma, P, R, O)$$

4. Property read effect:

$$\mathit{handle}_P(\overbrace{\mathbf{Rp}(a, v)}^{\mathit{eff}}, \varsigma, P, R, O) = \mathit{handleRead}(\mathit{eff}, (a, v), \varsigma, P, R, O)$$

Handling a read potentially updates the effect class of a function and the read table. The effect handler $\mathit{handleRead}$ delegates to helper function $\mathit{travStack}_R$ to inspect all active function executions.

$$\begin{aligned}
\mathit{handleRead} &: \mathit{Eff} \times \mathit{Res} \times \mathit{State} \times \mathit{Purity} \times \mathit{Read} \times \mathit{Obs} \\
&\qquad \to \mathit{Purity} \times \mathit{Read} \times \mathit{Obs} \\
\mathit{handleRead}(\mathit{eff}, \mathit{res}, \varsigma, P, R, O) &= (P', R', O) \\
\text{where } (P', R') &= \mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, \varnothing, \{\kappa\}, P, R, O) \\
(\ldots, \kappa, \_) &= \varsigma
\end{aligned}$$

Handling a write potentially updates the effect class of active functions and the observer table for the affected resource. The effect handler $\mathit{handleWrite}$ delegates to helper function $\mathit{travStack}_W$ to inspect all active function executions. Updating observers does not depend on the observability of the write effect. If a resource is written, then every procedure that has a read dependency on that resource is unconditionally moved to the observer table.

$$\begin{aligned}
\mathit{handleWrite} &: \mathit{Eff} \times \mathit{Res} \times \mathit{State} \times \mathit{Purity} \times \mathit{Read} \times \mathit{Obs} \\
&\quad \to \mathit{Purity} \times \mathit{Read} \times \mathit{Obs} \\
\mathit{handleWrite}(\mathit{eff}, \mathit{res}, \varsigma, P, R, O) &= (P', R, O') \\
\text{where } O' &= O \sqcup [\mathit{res} \mapsto R(\mathit{res})] \\
P' &= \mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, \varnothing, \{\kappa\}, P) \\
(\ldots, \kappa, \_) &= \varsigma
\end{aligned}$$

3.2. Stack Traversal

Our purity analysis determines the observability of an effect for every function that is computing a return value at the point at which the effect is produced. It needs to find all the dynamic extents in which the effect occurs and does so by traversing the execution context stack. Because continuations in our semantics are delimited by function applications, we ignore local continuations entirely, and instead focus on meta-continuations and stack addresses in the stack store.

The stack-traversing operation $\mathit{travStack}_R$ propagates an effect down the stack while remembering the execution contexts that it has already seen to avoid infinite recursion.

$$\mathit{travStack}_R : \mathit{Eff} \times \mathit{Res} \times \mathit{State} \times \mathcal{P}(\mathit{Kont}) \times \mathcal{P}(\mathit{Kont}) \times \mathit{Purity} \times \mathit{Read} \times \mathit{Obs} \to \mathit{Purity} \times \mathit{Read}$$

The first parameter of $\mathit{travStack}_R$ is an effect that is propagated down the stack. The second parameter is the resource targeted by this effect: a value store address for effects on variables, or a value store address coupled to a property name for effects on object properties. The third parameter is the program state in which the effect occurs and from which the call stack is obtained. The fourth parameter is the set of seen execution contexts, and is initially empty. The set of seen contexts is used to prevent infinite cycles when traversing the stack. The fifth parameter is the work list of execution contexts that still need to be visited, and initially contains the running execution context of the program state in which stack traversal is performed. The final three parameters are the effect maps, of which potentially the purity and read map are updated during stack traversal. Function $\mathit{travStack}_R$ has to consider five distinct cases.

1. Stack traversal terminates when the work list is empty. The purity and read maps are returned.

$$\mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S, \varnothing, P, R, O) = (P, R)$$

2. An execution context that was already visited is not visited again.

$$\mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\kappa\}, W \uplus \{\kappa\}, P, R, O) = \mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\kappa\}, W, P, R, O)$$

3. The root context ϵ signals that the bottom of the stack has been reached. We do not treat the root (program) context as an execution context, so the effect maps are not updated.

$$\mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S, W \uplus \{\epsilon\}, P, R, O) = \mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\epsilon\}, W, P, R, O)$$

4. If a read effect occurs on an address that is mapped in the caller store σ, then the effect is on a resource that is reachable by the caller. In this case the function being applied (f) is marked as read-dependent on the resource by adding it to R. If this function is registered as a potential observer for the read resource, then the function is marked as an observer in P. Both the caller store σ and the applied function f are components of the current execution context τ, which is added to the set of seen contexts. Context τ is also dereferenced in the stack store to add all

underlying execution contexts to the work list.

$$\begin{aligned} \mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S, W \uplus \{\tau\}, P, R, O) &= \mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S', W', P', R', O) \\ &\quad \text{if } a \in \mathrm{Dom}(\sigma) \\ \text{where } S' &= S \cup \{\tau\} \\ W' &= W \cup \{\kappa' \mid (\_, \kappa') \in \Xi(\tau)\} \\ P' &= P \sqcup \bigsqcup \{f \mapsto \mathsf{obs} \mid f \in O(\mathit{res})\} \\ R' &= R \sqcup [\mathit{res} \mapsto \{f\}] \\ (\ldots, \Xi) &= \varsigma \\ (\_, (f, \_), \_, \_, \sigma) &= \tau \end{aligned}$$

5. If the read address is not in the domain of the caller store, then the effect is local to the function execution and can be masked. Because the effect also cannot be observed in underlying execution contexts, the analysis does not need to examine underlying stacks to add contexts to the work list. The current execution context τ is added to the set of seen contexts.

$$\mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S, W \uplus \{\tau\}, P, R, O) = \mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\tau\}, W, P, R, O)$$

Function $\mathit{travStack}_W$ similarly traverses the stack, checking whether the address of the written resource is mapped in the caller store of an active function execution.

$$\mathit{travStack}_W : \mathit{Eff} \times \mathit{Res} \times \mathit{State} \times \mathcal{P}(\mathit{Kont}) \times \mathcal{P}(\mathit{Kont}) \times \mathit{Purity} \times \mathit{Read} \times \mathit{Obs} \rightarrow \mathit{Purity}$$

The signature of $\mathit{travStack}_W$ is similar to the signature of $\mathit{travStack}_R$, but it only potentially updates the purity map. The function also has to consider five distinct cases.

1. Stack traversal terminates when the work list is empty. The purity map is returned.

$$\mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S, \varnothing, P) = P$$

2. An execution context that was already visited is not visited again.

$$\mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\kappa\}, W \uplus \{\kappa\}, P) = \mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\kappa\}, W, P)$$

3. The root context ϵ signals that the bottom of the stack has been reached, so the effect maps are not updated.

$$\mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S, W \uplus \{\epsilon\}, P, R, O) = \mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\epsilon\}, W, P, R, O)$$

4. If the address of the written resource is mapped in the caller store of an active function execution, then the effect is observable from the point of view of the caller, and the function is marked as a procedure in P. The current execution context τ is added to the set of seen contexts, and τ is dereferenced in the stack store to add all underlying execution contexts to the work list.

$$\begin{aligned} \mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S, W \uplus \{\tau\}, P) &= \mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S', W', P') \\ &\quad \text{if } a \in \mathrm{Dom}(\sigma) \\ \text{where } S' &= S \cup \{\tau\} \\ W' &= W \cup \{\kappa' \mid (\_, \kappa') \in \Xi(\tau)\} \\ P' &= P \sqcup [f \mapsto \mathsf{proc}] \\ (\ldots, \Xi) &= \varsigma \\ (\_, (f, \_), \_, \_, \sigma) &= \tau \end{aligned}$$

5. If the written address is not in the domain of the caller store, then the effect is local to the function execution and can be masked. The effect cannot be observed in underlying execution contexts and the current execution context τ is added to the set of seen contexts.

$$\mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S, W \uplus \{\tau\}, P) = \mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\tau\}, W, P)$$

```
1  function F(f) {
2    var a = this;
3    a.f = f;
4  }
5
6  F.create =
7    function (n) {
8      var f;
9      if (n < 1) {
10       f = null;
11     } else {
12       f = F.create(n-1);
13     }
14     return new F(f);
15   }
16
17 F.create(3);
```

Figure 4. Example program with recursive pattern.

4. FRESHNESS ANALYSIS

Address-based purity analysis from Section 3 is a close fit with the store semantics of the abstract machine presented in Section 2.2. Resources are represented as addresses, coupled to property names for property effects, and side effects are determined based on the addresses read from and written to. While effect masking based on addresses is attractive in a store semantics, it is also problematic in a typical static analysis setting, which motivates the addition of lexical freshness analysis.

4.1. Motivation

In concrete semantics, resources that should be distinct are treated as such, because they are referenced through distinct addresses. When our abstract machine is configured with concrete value and stack store allocators, addresses are generated with full precision, and the machine allocates like a regular interpreter for JavaScript would at runtime. As a result, concrete side-effect analysis determines procedure side effects with full precision, i.e., with no false positives or negatives. It follows that effect masking based on addresses then also is fully precise. However, in a static analysis setting this is not realistic.

4.1.1. Problem: Limited Precision for Addresses To guarantee that an analysis runs in reasonable (finite) time and space, our compile-time analysis sacrifices precision primarily by *limiting the number of addresses* the abstract machines allocator may choose from while analyzing the input program. This strategy is at the heart of static analysis techniques for higher-order languages like AAM [36] and AAC [20], on which we based our semantics. For example, 0CFA is a widely used address allocation policy for abstract semantics in which every syntactic variable is its own address (Section 2.2.7).

When effect masking hinges on addresses, we must assess the impact of this precision loss on our approach. As it turns out, the precision loss can be problematic, as the following examples illustrate.

Example 1. The program depicted in Figure 4 represents the essence of a pattern for constructing a composite data structure. Suppose that every object created on line 14 is allocated at a single address a. When constructor `F` on line 14 is called after the recursive call `F.create` on line 12 in the `else` branch, that recursive call has already allocated an object at address a. Therefore, our purity analysis concludes that property load `a.f` on line 3 in the constructor writes to an address that already exists in the caller store. As a result, constructor `F` is imprecisely considered to be impure, although a constructor should be allowed to mutate the object referenced by its `this` parameter without generating an observable side effect.

In the same example program in Figure 4, we identify a second problem. Suppose that variable `f` on line 8 is always allocated at the same address. Then, in a recursive call to `F.create` both assignments to `f` (lines 10 and 12) are considered to be a write to an address that exists in the caller store, imprecisely regarding constructor `F` as impure. □

The above example—involving write effects, but read effects suffer the same fate—shows that effect masking based solely on addresses suffers from inherent imprecision introduced by selecting addresses from a finite set. We say "inherent", because while it is always possible to recover some loss of precision by for example generating context-sensitive addresses (e.g. using 1CFA allocation [34]), at one point or another the machine will run out of fresh addresses for a particular resource. Increasing context-sensitivity may cause more function executions to be considered free from observable side effects by our analysis, but it takes only a single function execution with an observable side effect for the applied function to be marked as exhibiting side effects, and increasing context-sensitivity only delays the inevitable.

4.1.2. Solution: Lexical Freshness The precision of address-based purity analysis can be improved by taking into account certain invariants that are independent of addresses and therefore of the actual choice of address allocation policy. For example, mutating the newly created object in a constructor call is never an observable side effect. Similarly, writing to a local variable does not generate an observable side effect either. These and other invariants have one thing in common that we want to check for: *freshness*.

