Detecting Function Purity in JavaScript

Jens Nicolay, Carlos Noguera, Coen De Roover, Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
{jnicolay, cnoguera, cderoove, wdmeuter}@vub.ac.be

***Abstract*—We present an approach to detect function purity in JavaScript. A function is pure if none of its applications cause observable side-effects. The approach is based on a pushdown flow analysis that besides traditional control and value flow also keeps track of write effects. To increase the precision of our purity analysis, we combine it with an intraprocedural analysis to determine freshness of variables and object references. We formalize the core aspects of our analysis, and discuss our implementation used to analyze several common JavaScript benchmarks. Experiments show that our technique is capable of detecting function purity, even in the presence of higher-order functions, dynamic property expressions, and prototypal inheritance.**

I. Introduction

Mathematically speaking, the only observable effect of a function is turning input arguments into a resulting value. However, in imperative programming languages like JavaScript, any callable entity (function, constructor, method, ...) can do more than that. Anything a JavaScript function does besides producing a value, is called a side-effect of that function. A side-effect is observable when it is visible from the point of view of the caller. This happens when during function application the function modifies state visible to the caller. A function is pure if it does not generate observable side-effects.

Research in different areas has demonstrated that purity aids program understanding, specification, testing, debugging, and maintenance [1]. Therefore, detection, verification, or even enforcement of purity is useful for software engineering purposes. Purity facilitates establishing security and confidentiality aspects of applications. Pure functions are more secure than impure ones, because they interfere less with the rest of the application. Purity also potentially reduces the number of bugs, and makes it easier to reproduce bugs. Pure functions can be safely called from assertions, they allow for more and better program optimizations, and they can speed up concurrency analyses by eliminating non-interfering interleavings. Proving the absence of side-effects has many more advantages and applications, which are discussed more extensively in related work on purity and side-effect analysis (Section VII).

In JavaScript, observable side-effects are a consequence of assigning variables and storing property values in objects. Variables and objects are stored at specific memory locations, and writing to a memory location is considered to be an effect. Our notion of purity allows unobservable side-effects, therefore allowing functions that allocate and mutate memory locations to still be pure. In our approach pure functions may return locally allocated objects.

Functions only generate effects upon application. Therefore our notion of function purity — a function as a syntactic entity appearing in a program — is linked to the behavior of all its applications at runtime. We therefore consider a function pure *if all its applications are pure*. We develop a formal definition of purity in Sections III and IV.

A. Challenges

Developing a purity analysis for JavaScript is challenging in several ways.

Purity analysis needs to correctly track control and value flow. In the example program below, function `f` is pure, but `g` and `h` are not.

```
function f()                // pure
{
  var o={};
  function g(p) { h(p) }    // impure
  function h(q) { q.x=4 }   // impure
  g(o)
}
f()
```

Because functions `h` (directly) and `g` (indirectly) mutate an object that exists in the caller state of their applications, they are both impure. Upon the occurrence of a property write effect, the analysis has to walk the call stack to correctly handle the effect for every function application that is active.

While in the above example control flow is straightforward to determine, JavaScript features higher-order functions. 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}    // impure
3  function f(h) { var o={}; h(o)}  // impure
4  f(g)
```

It is clear that function `g` is impure, since it not only writes to global property `z`, but also mutates an object through its parameter. However, the purity analysis has to determine that the application of `h` in the body of `f` on line 3 applies function `g`, making function `f` impure because it indirectly mutates `z`.

It is not always straightforward to determine the correct effects. Because of the semantics of JavaScript, variable `x` in the example program below is actually a property of the global object.

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

Assigning to `x` on line 6 should generate a property write effect on the global object, identical to the effect generated on line 2, instead of a variable write effect. It is important to distinguish between variables and properties because variables and objects behave and therefore influence purity differently. 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.

JavaScript has closures, and unlike purity analyses for more traditional object-oriented languages without closures, purity analysis for JavaScript has to be able to handle free variables.

```
 1  function f()                        // pure
 2  {
 3    function g()                      // pure
 4    {
 5      var z = 10;
 6      function h() { z = 20 };   // impure
 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 applications of `h`, `g`, and `f`. Function `h` is impure 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.

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

```
 1  function f() { this.x = 10 }  // impure function
 2  new f();  // pure application
 3  f();      // impure application
```

When function `f` in the above example is invoked as a constructor (line 2), then it is a pure application since `this` in the body of `f` is bound to a fresh object. The regular function application on the next line is impure because `this` is bound to the global object. As a result, function `f` is impure.