We develop a lexical freshness analysis for variables (for variable effects) and objects (for property effects). Unlike address-based purity analysis, we only consider variable and object freshness in the running execution context, i.e., the execution context on top of the stack resulting from the most recent function application. The reason is that address-based purity analysis is based on stack reachability and the stack embodies dynamic scope, while variables are lexically scoped. Therefore, applying freshness analysis in underlying execution contexts without performing additional analysis such as escape analysis would be unsound.

4.2. Variable Freshness

Under concrete semantics, whenever evaluation enters the body of a function, parameters and local variables are always freshly allocated. Therefore a variable is fresh with respect to the running execution context if it is a local variable in that context, no matter where the variable is allocated in the store. The handler for a variable effect can then be defined without the need for checking the address of that variable.

4.2.1. Extending Purity Analysis With Variable Freshness Variable freshness analysis plugs into the address-based purity analysis from Section 3 by adding two additional handlers for effects on variables that shortcut the previous cases of $handle_P$. These handlers only relies on lexical scoping information offered by predicate $isLocal \subseteq \mathsf{Var} \times \mathsf{Fun}$, which returns whether a variable is declared in an enclosing scope of a given function scope or not.

1. Writing to a local variable is not observable to callers.

$$\begin{gathered} handle_P(\mathbf{Wv}(\_, v), (\ldots, \kappa, \_), P, R, O) = (P, R, O) \\ \text{if } isLocal(v, f) \\ \text{where } (\_, (f, \_), \_, \_, \_) = \kappa \end{gathered}$$

2. Reading a local variable is not observable to callers.

$$\begin{gathered} handle_P(\mathbf{Rv}(\_, v), (\ldots, \kappa, \_), P, R, O) = (P, R, O) \\ \text{if } isLocal(v, f) \\ \text{where } (\_, (f, \_), \_, \_, \_) = \kappa \end{gathered}$$

4.3. Object Freshness

The goal of our object freshness analysis is to determine whether expressions only reference fresh objects or not, without involving object addresses. An object is fresh with respect to the running execution context if it was created in that context. Objects differ from variables in that object references are first-class values and object references are not statically anchored by scopes like variable references are. Instead, references to objects are allowed to flow freely throughout a program. Whenever an object is allocated, a reference to that object is returned in the form of an address. This reference is a first-class value that can be stored, either by binding to variables or parameters, or assignment to variables and object fields. Consequently, object freshness analysis is more complex in our approach than variable freshness.

We express object freshness by a join semi-lattice *Freshness* in which $\bot \sqsubset \mathbf{fr} \sqsubset \mathbf{unfr}$, so that for example $\mathbf{fr} \sqcup \mathbf{unfr} = \mathbf{unfr}$. An expression is fresh ($\mathbf{fr}$) if it only references fresh objects, else it is unfresh ($\mathbf{unfr}$).

$$\psi \in \mathit{Freshness} = \{\mathbf{fr}, \mathbf{unfr}\}$$

Determining object freshness in all program states requires two operations: a function that determines the freshness of expressions, and an analysis that propagates object freshness of variables in the running execution context.

We start by defining a function *fresh* that returns whether an expression only references fresh objects or not.

$$\begin{aligned} \mathit{fresh} &: \mathsf{Exp} \times \mathit{Kont} \times \mathit{Fresh} \to \mathit{Freshness} \\ F \in \mathit{Fresh} &= \mathsf{Var} \rightharpoonup \mathit{Freshness} \end{aligned}$$

The first parameter of function *fresh* is the expression (e) of which the freshness has to be determined. The second parameter is the running execution context (κ) in which this expression appears. The third parameter (*Freshness*) is a map from variables to object freshness that is in effect when determining the object freshness of the expression. The following cases define function *fresh*.

1. The object freshness of a local variable is looked up in map F.

$$\mathit{fresh}([\![v]\!], \kappa, F) = F(v) \quad \text{if } v \in \mathrm{Dom}(F)$$

2. An object constructed through `new` is fresh.

$$\mathit{fresh}([\![\texttt{new}\ v\,\texttt{(}s\texttt{)}\,]\!], \kappa, F) = \mathbf{fr}$$

3. Freshness of an assignment expression is determined by the freshness of its value, i.e., its right hand side.

$$\mathit{fresh}([\![v\texttt{=}e]\!], \kappa, F) = \mathit{fresh}(e, \kappa, F)$$

4. A reference to a newly constructed object through `this` in a constructor is fresh.

$$\mathit{fresh}([\![\texttt{this}]\!], ([\![\texttt{new}\ v\,\texttt{(}s\texttt{)}\,]\!], \ldots), F) = \mathbf{fr}$$

5. All other expressions are not fresh.

$$\mathit{fresh}(e, \kappa, F) = \mathbf{unfr}$$

When provided with a map $F \in \mathit{Fresh}$ that indicates whether a variable only points to fresh objects or not, function *fresh* can determine object freshness in all program states. We formalize an object freshness analysis to compute F that operates on the same flow graph as the purity analysis it aims to improve. The analysis propagates F until it reaches a fixpoint, and a current value of F is used through function *fresh* to compute updates to F during this fixpoint computation.

In our approach, object freshness only propagates through variable assignment within the running execution context. Resources that exist in an outer function scope are conservatively considered unfresh. The object freshness analysis does not track freshness through, for example, function calls or property loading and storing: for these kinds of object flow, purity analysis relies entirely on the underlying abstract interpretation and the addresses it allocates.

4.3.1. Graph Traversal Like the purity analysis from Section 3, object freshness analysis piggybacks on the underlying flow graph for control flow by using traversal function $\mathit{travGraph}_F$ that delegates to handle_F to update map F for newly encountered states.

$$\mathit{travGraph}_F : \mathcal{P}(\mathit{State}) \times \mathcal{P}(\mathit{State}) \times \mathit{Fresh} \to \mathit{Fresh}$$

The first parameter of $\mathit{travGraph}_F$ is the set of seen states. The second parameter is the work list of states, i.e., states that still need to be visited. The third parameter is a map from variables to object freshness, and represents the result this analysis computes. The initial argument for the freshness analysis is $(\varnothing, \{\varsigma_0\}, [])$, where ς_0 is the initial state of the flow graph. To use the results of object freshness analysis, function $\mathit{travGraph}_F$ has to be called before $\mathit{travGraph}_P$, which performs the actual purity analysis.

We define $\mathit{travGraph}_F$ using three cases.

1. If the work list of states W is empty then graph traversal has finished and the map Fresh from variables to object freshness is returned.

$$\mathit{travGraph}_F(S, \varnothing, F) = F$$

2. If the state pulled from the work list is already in the set of seen states S, it is removed from the work list and traversal continues.

$$\mathit{travGraph}_F(S \cup \{\varsigma\}, W \uplus \{\varsigma\}, F) = F$$

3. For a state that was not yet encountered, function $\mathit{travGraph}_F$ delegates to handle_F. If the latter function updates the freshness of variables, then the set of seen states is cleared before traversal continues to ensure that at the end of graph traversal the analysis has visited each state with the maximum attainable object freshness for each variable.

$$\begin{aligned} \mathit{travGraph}_F(S, W \uplus \{\varsigma\}, F) &= \mathit{travGraph}_F(S', W', F') \\ \text{where } W' &= W \cup \{\varsigma' \mid (\varsigma \to \varsigma') \in G_e\} \\ F' &= \mathit{handle}_F(\varsigma, F) \\ S' &= \begin{cases} \varnothing \text{ if } F \neq F' \\ S \cup \{\varsigma\} \text{ else} \end{cases} \end{aligned}$$

Object freshness information for variables is propagated through states, and is potentially updated when an evaluation state for an assignment expression is encountered. Continuation states do not modify the freshness map, because the analysis tracks freshness through expressions rather than values. State handler handle_F implements this logic.

$$\mathit{handle}_F : \mathit{State} \times \mathit{Fresh} \to \mathit{Fresh}$$

The first parameter of this function is a program state, and the second parameter is a map from variables to object freshness.

1. For a variable assignment expression, predicate $\mathit{isLocal} \subseteq \mathsf{Var} \times \mathsf{Fun}$ is used to determine whether the variable being assigned is local to the running function execution. If this is the case, then freshness is propagated from the right hand side to the left hand side. Else, the

variable becomes unfresh.

$$handle_F(\mathbf{ev}([\![v{=}e]\!], \rho, \sigma, \iota, \kappa, \Xi), F) = F'$$
$$\text{where } F' = \begin{cases} F \sqcup [v \mapsto \mathit{fresh}(e, \kappa, F)] \text{ if } \mathit{isLocal}(v, f) \\ F \sqcup [v \mapsto \mathbf{unfr}] \text{ else} \end{cases}$$
$$\kappa = (\_, (f, \_), \ldots)$$

2. For any other state the freshness map remains the same.

$$handle_F(\varsigma, F) = F$$

4.3.2. Extending Purity Analysis With Object Freshness Object freshness analysis plugs into the address-based purity analysis from Section 3 by adding three additional handlers for property effects that must be checked before the existing property effect handlers defined in that section. These handlers operate in the running execution context and avoid using store addresses for determining purity when a fresh object is involved, improving the precision of the resulting purity analysis. Let $F \in \mathit{Fresh}$ be the result of performing object freshness analysis. We define the three additional handlers as follows.

1. A property write effect happens through an explicit property store or when assigning to a top-level variable, although in the latter case the global object is never fresh during function execution. Storing a property on a fresh object in the running execution context is not observable to the most recent caller.

$$handle_P(\mathbf{Wp}(\_, \_), \overbrace{\mathbf{ko}(d, \sigma, \mathbf{st}(s, v, \rho) : \iota, \kappa, \Xi)}^{\varsigma}, P, R, O) = (P, R, O)$$
$$\text{if } \mathit{fresh}(s, \kappa, F) = \mathbf{fr}$$

2. A property read effect is the consequence of an explicit property access through *lookupProp* or when reading from a top-level variable (the latter case again involving an unfresh global object). Reading a property on a fresh object as a consequence of a property load in the running execution context is not observable to the most recent caller.

$$handle_P(\mathbf{Rp}(\_, \_), \mathbf{ev}([\![s \,.\, v]\!], \rho, \sigma, \iota, \kappa, \Xi), P, R, O) = (P, R, O)$$
$$\text{if } \mathit{fresh}(s, \kappa, F) = \mathbf{fr}$$

3. Similar to the previous case, reading a property on a fresh object as a consequence of a method call in the running execution context is not observable to the most recent caller.

$$handle_P(\mathbf{Rp}(\_, \_), \mathbf{ev}([\![s_0 \,.\, v\ (s_1)\,]\!], \rho, \sigma, \iota, \kappa, \Xi), P, R, O) = (P, R, O)$$
$$\text{if } \mathit{fresh}(s_0, \kappa, F) = \mathbf{fr}$$

5. RELATED WORK

5.1. Side-effect and Purity Analysis

There exists a large body of work on purity and closely related concepts such as side-effect analysis, referential transparency, and memoization. We give a small overview of related work.

Salcianu and Rinard [33] present a purity analysis that is based on an underlying pointer analysis (JPPA). Their analysis first constructs parameterized points-to graphs for every method, in which nodes are objects and edges are heap references. A later interprocedural step instantiates these points-to graphs at every call. The goal of the analysis is to distinguish objects allocated during invocation of a method from objects that already exist in the caller state. From a high-level perspective, our approach is comparable: we also perform an intraprocedural analysis to compute freshness, which we then complete with interprocedural information to determine purity. However, our intraprocedural step consists of an instrumentation of the language's semantics and is closer to how an interpreter works, whereas the approach of Sacianu and Rinard computes a points-to graph for every program point, which requires complex manipulation of graphs. Because JPPA targets Java programs, it does not support higher-order functions and closures, while our approach does. Extending JPPA to support more language features would require adapting the complex graph construction parts. With our approach, the changes would follow from the semantics of the language features. Like our approach, JPPA requires a whole-program analysis.