B. *Overview of Our Approach*

Our approach for designing a purity analysis is based on a pushdown abstract interpretation of the program that integrates control flow, value flow, and effects.

We define a core imperative language that represents an interesting and non-trivial subset of standard JavaScript semantics (Section II). The semantics of this input language is expressed as an abstract machine that transitions between states. The abstract machine generates appropriate write effects caused by writing to variables and object properties. Starting from an initial evaluation state, all possible successor states are explored. This results in a flow graph, which is a finite representation of the runtime behavior (control flow, value flow, effects) of the input program.

We then examine the flow graph to determine whether a given function is pure or impure by looking at all applications of a certain function (Section III). If all applications of a function are pure, then the function itself is pure. A function application is pure if it does not generate observable side-effects, i.e., it does not mutate state visible to the caller, during application. A write effect is observable to a caller when the address that is written to is mapped in the caller store that is in effect at function entry.

Although checking addresses as described above corresponds exactly with our definition of purity, we find that this technique is problematic in a static analysis setting. Because an analysis has to complete in finite space and time, the set of addresses is made finite, which decreases the precision of our purity analysis. We therefore recover some of this precision by defining a small lexical and intraprocedural freshness analysis on top of a flow graph (Section IV). Variables that are declared in the corresponding function scope of an application are fresh, while an object reference is fresh in a top-level function application if the object the reference points to was allocated during that application. The purity analysis can use freshness of variables and object references to mask certain write effects, thereby increasing the precision of the purity analysis.

After explaining our approach in detail, the remainder of the paper discusses our implementation and experiments (Sections V and VI), and we give an overview of related work (Section VII).

The contributions presented in this paper are the following:

- We present an abstract machine for a core JavaScript-like language that tracks write effects generated by assignments to variables and object properties during interpretation.
- We introduce a purity analysis over a flow graph annotated with effects.
- We define an intraprocedural freshness analysis over a flow graph to improve precision of our purity analysis.
- We implement a purity analysis for a substantial subset of JavaScript, and experiment with it on several common JavaScript benchmarks.

II. Setting

Because function call and return is the dominant pattern in higher-order, functional programs, an analysis needs to model call/return precisely. For this reason, we use a pushdown analysis [2] and not a more classic finite-state analysis.

A. *Input Language*

In order to simplify the formalization of our approach, we work on a core functional language with assignment. This input language, depicted in Figure 1, most notably features objects as maps, higher-order functions, prototypal inheritance, and assignment. Although it is a small language, its set

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

Fig. 1. Input language.

of features is sufficiently challenging for performing purity analysis. Our implementation (Section V), used to validate our approach, supports a larger subset of traditional features like iteration, non-local return flow, and typical features of JavaScript, including type coercions and parts of the standard built-in functions and objects. We assume that every element in our input language has a unique label ℓ so that different occurrences of the same expression can be distinguished.

B. Semantics

The small-step semantics of the input language is expressed as an abstract machine [3] that transitions between evaluation (**ev**) and continuation (**ko**) states. The resulting machine is a variation on the CESIK$^\star\Xi$ abstract machine described in Johnson and Van Horn [2]. This machine actually is an *abstract* abstract machine since it operates on abstract values, although it can be parameterized to express concrete semantics.

Figure 2 shows the abstract state-space. The control (e), environment (ρ), store (σ), and value (d) components of the machine are standard.

Stacks are stored inside states and consist of a local continuation (ι) delimited by a meta-continuation ($\hat{\kappa}$). The local continuation is a (possibly empty) list of frames, while the meta-continuation is a calling context. Calling contexts are generated at call sites, except for the root calling context that is created at the start of program evaluation.

Calling contexts that are generated at call sites serve as stack addresses pointing to underlying stacks that are stored in a stack store (Ξ). A stack address contains five components: a call expression (e), a callable (c), a list of arguments (d_{arg}), a `this` pointer (a_{this}), and a caller store that is in effect at function entry (σ). Allocating stacks with this kind of precision describes unbounded stacks in a finite way with the precision offered by pushdown systems, i.e., with full call/return precision.

In the remainder of this section we detail the operation of the abstract machine.

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

$$\begin{gathered}
\mathcal{I}(e) = \mathbf{ev}(e, [], \sigma_0, \langle\rangle, \epsilon) \\
\text{where } \hat{\kappa}_0 = (e, \bot, \bot, a_0, \sigma_0) \\
\sigma_0 = [a_0 \mapsto []]
\end{gathered}$$

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

2) 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 [4], but not all allocation schemes are useful. We assume the presence of allocation functions $allocVar$ for allocating variables, $allocCtr$ for allocating constructor objects, $allocFun$ for allocating function objects, and $allocProto$ for allocating prototypes of function objects.