Pearce introduces JPure [29], a modular purity analysis based on annotations that express freshness and locality, two concepts that map closely to how we use these terms in this work. When the focus is on maintaining purity, no whole-program interprocedural analysis is required. JPure is rooted in Java, and can automatically infer annotations. Similar to our technique, the tool starts from an intraprocedural dataflow analysis to model freshness and locality of object references. Purity inference then interprocedurally propagates information using static class hierarchy.

Madhavan et al. [23] present the pointer analysis used in Salcianu and Rinard [33] as an abstract interpretation. As the work of Madhavan et al. is a different formulation of the same technique used in JPPA, it is subject to the same limitations. Namely, extending it to support higher-order functions and closures would require a reformulation of the semantics, while this is incorporated at the core of our approach. Our work is also based on abstract interpretation, but is different from original JPPA formulation and thus also from its abstract interpretation reformulation.

Huang et al. [17] present ReImInfer, a type inference analysis for reference immutability in Java. ReIm, the underlying type system, qualifies references as being readonly, mutable, or polyread, the latter signifying that a reference is immutable in the current context but may not be in other contexts. Methods are pure if none of their parameters, including `this`, are inferred to be mutable. Like the purity analysis we present in this paper, ReImInfer is context-sensitive when applying viewpoint adaptation. Our execution contexts are more precise than any approach discussed in [17], but at the expense of running time and scalability. ReImInfer does not handle higher-order language constructs.

Rytz et al. [32] present a modular type-and-effect system for purity in Scala. It is strongly influenced by JPure and it also relies on annotations, while our analysis does not. Their effect system is flow-insensitive to make it suitable for higher-order languages, while our work is both flow-sensitive and capable of handling higher-order constructs.

Pitidis and Sagonas [30] treat purity in the setting of the functional language Erlang. They take the stricter definition of purity where a function also may not depend on external side effects. Similar to our approach, higher-order functions are supported. However, the authors mention that the dataflow analysis they rely on is "pretty simple" and "not so accurate", without giving more information. In this work, we employ a state-of-the-art flow analysis offering maximal call/return precision.

Finifter et al. [13] discusses purity from the point of view of a deterministic object-capability language (Joe-E) based on Java. Purity of methods can be enforced by statically declaring all parameters as having an immutable type. Declaring parameters as immutable is sufficient in a language that does not have closures with mutable free variables, but would not apply in our setting of JavaScript, which does have mutable free variables.

We adopted the term *effect class* from Gifford and Lucassen [14]. In their work on combining functional and imperative programming languages, they look at all possible combinations of reading, writing, and allocation of memory locations by expressions with the goal of classifying these expressions accordingly.

Functions can call other functions, and when an effect occurs an effect analysis must traverse the stack to find all "active" functions, i.e., functions that are in the process of computing a result. This is what Might and Prabhu [25] formulated as Harrisons dependence principle [16], which inspired us to use stack traversal to determine the observable extent of side effects. Reformulated in the context of side-effect and purity analysis, Harrison's principle states that a side effect occurs for all functions on the call stack. Might and Prabhu present a finite-state dependence analysis and does not treat compound values and objects. Our work employs a more precise pushdown analysis and supports objects.

5.2. *Static Analysis of JavaScript*

We use JIPDA[†] as the underlying abstract interpreter for producing flow graphs. There exist several other JavaScript analysis tools such as WALA[‡], TAJS [19], and JSAI [21]. We conjecture that it is possible to implement our approach in these tools. JIPDA was constructed as a prototype for analyzing a small subset of JavaScript following the AAC method [20] based on abstract machines, and therefore exposes the program representation and information required for performing our purity analysis by design. Enabling the implementation of our approach on top of existing tools therefore necessitates the effort of extending or adapting these tools so that the required program information becomes available.

Any implementation of our side-effect and purity analysis requires a flow analysis that models store-allocated resources, effects on these resources, and the call stack. It must also be able to associate the applied function and corresponding caller store with each function execution. Additionally, our object freshness and purity analysis require a flow-sensitive program analysis. We formulated our analyses on top of a flow graph that exposes the necessary information—more specifically nodes in this graph represent program states and edges are annotated with effects that occur on transition between states.

6. IMPLEMENTATION

We implemented the purity and freshness analyses discussed in previous sections as a proof of concept[§]. The prototype implementation is structured according to the approach we outline in this work. At the base level, we use JIPDA as the underlying abstract interpreter for producing flow graphs. We define our freshness analysis to be executed on top of the produced flow graphs. Our purity analysis consumes a flow graph and (optionally) the results of freshness analysis that was run over the same graph.

Our implementation extends and optimizes the syntax and semantics of JS_0 as presented in Section 2. We now discuss the most important extensions (Section 6.1) and optimizations (Section 6.2), some of which are also used in the detailed analysis examples presented in Section 7.

6.1. *Extensions*

We designed JS_0 to be sufficiently representative to expose the important points in the design of a purity analysis for JavaScript, omitting many constructs found in JavaScript and many other mainstream languages to simplify the presentation. In this section we augment JS_0 with some of these omitted constructs that we also incorporated in our implementation, and which should ease the task of implementors of our approach in a realistic setting.

6.1.1. Primitive values The only type of value directly expressible in JS_0 are objects, represented by addresses in the semantics. The value `undefined` can be obtained indirectly by looking up a non-existing property. Extending JS_0 with primitive values starts by expanding the set of simple expressions Simple with sets that represent these values. We illustrate this by showing the extensions

[†] https://github.com/jensnicolay/jipda
[‡] http://wala.sf.net
[§] https://github.com/jensnicolay/jipda/tree/jsep/protopurity

for the `undefined` value, `null`, numerals, booleans, and strings.

$$
\begin{aligned}
s \in \mathsf{Simple} ::=\;& \ldots \\
\mid\;& b && \text{[primitive value]} \\
b \in \mathsf{Prim} ::=\;& \texttt{undefined} && \text{[undefined value]} \\
\mid\;& \texttt{null} && \text{[null value]} \\
\mid\;& \mathit{num} && \text{[number value]} \\
\mid\;& \texttt{true} && \text{[true value]} \\
\mid\;& \texttt{false} && \text{[false value]} \\
\mid\;& \mathit{str} && \text{[string value]} \\
\mathit{num} \in \mathsf{Num} =\;& \text{a set of numerals} \\
\mathit{str} \in \mathsf{Str} =\;& \text{a set of strings}
\end{aligned}
$$

The state-space of JS_0, given in Figure 3, already contains the set Prim as an extension point for plugging in additional primitive values. In Section 2 it only contains value **undef**, because it is the only primitive value required for expressing the semantics of JS_0 as described in the remainder of that section. The set Prim must now be enlarged to also contain the other types of primitive values, and additionally we introduce the abstraction function $\alpha : \mathsf{Prim} \to \mathit{Prim}$ that maps primitive values to their abstracted values and on which Prim depends. For expressing concrete semantics, α can be the identity function. For (finite) abstract semantics, primitive values can be mapped to their type, resulting in the following definitions for Prim and α.

$$
\begin{aligned}
\mathit{Prim} &= \{\mathbf{undef}, \mathbf{null}, \mathbf{num}, \mathbf{bool}, \mathbf{str}\} \\
\alpha(\texttt{undefined}) &= \mathbf{undef} \\
\alpha(\texttt{null}) &= \mathbf{null} \\
\alpha(\mathit{num}) &= \mathbf{num} \\
\alpha(\texttt{true}) &= \mathbf{bool} \\
\alpha(\texttt{false}) &= \mathbf{bool} \\
\alpha(\mathit{str}) &= \mathbf{str}
\end{aligned}
$$

Finally, a new case is added to the evaluator for simple expressions from Section 2.2.8.

$$\mathit{evalSimple}(b, \rho, \sigma, \kappa) = (\{\alpha(b)\}, \varnothing)$$

6.1.2. Primitive operators JavaScript has unary and binary operators such as `typeof` and `+`. For brevity, we illustrate the addition of binary operators only. First, the syntax of JS_0 is extended with binary expressions and operators.

$$
\begin{aligned}
e \in \mathsf{Exp} =\;& \ldots \\
\mid\;& s_1 \diamond s_2 && \text{[binary expression]} \\
\diamond \in \mathit{BinOps} =\;& \{+, -, \star, \ldots\} && \text{[binary operator]}
\end{aligned}
$$

Next, the abstract machine semantics for JS_0 are extended with a transition rules for handling binary operators.

$$
\begin{aligned}
\mathbf{ev}([\![s_1 \diamond s_2]\!], \rho, \sigma, \iota, \kappa, \Xi) &\mapsto (\mathbf{ko}(d, \rho, \sigma, \iota, \kappa, \Xi), E) \\
\text{where } (d_r, E_0) &= \mathit{evalSimple}(s, \rho, \sigma, \kappa) \\
(d_r, E_1) &= \mathit{evalSimple}(s, \rho, \sigma, \kappa) \\
d &= \alpha(\diamond)(d, d'') \\
E &= E_0 \cup E_1
\end{aligned}
$$

To work with abstract values, operators themselves also must be abstracted and we illustrate this for addition. If we take $\oplus = \alpha(\llbracket + \rrbracket)$, then we can define $\oplus$ as follows.

$$\begin{aligned} \oplus(\{num\}, \{num\}) &= \{num\} \\ \oplus(\{num\}, \{string\}) &= \{string\} \\ \oplus(\{num, string\}, \{num\}) &= \{num, string\} \\ \oplus(\{num\}, \{bool\}) &= \{num\} \\ &\dots \end{aligned}$$

6.1.3. Conditional expressions Adding a conditional expression such as `if–then–else` to JS_0 poses no additional diffulties. First, the syntax of JS_0 is extended.

$$\begin{aligned} e \in \mathsf{Exp} ::= &\dots \\ &| \ \texttt{if (}s\texttt{)}\ e_0 \ \texttt{else}\ e_1 \end{aligned}$$

We define an **if** frame to continue with the `then` or `else` branch (or both), depending on the result of evaluation the condition.

$$\begin{aligned} \phi \in \mathit{Frame} ::= &\dots \\ &| \ \ \texttt{if}(e_0, e_1, \rho) \end{aligned}$$

We can now define two additional transition rules for the abstract machine.

1. When evaluating an `if` expression, the machine steps into the condition expression, pushing an **if** frame on the stack.

$$\begin{aligned} \mathbf{ev}(\llbracket \texttt{if (}s\texttt{)}\ e_0\ \texttt{else}\ e_1 \rrbracket, \rho, \sigma, \iota, \kappa, \Xi) \mapsto &(\mathbf{ev}(s, \rho, \sigma, \phi : \iota, \kappa, \Xi), \varnothing) \\ &\text{where } \phi = \mathbf{if}(e_0, e_1, \rho) \end{aligned}$$

2. When a condition evaluated to a truthy value, then the `else` branch is taken. Helper predicate $\mathit{isTruthy} \subseteq D$ returns whether this is the case.

$$\begin{aligned} \mathbf{ko}(d, \sigma, \mathbf{if}(e_0, \_, \rho)) : \iota, \kappa, \Xi) \mapsto &(\mathbf{ev}(e_0, \rho, \sigma, \iota, \kappa, \Xi), \varnothing) \\ &\text{if } \mathit{isTruthy}(d) \end{aligned}$$

3. When a condition evaluated to a falsy value, then the `else` branch is taken. Helper predicate $\mathit{isFalsy} \subseteq D$ returns whether this is the case.

$$\begin{aligned} \mathbf{ko}(d, \sigma, \mathbf{if}(\_, e_1, \rho)) : \iota, \kappa, \Xi) \mapsto &(\mathbf{ev}(e_1, \rho, \sigma, \iota, \kappa, \Xi), \varnothing) \\ &\text{if } \mathit{isFalsy}(d) \end{aligned}$$

Examples of falsy values in JavaScript are `undefined`, `null`, `false`, the number zero, and the emtpy string, while truthy values include `true`, non-zero numbers, and non-empty strings. It is possible that an abstract value is both truthy and falsy, so that both branches of an `if` are taken. For example, using the type abstraction defined earlier, both $\mathit{isTruthy}(\{\mathbf{bool}\})$ and $\mathit{isFalsy}(\{\mathbf{bool}\})$ are true.

6.1.4. Sequence expression Adding support for sequences follows the same pattern as for other compound expressions, starting with extending the syntax.

$$\begin{aligned} e \in \mathsf{Exp} = &\dots \\ &| \ \ e_1 \texttt{;} e_2 \end{aligned}$$

A **sq** frame is used by the machine to continue evaluation of the remainder of the sequence.

$$\begin{aligned} \phi \in \mathit{Frame} ::= &\dots \\ &| \ \ \texttt{sq}(e, \rho) \end{aligned}$$

Two additional transition rules, one for evaluating and one for continuing the evaluation of a sequence expression, are added to the abstract machine.

1. When evaluating a sequence expression, the machine starts by evaluating the first expression of the sequence, pushing a **sq** frame to evaluate the remainder of the sequence.

$$\begin{gathered}\mathbf{ev}([\![e_1\,\texttt{;}\,e_2]\!],\rho,\sigma,\iota,\kappa,\Xi) \mapsto (\mathbf{ev}(e_0,\rho,\sigma,\phi:\iota,\kappa,\Xi),\varnothing)\\ \text{where } \phi = \mathbf{sq}(e_1,\rho)\end{gathered}$$

2. Continuing with a value of an expression in a sequence moves evaluation to the next expression in that sequence, discarding the value.

$$\mathbf{ko}(\_,\sigma,\mathbf{sq}(e_1,\rho)) : \iota,\kappa,\Xi) \mapsto (\mathbf{ev}(e_1,\rho,\sigma,\iota,\kappa,\Xi),\varnothing)$$

6.1.5. Variable declarations The semantics in Section 2 assumed a single local variable hoisted to the top of its function scope. In JavaScript, however, local variables can be declared using the `var` keyword anywhere a statement is allowed. In JS$_0$ we encode a variable declaration as an expression.

$$\begin{aligned} e \in \mathsf{Exp} &= \ldots\\ &\mid\ \texttt{var}\ v\end{aligned}$$

A variable declaration using `var` is hoisted to the top of its immediately enclosing function scope, no matter where the declaration appears, and its implicit initial value is always `undefined`. Evaluating a hoisted `var` declaration therefore consists of calling the *allocVar* allocator function from Section 2.2.7 to generate a store address, and allocating `undefined` at that address. When evaluation moves to the actual function body, the remaining `var` declaration statements are no-ops.

$$\begin{aligned}\mathbf{ev}([\![\texttt{var}\ v]\!],\rho,\sigma,\iota,\kappa,\Xi) &\mapsto (\mathbf{ko}(\{\mathbf{undef}\},\rho',\sigma',\iota,\kappa,\Xi),\varnothing)\\ \text{where } \rho' &= \rho[v\mapsto a]\\ \sigma' &= \sigma \sqcup [a \mapsto \{\mathbf{undef}\}]\\ a &= \mathit{allocVar}(v,\rho,\sigma,\iota,\kappa)\end{aligned}$$

In case an initializer expression is present, the implicit initial value of all hoisted variables is still `undefined` and declaration sites with initializer expressions are evaluated as assignment expressions.

In addition to variable declaration statements, JavaScript also features function declarations, which are also hoisted to the top of the function scope in which they are declared but, unlike variable declarations with initializers, evaluating a hoisted function declaration immediately evaluates the function and the remaining `function` declaration statements are always no-ops.

6.1.6. Update expressions JavaScript has four types of update expressions: pre- and postincrement, and pre- and postdecrement. From the perspective of language semantics, we can also distinguish between an update of a variable and update of an object property. We illustrate how JS$_0$ is extended with a post-increment expression for variables—support for the other update variants is similar.

First, the syntax of JS$_0$ is extended with variable post-increment expressions.

$$\begin{aligned} e \in \mathsf{Exp} &::= \ldots\\ &\mid\ v\texttt{++}\end{aligned}$$

Next, a transition rule for evaluating a variable post-increment expression is added to the abstract machine. Because of the semantics of an *a posteriori* update the store is updated with the incremented value d' but the *initial* value d is returned as a result. Unlike the transition rules so far, an update expression generates a read and write effect within a single step on the same resource.

$$\begin{aligned}\mathbf{ev}([\![v\text{++}]\!], \rho, \sigma, \iota, \kappa, \Xi) \mapsto (&\mathbf{ko}(d, \sigma', \iota, \kappa, \Xi), \{\mathbf{Rv}(a, v), \mathbf{Wv}(a, v)\})\\ \text{where } a &= \rho(v)\\ d &= \sigma(a)\\ d' &= \oplus(d, \alpha(1))\\ \sigma' &= \sigma \sqcup [a \mapsto d']\end{aligned}$$

6.1.7. Additional control-flow constructs The rules of the abstract machine for JS_0 define the control-flow, most commonly by stepping into and out of expression by pushing and popping a single stack frame, respectively. In this fashion, additional traditional iterative constructs such as `for` and `while-do` loops can be expressed.

Calling a function is more involved because of the bounding of local continuations with meta-continuations, but essentially is represented by pushing a meta-continuation on the stack that serves as a marker for function return using a `return` expression. Returning from functions is different from other **ko** transition rules because it may involve popping more than one frame at a time. Pushing other kinds of marker frames enables the implementation of other "frame-skipping" control-flow operators such as `continue`, `break`, and `try-catch`.

6.1.8. Direct style The syntax and semantics of JS_0 in Section 2 make use of simple expressions ($s \in \mathsf{Simple}$) to simplify the presentation. It is straightforward to extend the language to allow a more direct style of programs in which any kind of expression is allowed at program points that we restricted to simple expressions only. For example, allowing any kind of expression $e \in \mathsf{Exp}$ in binary expressions would involve one **ev** rule to evaluate the left expression, pushing an **le** frame, one **ko** rule that continues with **le** and evaluates the right expression pushing a **ri** frame, and finally another **ko** rule that continues with **ri**, performs the actual addition, and continues with the result. From this example it is clear how using simple expressions avoids the definition of specific frames and rules for handling these frames on the stack.

6.1.9. Other extensions Our implementation further extends the input language and semantics that we presented so far in this section. The proof-of-concept implementation is closer to full-fledged JavaScript and includes support for arrays and implicit type coercions. The implementation also supports computed properties. Instead of working with a set of names (strings), the abstract interpreter used in our implementation works with *abstract* names that come from the value lattice Prim used to represent all primitive values during interpretation. Emitted property events therefore also contain abstract names and looking up property names happens through subsumption instead of equality.

We also added many of the built-in JavaScript functions and objects that were required to run the benchmark programs for our experiments (Section 8).

6.2. Optimizations

Our implementation contains two important optimization techniques to improve performance and precision of the flow analysis and purity analysis.

6.2.1. Abstract garbage collection Our proof-of-concept implementation uses abstract garbage collection (AGC) as a technique to increase performance and precision of abstract interpretation [10]. Abstract garbage collection reclaims unused addresses and, in principle, should increase the precision of an address-based purity analysis. Disabling AGC on smaller and synthetic benchmarks only incurred a small negative impact on precision, which was dominated by the absence or presence of freshness analysis. Furthermore, AGC was required to run the larger benchmarks. Scaling up the abstract interpreter and client analyses, also to better assess the impact of AGC, is an ongoing effort.

```
1  z = 0;
2
3  function f()
4  {
5    var o;
6    o = {};      // unobservable variable write
7    o.x = 123;   // unobservable property write
8    z = 4;       // observable property write
9    f()
10 }
11
12 f()
```

Figure 5. Example program with infinite recursion used for performing abstract flow analysis, freshness analysis, and purity analysis.

6.2.2. Abstract counting The formalization of the flow analysis we present in Section 2 is the abstract semantics using weak updates to modify the store by using join ($\sqcup$). Our prototype implements abstract counting [26], a technique that determines when it is safe to perform strong store updates during abstract interpretation. Strong updating *replaces* a previous store value with a new value, which increases precision.

Abstract counting is useful for handling variable declarations (Section 6.1.5) with improved precision. Hoisted function-scoped variables are implicitly initialized to value `undefined`, and without strong updates this implies that these variables are considered as possibly being `undefined` during their lifetime, even when assigned another value before being referenced. Especially in combination with abstract garbage collection, abstract counting results in an increased possibility of strong updates of local variables, where their initial (previous) value is replaced by rather than joined with a new value.

7. EXAMPLE

We illustrate our full approach on the example program in Figure 5. This program contains a sole function `f` that recursively calls itself, resulting in an infinite loop. Function `f` is impure because it writes property `z` on the global object on line 8. However, when the observable property write on line 8 is removed, function `f` is pure because the variable write in line 6 is on a local variable and the property write on line 7 targets a fresh object.

We start the example by performing flow analysis, resulting in a flow graph annotated with effects (Section 7.1). Next, we perform purity analysis by only looking at addresses of resources (Section 7.2). Finally, we show how freshness analysis improves the precision of address-based purity (Section 7.3).

7.1. Flow Analysis

Figure 6 gives a schematic overview of the flow graph produced by the flow analysis presented in Section 2 and with some of the extensions described in Section 6.1. The example uses abstract semantics, allocating each resource at a single address (which in this example maps one-to-one on using allocation sites as addresses) and abstracting primitive values (primarily numerical values in the example) to their type.

Starting from the initial state ς_0, the flow graph progresses linearly until state ς_{39}, where the computation of the flow graph reaches a fixpoint by transitioning to state ς_{24}, which was explored earlier.

The flow graph can be divided into three segments by looking at the running execution context.

1. The first segment (ς_0–ς_7) has root context ϵ as its running context. Properties `z` and `f` are added to the global object in this segment. It ends with the first application of `f` on line 12, which introduces execution context τ_1.

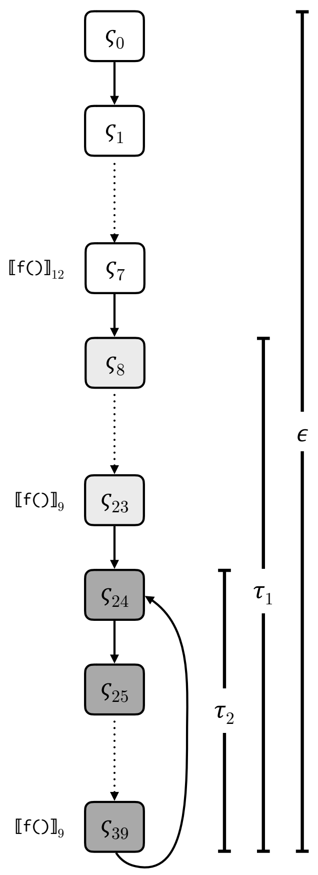


Figure 6. Overview of the finite flow graph generated by the flow analysis for the non-terminating program in Figure 5. State ς_7 represents the evaluation of the application of function `f` on line 12. States ς_{23} and ς_{39} represent the evaluation of the application of function `f` on line 9. After stepping state ς_{39}, the flow graph computation reaches a fixpoint by transitioning to the previously seen state ς_{24}. The states and transitions between states ς_{24}–ς_{39} are similar to those between states ς_8–ς_{23}. The three vertical bars indicate the extent of the three execution contexts, and the coloring of each state represents the running execution context: white for the root context ϵ, light gray for τ_1, and dark gray for τ_2. Details of the program states and transitions are given in Figure 7.