To express concrete semantics, we can take $Addr = \mathbb{N}$ and $allocX(e, \rho, \sigma, \iota, \hat{\kappa}) = 1 + \max(\mathrm{Dom}(\sigma))$, where $allocX$ is one of the allocation functions and Dom returns the domain of a function.

For abstract semantics, a monovariant allocation scheme (0CFA) would be $Addr = \mathsf{Exp}$ with $allocVar(e, \rho, \sigma, \iota, \hat{\kappa}) = e$, and similar definitions for the other allocators.

3) Simple Expressions: Function $evalSimple : \mathsf{Simple} \times Env \times Store \times \widehat{Kont} \mapsto D$ evaluates simple expressions: references and `this` expression. Looking up a reference means looking up its address in the lexical environment, returning the value associated with that address in the store.

$$\begin{gathered}
evalSimple(v, \rho, \sigma, \hat{\kappa}) = \sigma(a) \\
\text{if } v \in \mathrm{Dom}(\rho) \\
\text{where } a = \rho(v)
\end{gathered}$$

If the name is not available in the environment, then we perform a property lookup on the global object at address a_0.

$$\begin{gathered}
evalSimple(v, \rho, \sigma, \hat{\kappa}) = \omega(v) \\
\text{where } \omega = \sigma(a_0)
\end{gathered}$$

The value for a `this` expression is retrieved from the current calling context.

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

4) Transition Relation: In order to determine function purity, we need to be able to reason about write effects that occur as a result of mutating variables and object properties during evaluation. We make write effects explicit by modeling them on the transition relation that transitions between states: $(\mapsto) \sqsubseteq State \times State \times \mathcal{P}(E\!f\!f)$. Since reading and allocation of variables and objects in itself can never influence the purity

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

Fig. 2. State-space of the analysis.

of expressions, we are not interested in these effects and hence do not consider them.

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

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

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.

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

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.

$$
\begin{aligned}
\mathbf{ev}([\![\overbrace{v\,(s)}^{e}]\!], \rho, \sigma, \iota, \hat{\kappa}, \Xi) &\mapsto \mathit{evalCall}(c, d_{\text{arg}}, \sigma, \iota, \hat{\kappa}, \Xi, \hat{\kappa}') \\
\text{where } d_f &= \mathit{evalSimple}(v, \rho, \sigma, \hat{\kappa}) \\
d_{\text{arg}} &= \mathit{evalSimple}(s, \rho, \sigma, \hat{\kappa}) \\
a_f &\in d_f \\
\omega_f &= \sigma(a_f) \\
c &\in \omega_f(\text{call}) \\
\hat{\kappa}' &= (e, c, d_{\text{arg}}, a_0, \sigma)
\end{aligned}
$$

For a method call we additionally look up the method on the receiver, and we set the receiver as value for `this` in the new calling context.

$$
\begin{aligned}
\mathbf{ev}([\![\overbrace{s_0\,.\,v\,(s_1)}^{e}]\!], \rho, \sigma, \iota, \hat{\kappa}, \Xi) &\mapsto \mathit{evalCall}(c, d_{\text{arg}}, \sigma, \iota, \hat{\kappa}, \Xi, \hat{\kappa}') \\
\text{where } d_{\text{this}} &= \mathit{evalSimple}(s_0, \rho, \sigma, \hat{\kappa}) \\
d_{\text{arg}} &= \mathit{evalSimple}(s_1, \rho, \sigma, \hat{\kappa}) \\
a_{\text{this}} &\in d_{\text{this}} \\
d_f &\in \mathit{lookupProp}(v, a_{\text{this}}, \sigma) \\
a_f &\in d_f \\
\omega_f &= \sigma(a_f) \\
c &\in \omega_f(\text{call}) \\
\hat{\kappa}' &= (e, c, d_{\text{arg}}, a_{\text{this}}, \sigma)
\end{aligned}
$$

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

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

A constructor call allocates a new object on the heap, and sets a reference to this object as value for `this` in the new calling 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.

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

Function *evalCall* applies a function to an argument in a given context. It extends the static environment by binding the argument, and moves evaluation to the body of the function.

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

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

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

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

$$\begin{aligned}
\mathbf{ev}([\![s\,\textbf{.}\,v]\!],\rho,\sigma,\iota,\hat\kappa,\Xi)&\mapsto(\mathbf{ko}(d,\sigma,\iota,\hat\kappa,\Xi),\varnothing)\\
\text{where } d_r&=\mathit{evalSimple}(s,\rho,\sigma,\hat\kappa)\\
a&\in d_r\\
d&\in\mathit{lookupProp}(v,a,\sigma)
\end{aligned}$$

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

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

A declared variable is added to the lexical environment with a value of `undefined`, which is also the result of the entire "expression". Variable allocation does not generate an effect.