2. The second segment (ς_7–ς_{23}) represents the first execution of `f` with τ_1 as running context and runs until the second application of `f` .
3. The third segment (ς_{24}–ς_{39}) represents all subsequent executions of `f` as a result of the function application on line 9, and has execution context τ_2 as its running context.

The details of the program states and transitions between them are given in Figure 7. From the details we observe that the store reaches a fixpoint in state ς_{18}, denoted by σ_5 and obtained after adding property `x` to object `o`. This allows the control flow to reach a fixpoint in state ς_{39}, where applications of `f` result in an identical execution context τ_2 containing caller store σ_5. As a result, states in the third segment are similar to the states in the second segment, only differing in their stacks (execution context τ_2 and stack store Ξ_2 in the third segment instead of τ_1 and Ξ_1 in the second segment, respectively) and value stores (σ_5 in the third segment, subsuming the value stores appearing in the second segment).

7.2. Address-Based Purity Analysis

Based on the flow graph computed by flow analysis, we perform purity analysis by looking at the addresses of resources as described in Section 3. For simplicity, we only distinguish between pure

$$\varsigma_0 : \mathbf{ev}(\llbracket \texttt{z=0;}\ \underbrace{\texttt{function f() \{...\};}}_{}\ \texttt{f()} \rrbracket,\ \rho_0, \sigma_0, \langle\rangle, \epsilon, \Xi_0)$$

(with f over the brace)

$$\text{where } \rho_0 = [\,],\quad \sigma_0 = [a_0 \mapsto \omega_0],\quad \Xi_0 = [\,]$$

↧ $\{\mathbf{Wp}(a_0, \llbracket \texttt{f} \rrbracket)\}$

$$\varsigma_1 : \mathbf{ev}(\llbracket \texttt{z=0} \rrbracket, \rho_0, \sigma_1, \phi_0 : \langle\rangle, \epsilon, \Xi_0)$$
$$\begin{aligned} \text{where } \sigma_1 &= \sigma_0 \sqcup [a_f \mapsto \omega_f, a_0 \mapsto \omega_1] \\ \phi_0 &= \mathbf{sq}(f \llbracket \texttt{f()} \rrbracket, \rho_0) \\ \omega_1 &= \omega_0[\llbracket \texttt{f} \rrbracket \mapsto \{a_f\}] \\ \omega_f &= [\text{“call”} \mapsto \{c_f\}] \\ c_f &= (f, \rho_0) \\ a_f &= \mathit{allocFun}(f, \ldots) \end{aligned}$$

↧

$$\varsigma_2 : \mathbf{ev}(\llbracket \texttt{0} \rrbracket, \rho_0, \sigma_1, \phi_1 : \phi_0 : \langle\rangle, \epsilon, \Xi_0)$$
$$\text{where } \phi_1 = \mathbf{as}(\llbracket \texttt{z} \rrbracket, \rho_0)$$

↧

$$\varsigma_3 : \mathbf{ko}(\{\mathbf{num}\}, \sigma_1, \phi_1 : \phi_0 : \langle\rangle, \epsilon, \Xi_0)$$

↧ $\{\mathbf{Wp}(a_0, \llbracket \texttt{z} \rrbracket)\}$

$$\varsigma_4 : \mathbf{ko}(\{\mathbf{num}\}, \sigma_2, \phi_0 : \langle\rangle, \epsilon, \Xi_0)$$
$$\begin{aligned} \text{where } \sigma_2 &= \sigma_1 \sqcup [a_0 \mapsto \omega_2] \\ \omega_2 &= \omega_1[\llbracket \texttt{z} \rrbracket \mapsto \{\mathbf{num}\}] \end{aligned}$$

↧

$$\varsigma_5 : \mathbf{ev}(f, \rho_0, \sigma_2, \phi_2 : \langle\rangle, \epsilon, \Xi_0)$$
$$\text{where } \phi_2 = \mathbf{sq}(\llbracket \texttt{f()} \rrbracket, \rho_0)$$

↧

$$\varsigma_6 : \mathbf{ko}(\{\mathbf{undef}\}, \sigma_2, \phi_2 : \langle\rangle, \epsilon, \Xi_0)$$

↧

$$\varsigma_7 : \mathbf{ev}(\llbracket \texttt{f()} \rrbracket, \sigma_2, \langle\rangle, \epsilon, \Xi_0)$$

↧ $\{\mathbf{Rp}(a_0, \llbracket \texttt{f} \rrbracket)\}$

$$\varsigma_8 : \mathbf{ev}(\llbracket \texttt{var o; ...; f()} \rrbracket, \rho_1, \sigma_3, \langle\rangle, \tau_1, \Xi_1)$$
$$\begin{aligned} \text{where } \rho_1 &= \rho_0[\llbracket \texttt{o} \rrbracket \mapsto a_v] \\ \sigma_3 &= \sigma_2 \sqcup [a_v \mapsto \{\mathbf{undef}\}] \\ a_v &= \mathit{allocVar}(\llbracket \texttt{o} \rrbracket, \ldots) \\ \tau_1 &= (\llbracket \texttt{f()} \rrbracket_{12}, c_f, \langle\rangle, a_0, \sigma_2) \\ \Xi_1 &= \Xi_0 \sqcup [\tau_1 \mapsto \{(\epsilon, \Xi_0)\}] \end{aligned}$$

↧

$$\varsigma_9 : \mathbf{ev}(\llbracket \texttt{var o} \rrbracket, \rho_1, \sigma_3, \phi_3 : \langle\rangle, \tau_1, \Xi_1)$$
$$\text{where } \phi_3 = \mathbf{sq}(\llbracket \texttt{o=\{\}; ...; f()} \rrbracket, \rho_1)$$

↧

$$\varsigma_{10} : \mathbf{ko}(\{\mathbf{undef}\}, \sigma_3, \phi_3 : \langle\rangle, \tau_1, \Xi_1)$$

↧

$$\varsigma_{11} : \mathbf{ev}(\llbracket \texttt{o = \{\}} \rrbracket, \rho_1, \sigma_3, \phi_4 : \langle\rangle, \tau_1, \Xi_1)$$
$$\text{where } \phi_4 = \mathbf{sq}(\llbracket \texttt{o.x=123; ...; f()} \rrbracket, \rho_1)$$

↧

$$\varsigma_{12} : \mathbf{ev}(\llbracket \texttt{\{\}} \rrbracket, \rho_1, \sigma_3, \phi_5 : \phi_4 : \langle\rangle, \tau_1, \Xi_1)$$
$$\text{where } \phi_5 = \mathbf{as}(\llbracket \texttt{o} \rrbracket, \rho_1)$$

↧

$$\varsigma_{13} : \mathbf{ko}(\{a_o, \sigma_3\}, \phi_5 : \phi_4 : \langle\rangle, \tau_1, \Xi_1)$$

↧ $\{\mathbf{Wv}(a_v, \llbracket \texttt{o} \rrbracket)\}$

$$\varsigma_{14} : \mathbf{ko}(\{a_\omega\}, \sigma_4, \phi_4 : \langle\rangle, \tau_1, \Xi_1)$$
$$\text{where } \sigma_4 = \sigma_3[a_v \mapsto \{a_\omega\}]$$

↧

$$\varsigma_{15} : \mathbf{ev}(\llbracket \texttt{o.x=123} \rrbracket, \rho_1, \sigma_4, \phi_6 : \langle\rangle, \tau_1, \Xi_1)$$
$$\text{where } \phi_6 = \mathbf{sq}(\llbracket \texttt{z=4; f()} \rrbracket, \rho_1)$$

↧

$$\varsigma_{16} : \mathbf{ev}(\llbracket \texttt{123} \rrbracket, \rho_1, \sigma_4, \phi_7 : \phi_6 : \langle\rangle, \tau_1, \Xi_1)$$
$$\text{where } \phi_7 = \mathbf{st}(\llbracket \texttt{o} \rrbracket, \llbracket \texttt{x} \rrbracket, \rho_1)$$

↧

$$\varsigma_{17} : \mathbf{ko}(\{\mathbf{num}\}, \sigma_4, \phi_7 : \phi_6 : \langle\rangle, \tau_1, \Xi_1)$$

↧ $\{\mathbf{Rv}(a_v, \llbracket \texttt{o} \rrbracket), \mathbf{Wp}(a_\omega, \llbracket \texttt{x} \rrbracket)\}$

$$\varsigma_{18} : \mathbf{ko}(\{\mathbf{num}\}, \sigma_5, \phi_6 : \langle\rangle, \tau_1, \Xi_1)$$
$$\begin{aligned} \text{where } \sigma_5 &= \sigma_4 \sqcup [a_\omega \mapsto \omega_3] \\ \omega_3 &= \omega_2[\llbracket \texttt{x} \rrbracket \mapsto \{\mathbf{num}\}] \end{aligned}$$

↧

$$\varsigma_{19} : \mathbf{ev}(\llbracket \texttt{z=4} \rrbracket, \rho_1, \sigma_5, \phi_8 : \langle\rangle, \tau_1, \Xi_1)$$
$$\phi_8 = \mathbf{sq}(\llbracket \texttt{f()} \rrbracket_9, \rho_1)$$

↧

$$\varsigma_{20} : \mathbf{ev}(\llbracket \texttt{4} \rrbracket, \rho_1, \sigma_5, \phi_9 : \phi_8 : \langle\rangle, \tau_1, \Xi_1)$$
$$\text{where } \phi_9 = \mathbf{as}(\llbracket \texttt{z} \rrbracket, \rho_1)$$

↧

$$\varsigma_{21} : \mathbf{ko}(\{\mathbf{num}\}, \sigma_5, \phi_9 : \phi_8 : \langle\rangle, \tau_1, \Xi_1)$$

↧ $\{\mathbf{Wp}(a_0, \llbracket \texttt{z} \rrbracket)\}$

$$\varsigma_{22} : \mathbf{ko}(\{\mathbf{num}\}, \sigma_5, \phi_8 : \langle\rangle, \tau_1, \Xi_1)$$

↧

$$\varsigma_{23} : \mathbf{ev}(\llbracket \texttt{f()} \rrbracket_9, \rho_1, \sigma_5, \langle\rangle, \tau_1, \Xi_1)$$

↧ $\{\mathbf{Rp}(a_0, \llbracket \texttt{f} \rrbracket)\}$

$$\varsigma_{24} : \mathbf{ev}(\llbracket \texttt{var o; ...; f()} \rrbracket, \rho_1, \sigma_5, \langle\rangle, \tau_2, \Xi_2)$$
$$\begin{aligned} \text{where } \tau_2 &= (\llbracket \texttt{f()} \rrbracket_{12}, c_f, \langle\rangle, a_0, \sigma_5) \\ \Xi_2 &= \Xi_1 \sqcup [\tau_2 \mapsto \{(\epsilon, \Xi_1)\}] \end{aligned}$$

↧

...

↧

$$\varsigma_{39} : \mathbf{ev}(\llbracket \texttt{f()} \rrbracket_9, \rho_1, \sigma_5, \langle\rangle, \tau_2, \Xi_2)$$

↧ $\mathbf{Rp}(a_0, \llbracket \texttt{f} \rrbracket)$

ς_{24}

Figure 7. Details of the flow analysis for the example program depicted in Figure 5. The transition relation $\mapsto$ is annotated with a set of effects when this set is not empty.

and impure methods and disregard observers in the example. This means that we are only interested in write effects and only require the purity effect map $P \in \mathit{Purity}$ of our purity analysis but not maps Read and Obs.

Graph traversal starts at state ς_0 with $P = []$. The property write effects that occur in the root context ϵ do not affect P because they do not occur in the context of a function execution. The outgoing edge of state ς_{13} is annotated with variable write effect $\mathbf{Wv}(a_v, [\![\texttt{o}]\!])$ in running execution context τ_1 with caller store σ_2. Because $a_v \notin \mathrm{Dom}(\sigma_2)$, stack traversal in state ς_{13} is limited to the running context τ_1, and map P is not modified in the process:

$$\begin{aligned} \mathit{travStack}_W(\mathit{eff}_1, a_v, \varsigma_{13}, \varnothing, \{\tau_1\}, P) &= \mathit{travStack}_W(\mathit{eff}_1, a_v, \varsigma_{13}, \{\tau_1\}, \varnothing, P) \\ &= P \\ \text{where } \mathit{eff}_1 &= \mathbf{Wv}(a_v, [\![\texttt{o}]\!]) \end{aligned}$$

When purity analysis visits state ς_{17}, it is the case that $a_\omega \notin \mathrm{Dom}(\sigma_2)$, so again P is not modified:

$$\begin{aligned} \mathit{travStack}_W(\mathit{eff}_2, a_\omega, \varsigma_{17}, \varnothing, \{\tau_1\}, P) &= \mathit{travStack}_W(\mathit{eff}_2, a_\omega, \varsigma_{17}, \{\tau_1\}, \varnothing, P) \\ &= P \\ \text{where } \mathit{eff}_2 &= \mathbf{Wp}(a_\omega, [\![\texttt{x}]\!]) \end{aligned}$$

However, when the purity analysis visits state ς_{21}, it processes property write effect eff_3 targeting the global object allocated at address a_0 which is mapped in the caller store: $a_0 \in \sigma_2$. Because eff_3 is observable to the caller of `f`, function `f` is marked as a procedure in P:

$$\begin{aligned} \mathit{travStack}_W(\mathit{eff}_3, a_0, \varsigma_{21}, \varnothing, \{\tau_1\}, P) &= \mathit{travStack}_W(\mathit{eff}_3, a_0, \varsigma_{21}, \{\tau_1\}, \{\epsilon\}, P') \\ &= \mathit{travStack}_W(\mathit{eff}_3, a_0, \varsigma_{21}, \{\tau_1, \epsilon\}, \varnothing, P') \\ &= P' \\ \text{where } P' &= P \sqcup [[\![\texttt{f}]\!] \mapsto \mathsf{proc}] \\ \mathit{eff}_3 &= \mathbf{Wp}(a_0, [\![\texttt{z}]\!]) \end{aligned}$$

When purity analysis visits effects eff_1 and eff_2 again in the third segment of the flow graph, the result is different from visiting these effects in the second segment. In state ς_{29}, the running context is τ_2 with caller store σ_5 for which we have $a_v \in \sigma_5$ and $a_\omega \in \sigma_5$. Therefore, in state ς_{29} for effect eff_1, function `f` is (imprecisely) marked as a procedure in P:

$$\begin{aligned} &\mathit{travStack}_W(\mathit{eff}_1, a_v, \varsigma_{29}, \varnothing, \{\tau_2\}, P) \\ &\qquad = \mathit{travStack}_W(\mathit{eff}_1, a_v, \varsigma_{29}, \{\tau_2\}, \{\tau_1\}, P') \\ &\qquad = \mathit{travStack}_W(\mathit{eff}_1, a_v, \varsigma_{29}, \{\tau_2, \tau_1\}, \varnothing, P') \\ &\qquad = P' \\ &\text{where } P' = P \sqcup [[\![\texttt{f}]\!] \mapsto \mathsf{proc}] \end{aligned}$$

Purity analysis traverses the stack by moving from τ_2 to τ_1, because $\Xi_2(\tau_2) = \{(\_, \tau_1)\}$, but does not inspect underlying execution contexts because eff_1 is not observable outside τ_1.

In state ς_{33} and for effect eff_2 the address-based purity analysis similarly marks `f` to be a procedure as the result of loss of precision in the abstract semantics.

The result is that address-based purity will reach a fixpoint with $P = [[\![\texttt{f}]\!] \mapsto \mathsf{proc}]$, even when the observable write effect for the property store on line 8 would not be present.

7.3. *Purity Analysis With Lexical Freshness*

The freshness analysis presented in Section 4 is able to remove the false positives obtained in Section 7.2 for effects eff_1 and eff_2 in the third segment of the flow graph.

Variable write effect $\mathit{eff}_1 = \mathbf{Wv}(a_v, [\![\texttt{o}]\!])$ in state ς_{29} occurs in running context $\tau_2 = (\_, (f, \rho_0), \ldots)$ that executes function `f`. Because predicate $\mathit{isLocal}(a_v, f)$ holds, writing to local variable `o` is intercepted in the state handler of purity analysis and classified as non-observable:

$$handle_P(\mathit{eff}_2, \tau_2, P) = P$$

For object property write effect $\mathit{eff}_2 = \mathbf{Wp}(a_\omega, [\![\mathtt{x}]\!])$, object freshness analysis determines that variable `o` only points to fresh objects on line 7. It propagates map $F \in \mathit{Fresh}$ through the flow graph, starting with $F_0 = []$ in initial state ς_0. Although in ς_1 property `z` on the global object is written, the statement syntactically assigns a to a variable and because initial expression `0` is not fresh, we have $F_1 = [[\![\mathtt{z}]\!] \mapsto \mathbf{unfr}]$. The fact that this property write is considered as a (syntactic) variable write in the context of freshness analysis is not problematic, because our flow and purity analysis correctly distinguish between variable and property effects, regardless of syntactic appearance.

The next variable assignment expression is evaluated in state ς_{11}, where freshness analysis updates F_1 to $F_2 = F_1 \sqcup [[\![\mathtt{o}]\!] \mapsto \mathbf{fr}]$. At this point no variable assignment or other expressions in other program states add new information to F_2 and so object freshness analysis reaches a fixpoint after visiting all states once.

Variable `o` is marked as fresh when visiting state ς_{33} with frame $\phi_7 = \mathbf{st}([\![\mathtt{o}]\!], [\![\mathtt{x}]\!], \rho_1)$ on top of the local continuation stack. Because $\mathit{fresh}([\![\mathtt{o}]\!], \tau_2, F_2) = \mathbf{fr}$, property write on line 7 is considered as unobservable to callers of function `f`:

$$handle_P(\mathit{eff}_2, \varsigma_{33}, P) = P$$

As a result, purity analysis with lexical freshness reaches a fixpoint with $P = []$ when line 8 is removed from the example program in Figure 5 (but still considers `f` impure in the unmodified program).

8. EXPERIMENTS

In this section we report on several experiments we ran to evaluate the purity analysis presented in this paper. Before describing the actual evaluation of our approach and its results in terms of correctness, soundness, precision, and performance (Section 8.3), we give an overview of the abstract machine configurations (Section 8.1) and benchmark programs (Section 8.2) used during evaluating. We conclude the section with a comparison of our approach to existing tools in terms of precision (Section 8.4).

8.1. Abstract Machine Configurations

The abstract machine implementation is parameterized to be able to express both concrete and abstract semantics. To perform the evaluation, we use a concrete and an abstract configuration.

- In the *concrete configuration*, the abstract machine is configured with a concrete value store allocator (Section 2.2.7) that uses unique natural numbers as addresses. This configuration also equips the abstract machine with a concrete lattice for values and operations, which borrows from the underlying JavaScript runtime. In this configuration, the AAC stack store allocator (Section 2.2.4) guarantees full call/return precision. The value store is modified using strong updating. As a result, the concrete configuration performs flow analysis with *full precision*, because no unnecessary merging occurs.
- The *abstract configuration* uses the abstract 0CFA value store address allocator (Section 2.2.7), which uses syntactic elements as addresses. This configuration uses a type lattice in which primitive values are abstracted into sets of primitive types. The stores are weakly updated, except when abstract counting allows strong updating for the value store (Section 6). Using the abstract configuration guarantees that the computed flow analysis is finite, but at the cost of introducing *imprecision*.

Benchmark	Suite	LOC	Functions
`access-nbody`	SunSpider	165	11
`controlflow-recursive`	SunSpider	21	3
`crypto-sha1`	SunSpider	210	8
`math-spectral-norm`	SunSpider	46	5
`tree-add`	JOlden	157	6
`navier-stokes`	Octane	384	29
`richards`	Octane	496	32
`bisort`	JOlden	230	11
`em3d`	JOlden	242	13
`mst`	JOlden	245	18

Table I. Overview of the 10 selected benchmarks. *Suite* is the benchmark suite from which the benchmark originates. *LOC* is the approximate number of lines of code. *Functions* is the total number of functions called in the benchmark under abstract semantics.

8.2. Benchmarks

8.2.1. Type of Benchmark Programs During implementation and evaluation, we used two types of test programs:

1. Small *unit test programs* that test a specific feature that is challenging for purity analysis and the underlying flow analysis, such as higher-order functions, dynamic property expressions, and prototypal inheritance. These programs are designed in such a way that it is possible to manually specify the expected outcome of an analysis. A unit test therefore consists of a program paired with its expected outcome.
2. Larger *existing benchmark programs* taken from the SunSpider¶ and Octane‖ JavaScript benchmark suites, and programs from the Java JOlden** benchmark suite that we converted to JavaScript. (Section 8.4).

8.2.2. Benchmark Programs In the evaluation (Section 8.3), we report on the precision and performance of purity analysis on a selected set of 10 existing benchmark programs that each contain a number of challenges for our purity analysis. Table I lists these benchmark programs together with their size and number of functions that are called under abstract semantics. We give a short description of the benchmark programs.

- `access-nbody` is a SunSpider benchmark program that models the orbits of Jovian planets. It focuses mostly on arithmetic, loops, and property access.
- `controlflow-recursive` is a SunSpider benchmark program that tests different recursive patterns by invoking three different recursive pure functions: Fibonacci, tak, and Ackermann.
- `crypto-sha1` is a Sunspider benchmark program that tests cryptographic functions, and mostly covers bitwise operations and string operations. The majority of the functions that are actually called in this benchmark are pure functions.
- `math-spectral-norm` is a SunSpider benchmark program that calculates the spectral norm of an infinite matrix. It contains arrays and loops.
- `tree-add` is a JOlden benchmark program that computes a recursive sum of values in a balanced B-tree. The code in this program makes heavy use of objects and is predominantly recursive and functional in nature.
- `navier-stokes` is an Octane benchmark program implementing a 2D NavierStokes equations solver. The program passes around numeric arrays between functions that update these arrays in place, making these functions impure.

¶`https://www.webkit.org/perf/sunspider/sunspider.html`
‖`https://developers.google.com/octane`
**`ftp://ftp.cs.umass.edu/pub/osl/benchmarks`

- `richards` is an Octane benchmark program that simulates an OS kernel. This program mostly covers property load and store, and function and method calls.
- `bisort` is a JOlden benchmark implementation of the traveling salesman problem. It uses a binary tree as data structure and focuses on recursion and property load and store.
- `em3d` is a JOlden benchmark program that simulates electromagnetic wave propagation in a 3D object. It uses a singly linked list as data structure, and contains mostly loops and property load and store.
- `mst` is a JOlden benchmark program that computes the minimum spanning tree of a graph. The program maintains an array of singly linked lists, and focuses on loops, objects, and method calls.