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

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

$$\begin{aligned}
\mathbf{ev}([\![\texttt{return}\ s]\!],\rho,\sigma,\iota,\hat\kappa,\Xi)&\mapsto(\mathbf{ko}(d,\rho,\sigma,\langle\rangle,\hat\kappa,\Xi),\varnothing)\\
\text{where } d&=\mathit{evalSimple}(s,\rho,\sigma,\hat\kappa)
\end{aligned}$$

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.

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

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

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

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

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

5) Function Exit: 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. In all other cases 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. For simplicity we always return a reference to the newly created object for a constructor call, regardless of how function exit is reached.

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

Similarly, we always return the current value when exiting from a function call.

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

6) Flow Graph: We determine function purity by reasoning about write 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: $\hat{\varsigma} \hookrightarrow \hat{\varsigma}' \iff \hat{\varsigma} \mapsto (\hat{\varsigma}', E)$. Evaluation can be expressed as computing the transitive closure of $\hookrightarrow$ after injection.

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

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

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

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 2). For finite programs the entire state space is then finite as well, and $\hookrightarrow$, which is monotonic, has a least fixpoint.

III. Purity Analysis

Using the flow graph from the previous section, we are able to determine function purity by examining all function applications. The result is a map P from functions to their effect class.

$$\begin{aligned}class \in Class &= \{\bot, \mathsf{pure}, \mathsf{impure}\}\\ P \in Purity &= \mathsf{Fun} \mapsto Class\end{aligned}$$

The effect class is a join semi-lattice in which $\bot \sqsubset \mathsf{pure} \sqsubset \mathsf{impure}$ and $\mathsf{pure} \sqcup \mathsf{impure} = \mathsf{impure}$.

Our definition of purity requires the set of active function applications on the stack, with each active application coupled to the caller store that was in effect at the respective call site. Function *contexts* collects the set of active calling contexts by walking over the stack. In order to guard against infinite recursion, helper function *contexts*$*$ keeps a set of seen contexts (*ctxs*).

$$\begin{aligned}contexts(\hat{\kappa},\Xi) &= contexts^*(\hat{\kappa},\Xi,\varnothing)\\ contexts^*(\hat{\kappa},\Xi,ctxs) &= ctxs \text{ if } \hat{\kappa} \in ctxs\\ contexts^*(\hat{\kappa},\Xi,ctxs) &= \bigcup_{(\_,\hat{\kappa}')\in\Xi(\hat{\kappa})} contexts^*(\hat{\kappa}',\Xi,ctxs\cup\{\hat{\kappa}\})\end{aligned}$$

The caller store is needed to check whether effects that occur during function application are observable or not. If a write effect occurs on an address that is mapped in the caller store, then the effect is observable. If the address is not in the domain of the caller store, then the effect is local to the application and can be masked. The following write effect handler deals with both variable and property effects by considering the address of the effect. If the address is mapped in the caller store of an active function application, then the effect is observable from the point of view of the caller, and the function is marked as impure.

$$\begin{aligned}handle(\hat{\varsigma},\mathbf{Wv}/\mathbf{p}(a,\_)) &= P\\ \text{where } P &= \{[f \mapsto \mathsf{impure}] \mid a \in \mathrm{Dom}(\sigma)\\ &\qquad \wedge\ (e,(f,\rho),d_{\text{arg}},\sigma) \in ctxs\}\\ ctxs &= contexts(\hat{\kappa},\Xi)\\ (\ldots,\hat{\kappa},\Xi) &= \hat{\varsigma}\end{aligned}$$

We define relation $\mapsto_{\text{purity}}$ that navigates flow graph G_e and propagates information about function purity. For every transition, it delegates to a handler for every effect that occurs on that transition. We mark every active function pure here to avoid burdening effect handlers with dealing with this case. As a result, functions that are mapped onto $\bot$ in the resulting purity map are functions that were not applied during abstract interpretation.

$$\begin{aligned}(\hat{\varsigma},P) &\mapsto_{\text{purity}} (\hat{\varsigma}',P')\\ \text{where } (\hat{\varsigma} \xrightarrow{E} \hat{\varsigma}') &\in G_e\\ P'' &= P \bigsqcup \{[f \mapsto \mathsf{pure}] \mid\\ &\qquad (e,(f,\rho),d_{\text{arg}},\sigma) \in ctxs\}\\ P' &= P'' \bigsqcup \{handle(\hat{\varsigma},\mathit{eff}) \mid \mathit{eff} \in E\}\\ ctxs &= contexts(\hat{\kappa},\Xi)\\ (\ldots,\hat{\kappa},\Xi) &= \hat{\varsigma}\end{aligned}$$