We chose these programs in a pragmatic way, looking for the presence of a variety of control flow, data structures, and types of effects, but that were not too complex to analyze in terms of the use of JavaScript semantics and built-in functions and objects. For example, benchmark `crypto-sha1` has, as can reasonably be expected, a number of pure functions, enabling us to evaluate whether our purity analysis implementation is capable of detecting them. On the other end of the spectrum is, for example, the `navier-stokes` benchmark, which passes around and manipulates arrays and in which most functions are therefore impure.

8.3. *Evaluation*

We report on the correctness, soundness, precision, and performance of purity analysis results on our set of unit tests and benchmark programs.

8.3.1. Correctness When testing correctness, we check that the actual outcome of concrete purity analysis is equal to the expected outcome. We manually specified the expected function classification (pure, observer, or procedure) for an extensive set of unit tests and benchmark programs. We mechanically checked that the actual results of concrete purity analysis without freshness analysis are equal to the expected outcome. We then verified that this remains the case with freshness analysis enabled. From this evaluation we conclude that, for the set of programs under test, our implementation is correct.

We verified correctness of the underlying flow analysis in the same manner.

8.3.2. Soundness The result of abstract purity analysis is sound when it reflects all outcomes that are computed by concrete purity analysis. We mechanically checked whether the classifications computed by abstract purity analysis subsume the concrete computed classifications after abstraction [2]. For example, when abstract purity analysis predicts a function to be pure while the concrete analysis computes it as a procedure (or an observer), then that constitutes an unsound result for the abstract analysis. We conclude that the results of abstract purity analysis for our set of test programs are sound.

We verified soundness of the underlying flow analysis in the same manner.

8.3.3. Precision Precision is tested using a similar setup as used for soundness. A sound abstract purity analysis is allowed to overapproximate, i.e., is allowed to predict possibilities that do not occur under concrete semantics. We quantify abstract analysis precision by measuring the amount of *false positives*, which are abstract results that are less precise than their concrete counterpart. For example, when abstract purity analysis predicts a function to be a procedure while the concrete analysis computes it as pure (or an observer), then that constitutes a false positive for the abstract analysis. The fewer false positives, the higher the precision of the abstract analysis.

For the benchmarks in Table I, we determined the precision of our abstract purity analysis with freshness enabled and also compared it to the abstract purity analysis without freshness to assess the impact of lexical freshness analysis.

Concrete vs. abstract analysis The concrete purity analysis without lexical freshness computes effect classes with full precision. The abstract purity analysis *with* freshness analysis enabled is our

	— Concrete —			— Abstract —		
Benchmark	pure	obs	proc	pure	obs	proc
`access-nbody`	6	1	4	6	1	4
`controlflow-recursive`	3	0	0	3	0	0
`crypto-sha1`	7	0	1	7	0	1
`math-spectral-norm`	2	0	3	2	0	3
`tree-add`	3	0	2	2	1	2
`navier-stokes`	3	0	24	3	0	24
`richards`	8	1	23	8	1	23
`bisort`	5	0	5	5	0	5
`em3d`	9	0	3	3	0	9
`mst`	8	0	7	6	1	8

Table II. Purity analysis results for the concrete configuration without lexical freshness and the abstract configuration with freshness enabled. For each benchmark we report the number of functions in each effect class as determined by our analysis.

	— Without fresh —			— With fresh —		
Benchmark	pure	obs	proc	pure	obs	proc
`access-nbody`	1	1	9	6	1	4
`controlflow-recursive`	3	0	0	3	0	0
`crypto-sha1`	6	0	2	7	0	1
`math-spectral-norm`	2	0	3	2	0	3
`tree-add`	1	0	5	3	1	2
`navier-stokes`	3	1	25	3	1	25
`richards`	4	1	27	8	1	23
`bisort`	4	1	8	6	1	6
`em3d`	3	0	10	4	0	9
`mst`	5	1	12	7	1	10

Table III. Purity analysis results for the abstract configuration without lexical freshness and the abstract configuration with freshness enabled. For each benchmark we report the number of functions in each effect class as determined by our analysis.

most precise abstract analysis (see next experiment). Table II lists the number of functions in each effect class as reported by the analysis for the two configurations. Only functions that are actually applied in the concrete are considered.

The results show that our approach can classify functions with high precision for our set of 10 benchmarks. In 7 out of 10 benchmarks, the abstract purity analysis is as precise as the concrete analysis, i.e., fully.

Benchmark `em3d` is an outlier in the negative sense; we discuss this benchmark in more detail in the next experiment.

Impact of lexical freshness To assess the impact of lexical freshness analysis under the abstract configuration, we compare the precision of purity analysis without freshness (i.e., address-based) to the analysis with freshness enabled. Table II lists the number of functions in each effect class as reported by the analysis for the two settings. All functions that are applied during abstract interpretation are taken into account, which may be more functions than under concrete semantics.

From the benchmark results, we observe that incorporating freshness analysis improves precision in 7 out of 10 benchmarks. For example, `tree-add` is a JOlden benchmark that we converted from Java into JavaScript. Although it is a small benchmark, it exhibited poor precision when analyzed without freshness: only 1 pure function was detected out of 6 functions that are determined pure with freshness analysis enabled. The pattern in Figure 4, illustrating some of the weaknesses of address-only purity analysis, was distilled from this benchmark.

Generalizing over the cases in which freshness analysis has no or only a small impact on precision, we make two observations.

		— Without fresh —	— With fresh —
Benchmark	Flow time	Purity time	Purity time
`access-nbody`	ε	ε	ε
`controlflow-recursive`	ε	ε	ε
`crypto-sha1`	ε	ε	ε
`math-spectral-norm`	ε	ε	ε
`tree-add`	0'01"	ε	ε
`navier-stokes`	2'50"	3'18"	3'08"
`richards`	4'33"	0'50"	0'37"
`bisort`	0'07"	0'03"	0'01"
`em3d`	0'02"	0'12"	0'10"
`mst`	0'04"	0'15"	0'12"

Table IV. Flow and purity analysis timing under the abstract configuration. *Flow time* is the running time of the flow analysis creating a flow graph. *Purity time* is the running time of the purity analysis on top of the flow graph. We use ε to denote a running time smaller than 1 second.

- When the precision of the address-based purity analysis is already high, there is no or not much room left for freshness analysis to improve precision. This is the case for `navier-stokes` for example.
- The nature of the program can make it difficult for our freshness analysis to detect fresh resources. This is the case for `em3d` for example. Unlike `navier-stokes`, which is also a array-heavy benchmark, `em3d` is also object-heavy by storing objects in these arrays and featuring a linked list. The object-based freshness analysis from Section 4 does not track this type of object flow.

8.3.4. Performance The performance of an analysis is measured as the time it takes for an analysis to compute an answer and the memory it takes to do so. We tested the performance of our implementation to evaluate how it performs on current off-the-shelf hardware. The experiments were performed on a machine equipped with a quad-core 2.8 GHz Intel Core i7 processor and 16 GB of DDR3 memory. V8 [††] version 5.4.0 was used as JavaScript runtime. Although there were 4 physical (8 logical) cores available on the test machine, our implementation is single-threaded and does not benefit from additional cores.

Table IV lists, for our set of 10 benchmark programs, the running time of abstract flow analysis, and the running time of abstract purity analysis with and without lexical freshness.

From the results, we conclude that performance is suitable for use by clients that do not require on-the-fly results such as on-demand static analysis tools. We also observe that incorporating lexical freshness analysis improves the overall running time of the analysis because lexical freshness avoids stack traversal when effects occur on fresh resources.

We did not measure memory consumption in detail but we obtained the best performance when executing the test runner that produces Table IV with the available memory limited to 8 GB. Limiting available memory to 4 GB still allowed the test setup to finish but notably increased the running time of our test runner due to garbage collection overhead. Limiting the memory to 2 GB resulted in an out-of-memory error.

8.4. Comparison to Existing Work

Comparing our approach to existing approaches in terms of results is difficult because to the best of our knowledge, our previous work [27] is the first purity analysis that specifically targets JavaScript. The analysis that we present here extends our original analysis into one that additionally handles read effects and is therefore capable of classifying functions into a ternary scheme (pure/observer/procedure) instead of a binary one (pure/impure). Other related work (Section 5) focuses primarily on method purity for Java and analyzes benchmarks from the JOlden suite.

[††]`https://developers.google.com/v8/`

		ReImInfer			JPPA			JPure		
Program	JIPDA	=	+	−	=	+	−	=	+	−
`bisort`	5	5	0	0	5	0	0	5	0	0
`tree-add`	3	3	0	0	2	0	1	3	0	0
`em3d`	4	3	1	1	3	1	1	3	1	1
`mst`	6	5	0	1	5	0	1	5	0	1

Table V. Comparison between multiple purity analyses on benchmarks from the JOlden suite. Column *JIPDA* shows the number of functions identified as pure (pure or obs in our terminology) by our tool. Column = counts the methods which are detected as pure by both JIPDA and the other tool. Column + shows the number of functions detected as pure by the other tool but not by JIPDA. Column − shows the number of functions identified as pure by JIPDA but not by the other tool.

Programs `tree-add`, `bisort`, `em3d`, and `mst` are JOlden benchmarks that we manually converted to JavaScript. We compare our results (JIPDA) for these benchmarks with ReImInfer [17], JPPA [33], and JPure [29] in Table V. We manually verified and compared the results reported by each tool on every method or function that is present in both the original program and the JavaScript translation of the benchmark.

The results show that JIPDA is on par with currently existing tools. Only for `em3d`, JIPDA misses a pure function detected by the other tools, while the other tools miss a pure function that is correctly detected by JIPDA. Therefore, in that benchmark, every tool detect 4 out of 5 pure functions. For the `mst` benchmark, only JIPDA detects the 6 pure functions, while all the other tools miss one pure function. For `bisort` and `tree-add`, JIPDA detects all pure functions as pure, while JPPA misses one pure in `tree-add`.

From this experiment, we conclude that for our set of 10 benchmarks our approach is as precise as existing approaches and sometimes is able to detect more pure functions.

9. LIMITATIONS AND FUTURE WORK

Despite the fact that our purity analysis for JavaScript is capable of determining function purity with sufficient precision to be useful in a number of software engineering scenarios and applications (Section 10), there is still room for improvement. We identified the following limitations, and overcoming these limitations is considered future work.

9.1. *Soundness of Analyses*

We empirically demonstrated that the results produced by our implementation of our flow and purity analysis are sound for a suite of challenging benchmark programs. Empirical soundness was established by verifying that every abstract analysis result subsumes the corresponding concrete result [2]. We believe that this enables the use of our implementation in a variety of scenarios that require sufficient confidence in the soundness of the results. Using the analysis in an optimizing compiler, however, would require more thorough soundness guarantees.

Assuming the flow analysis is sound, we believe that formally proving soundness of address-based purity analysis (Section 3) would present no major obstacle because effects and address freshness represent minor additions to the original abstract state-space and semantics of the flow analysis, respectively. Proving correctness of concrete lexical object freshness analysis (Section 4.3) and proving it sound in the abstract semantics represents more challenging future work.

9.2. *Scalability*

The purpose of our implementation is to demonstrate the feasibility of using stack reachability for determining function purity and also for performing experiments. Performance and scalability were not our primary concerns and our experimental results show that scalability is an issue.

An inherent limitation of the purity analysis that we presented is that it is a whole-program analysis that can be costly to perform. In a similar but simpler setting, the worst-case computational cost of AAC flow analysis [20], the technique employed in our purity analysis, was found to be $O(n^8)$ [15], with n

A modular analysis, on the contrary, can be performed on parts of the program. This usually results in a more performant analysis, especially in scenarios where an input program is reanalyzed after only parts of it are modified. At this point it is unknown whether and how the techniques proposed in this work can be modularized. Other related work also describes tools that require an underlying whole-program analysis [23, 33], while alternate approaches—primarily those rooted in type systems—are modular in nature [13, 17, 29, 32].

9.3. *Limited JavaScript Semantics*

Real-world JavaScript applications include dynamic, asynchronous, and event-driven features, and web applications in particular also target the Document Object Model (DOM). These features are not modeled in our semantics and are not supported by our implementation, but would be time-consuming to model and implement on the level of a flow analysis. However, we believe these and other features are not essential to our approach to purity analysis and therefore out of scope for this work. For example, consider asynchronous computation in JavaScript, which is organized around the concept of "Jobs" and "Job Queues" [11]. A Job can only start when the call stack is empty and runs to completion, and therefore asynchronous computation does not interfere with how our analysis determines effects, freshness, and purity with respect to function application contexts. The DOM, as another example, can be accessed and manipulated by JavaScript as a collection of objects and (asynchronous) functions. Although the DOM is complex, it inherently does not pose any additional challenging difficulties for determining purity following our approach.

In summary, our approach to purity analysis works with any flow analysis that models store-allocated resources, effects on these resources, and a call stack that allows function calls to be associated with contexts, regardless of other language features and global environments that are supported.

9.4. *Limited Object Freshness Analysis*

The lexical object freshness analysis of Section 4.3 was designed as a supporting analysis for function purity analysis, and as such does not offer full precision, even when the underlying flow graph does. However, object freshness analysis still improves the precision and performance of our purity analysis. Modifying our object freshness analysis (or the underlying flow analysis) so that freshness analysis becomes fully precise in the concrete, and evaluating the impact on precision in the abstract, is an interesting but challenging avenue for future work. Storeless semantics [6] and access paths [22] are relevant and more formally established techniques to reason over the heap and value flow without requiring store semantics.

10. APPLICATIONS

In this section we discuss some immediate applications of the purity analysis we presented. We consider each application as an interesting avenue of future research.

10.1. *Referential Transparency*

Expressions that are referentially transparent can always be replaced by their value without changing the behavior of the program [35]. Referentially transparent expressions must be pure, but additionally should not allocate resources.

Our approach can be extended to including tracking of allocation effects, so that referential transparency can become a classification of functions in our purity analysis that is more precise than (i.e., "above") purity: all referentially transparent procedures are pure, but the inverse is not necessarily true. Referential transparency enables optimizations such as lazy evaluation, common subexpression elimination, memoization (Section 10.2), and parallelization (Section 10.3). We plan on performing this extension to investigate how and to which extent it is possible to exploit referential transparency to optimize concrete and abstract interpretation.

10.2. Memoization

Memoization is a fundamental optimization technique that avoids recomputing previously computed results by storing them in a table from which they can be looked up, effectively trading in space for time [1]. In the context of programming languages, a typical memoization target is a procedure call. If calling conditions match previously seen calling conditions, then execution of the function body is skipped and cached values are returned instead. If memoization is done at the program level (i.e., not in the interpreter) and only returned values are cached (but no other program state), then the memoized function must be pure. Although the goal of memoization is to increase performance, memoization itself introduces overhead because it stores cached results and requires lookups in this cache.

A memoization analysis would build upon purity analysis by requiring functions that are candidates for memoization to at least be pure in the strictest sense (Section 1). Unlike the purity analysis presented in this work, memoization also takes fresh resources as return values of functions into account. For example, if a function returns a fresh pair, and that pair is cached in the memo table, then the pair is shared between all callers of the memoized function. This is problematic if the pair is subsequently modified. A memoization analysis could disallow memoization for functions that are not referentially transparent. An additional verification on functions that return fresh resources would be to disallow subsequent modification of the returned resource, ensuring that resources cached as return values are immutable.

Another difficulty for memoization analysis is the common scenario in which resources are passed as arguments to functions. Memoization has to decide under which condition resources are equal. For example, if an array is passed as an argument, the memoization algorithm has to decide during memo lookup whether the argument array is equivalent to a cached entry. This can happen by comparing memory locations or by recursively comparing the contents of the array. We agree with Finifter et al. [13] that equivalence of arguments is a parameter of function memoization.

10.3. Parallelization

In Nicolay et al. [28], we describe a program transformation that automatically parallelizes higher-order imperative Scheme programs. The key idea in that work is to look at every series of nested `let` expressions that appear in a program, and decide where it is possible to evaluate binding expressions in parallel. The approach must be instantiated with a sufficiently precise dependence analysis, for dependent expressions cannot safely be evaluated in parallel. Expression e_2 is *dependent* on expression e_1 if e_1 is evaluated before e_2 and e_2 accesses or modifies a resource that was accessed or modified before by e_1.

The implementation of the underlying flow and dependence analysis in Nicolay et al. [28] has some limitations. It builds on the finite-state dependence analysis by Might and Prabhu [25]. It also does not separately model the individual elements of vectors but considers vectors to be a single resource, which leads to a precision penalty.

The approach presented in this paper overcomes these shortcomings. We use a pushdown analysis that offers more precise call/return precision than a classic finite-state analysis. Our side-effect analysis models the individual elements of an array (up to the precision offered by the value lattice). In future work we will recast the flow analysis with effects presented in this work into a dependence analysis with the goal of parallelizing higher-order imperative programs with more precision.

11. CONCLUSION

We presented a purity analysis for JavaScript that handles closures, higher-order functions, and prototypal inheritance. We employ pushdown flow analysis to compute the control and value flow of a program, while generating effects when variables and object properties are read and written.

By comparing addresses in effects with addresses in caller stores associated with active function executions on the call stack, we distinguish observable from unobservable side effects. Doing this for all read and write effects in all executions of a function enables us to classify functions as pure, observer, or procedure.

While flow analysis is adequate for tracking object flow in a program, it does however lose precision when the number of addresses it can choose from is limited. We therefore add a lexical freshness analysis for variables and objects to identify unobservable effects that would be considered observable by the address-based purity analysis. Adding freshness analysis improves precision of our purity analysis considerably, and reduces execution time, as demonstrated in our preliminary experiments.

When comparing our purity analysis against existing solutions, we find that our implementation is as precise, and sometimes more precise, than results produced by other tools. Despite its limitations, we believe our analysis is capable of determining function purity with sufficient precision to be useful in a number of software engineering scenarios.

ACKNOWLEDGMENTS

Jens Nicolay is funded by the SeCloud project sponsored by Innoviris, the Brussels Institute for Research and Innovation. Coen De Roover is funded by the *Cha-Q* project sponsored by IWT Vlaanderen. Quentin Stievenart is funded by the GRAVE project of the "Fonds Wetenschappelijk Onderzoek".

REFERENCES

1. U. A. Acar, G. E. Blelloch, and R. Harper. *Selective memoization*, volume 38. ACM, 2003.
2. E. S. Andreasen, A. Møller, and B. B. Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 31–36. ACM, 2017.
3. S. Artzi, A. Kiezun, D. Glasser, and M. D. Ernst. Combined static and dynamic mutability analysis. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 104–113. ACM, 2007.
4. M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on formal techniques for Java-like programs (FTfJP)*, 2004.
5. M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. Allowing state changes in specifications. In *Emerging Trends in Information and Communication Security*, pages 321–336. Springer, 2006.
6. M. Bozga, R. Iosif, and Y. Laknech. Storeless semantics and alias logic. In *ACM SIGPLAN Notices*, volume 38, pages 55–65. ACM, 2003.
7. L. R. Clausen. A java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
9. S. Dai, T. Wei, C. Zhang, T. Wang, Y. Ding, Z. Liang, and W. Zou. A framework to eliminate backdoors from response-computable authentication. In *2012 IEEE Symposium on Security and Privacy*, pages 3–17. IEEE, 2012.
10. C. Earl, M. Might, and D. V. Horn. Pushdown control-flow analysis of higher-order programs. In *Proceedings of the 2010 Workshop on Scheme and Functional Programming (Scheme 2010)*, Montreal, Quebec, Canada, August 2010.
11. Ecma. ECMAScript Language Specification. `https://tc39.github.io/ecma262/`, 2017.
12. M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 314–. ACM, 1987.
13. M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in java. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 161–174. ACM, 2008.
14. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.
15. T. Gilray, S. Lyde, M. D. Adams, M. Might, and D. V. Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43th Annual ACM Symposium on the Principles of Programming Languages (POPL 2016)*, St. Petersburgh, Florida, USA, January 2016.
16. I. Harrison, Williams Ludwell. The interprocedural analysis and automatic parallelization of scheme programs. *LISP and Symbolic Computation*, 2(3-4):179–396, 1989.
17. W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. Reim & reiminfer: Checking and inference of reference immutability and method purity. In *ACM SIGPLAN Notices*, volume 47, pages 879–896. ACM, 2012.

18. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*, pages 41–55. Springer, 2011.
19. S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.
20. J. I. Johnson and D. Van Horn. Abstracting abstract control. In *Proceedings of the 10th ACM Symposium on Dynamic languages*, pages 11–22. ACM, 2014.
21. V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. Jsai: a static analysis platform for javascript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132. ACM, 2014.
22. J. Lerch, J. Späth, E. Bodden, and M. Mezini. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 619–629. IEEE, 2015.
23. R. Madhavan, G. Ramalingam, and K. Vaswani. Purity analysis: An abstract interpretation formulation. In *Static Analysis*, pages 7–24. Springer, 2011.
24. M. Might and P. Manolios. *A posteriori* soundness for non-deterministic abstract interpretations. In *Proceedings of the 10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2009)*, Savannah, Georgia, USA, January 2009.
25. M. Might and T. Prabhu. Interprocedural dependence analysis of higher-order programs via stack immutability. In *Proceedings of the 2009 Workshop on Scheme and Functional Programming (Scheme 2009)*, Boston, Massachussetts, USA, August 2009.
26. M. Might and O. Shivers. Improving flow analyses via γcfa: abstract garbage collection and counting. In *ACM SIGPLAN Notices*, volume 41, pages 13–25. ACM, 2006.
27. J. Nicolay, C. Noguera, C. D. Roover, and W. D. Meuter. Detecting function purity in javascript. In *Proceedings of the Fifteenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2015)*, pages 101–110, Bremen, DE, September 2015.
28. J. Nicolay, C. D. Roover, W. D. Meuter, and V. Jonckers. Automatic parallelization of side-effecting higher-order scheme programs. In *Proceedings of the Eleventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2011)*, pages 185–194, Williamsburg, VA, USA, September 2011.
29. D. J. Pearce. Jpure: a modular purity system for java. In *Compiler Construction*, pages 104–123. Springer, 2011.
30. M. Pitidis and K. Sagonas. Purity in erlang. In *Implementation and Application of Functional Languages*, pages 137–152. Springer, 2011.
31. D. Prountzos, R. Manevich, K. Pingali, and K. S. McKinley. A shape analysis for optimizing parallel graph programs. In *ACM SIGPLAN Notices*, volume 46, pages 159–172. ACM, 2011.
32. L. Rytz, N. Amin, and M. Odersky. A flow-insensitive, modular effect system for purity. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, page 4. ACM, 2013.
33. A. Salcianu and M. Rinard. Purity and side effect analysis for java programs. *Lecture notes in computer science*, pages 199–215, 2005.
34. O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Citeseer, 1991.
35. H. Søndergaard and P. Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27(6):505–517, 1990.
36. D. Van Horn and M. Might. Abstracting abstract machines. In *ACM Sigplan Notices*, volume 45, pages 51–62. ACM, 2010.
37. D. Vardoulakis and O. Shivers. Cfa2: A context-free approach to control-flow analysis. In *Programming Languages and Systems*, pages 570–589. Springer, 2010.