Purity analysis is performed by taking the transitive closure of $\mapsto_{\text{purity}}$, starting from the bottom element of the analysis domain.

$$purityAnalysis(G_e) = \bigsqcup \{P \mid (\hat{\varsigma}_0,[]) \mapsto^*_{\text{purity}} P\}$$
$$\text{where } \hat{\varsigma}_0 \text{ is the initial state of } G_e$$

This purity analysis is finite if the underlying flow graph is finite, because the purity map monotonically increases in a finite domain.

```
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);
```

Fig. 3. Example program with recursive pattern.

IV. Freshness Analysis

A. Problem: Limited Precision for Addresses

The purity analysis from the previous section is attractive because it exactly expresses our definition of purity: if a function during application modifies an address that exists in the caller store, then that constitutes an observable side-effect which renders that function impure. But therein lies its greatest weakness as well: side-effects are classified solely based on their address. When the abstract machine from Section III is configured with a concrete allocator, addresses are generated with full precision. As a result, our purity analysis will determine function purity with maximal precision as well, without false positives (except for idempotent writes) or negatives. However, in a static analysis setting this is not realistic. In order to guarantee that an analysis runs in finite time and space, we sacrifice concrete precision primarily by *limiting the number of addresses* the abstract machine may choose from while it is evaluating the input program. Since our purity analysis hinges on addresses, we need to assess the impact of this precision loss in our approach.

As it turns out, the purity analysis 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, at one point or another the machine will run out of fresh addresses. Although increasing context-sensitivity may cause more applications to be considered pure by our analysis, only a single impure application renders the function impure, and increasing context-sensitivity only delays the inevitable.

Example: The program depicted in Figure 3 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 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 3, 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 19 and 12) are also considered to be a write to an address that exists in the caller store.

B. Solution: Freshness

Clearly we can do better by taking into account certain invariants that hold during concrete interpretation. For example, mutating the newly created object in a constructor call is never an observable side-effect. 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 check freshness on the level of variables (for variable effects) and object references (for property effects).

1) Variables: A variable is fresh with respect to a calling context if it is a local variable in that context. Because purity analysis needs to walk the stack of active function applications, there are actually three cases to consider: variables in the same scope (local variables), "outer" variables in an enclosing scope (visible or not), and all other variables. We therefore turn the check for freshness around by observing that only outer variables are not fresh, and writing to them causes observable side-effects. The handler for a variable write effect can then be defined without the need for checking the address of the written variable. Instead the handler only needs to rely on lexical scoping information offered by helper function $declNode : \mathsf{Var} \mapsto \mathsf{Var}$ that returns the variable declaration for a reference, and predicate $isOuter : \mathsf{Var} \times \mathsf{Fun}$, which returns whether a variable is declared in an enclosing scope of a given function scope or not.

$$\begin{aligned}
&handle(\hat{\varsigma}, \mathbf{Wv}(a, v)) = P \\
&\qquad \text{where } v_{\text{decl}} = declNode(v) \\
&\qquad\qquad P = \{[f \mapsto \mathsf{impure}] \mid isOuter(v_{\text{decl}}, f) \\
&\qquad\qquad\qquad \wedge\ (e, (f, \rho), d_{\text{arg}}, \sigma) \in ctxs\} \\
&\qquad\qquad ctxs = contexts(\hat{\kappa}, \Xi) \\
&\qquad\qquad (\ldots, \hat{\kappa}, \Xi) = \hat{\varsigma}
\end{aligned}$$

2) Object References: An object reference is fresh with respect to the top-level calling context if it points to an object that was created in that context. There are two operations that need to come together: we have sources at which fresh objects are created and/or bound to references, and we also have to propagate reference freshness.

Sources for fresh references are intuitive to find in our small language. Suppose that *fresh* is a set of variables, represented by their declaration, known to reference fresh objects. Then an object reference is fresh if its variable is in this set of fresh references. Additionally, the result of object construction through `new` is fresh, and a reference to the newly constructed object through `this` in a constructor is fresh. All other expressions are not fresh. The following rules define predicate $\mathit{isFresh} : \mathsf{Exp} \times \widehat{\mathit{Kont}} \times \mathcal{P}(\mathsf{Var})$ that captures our notion of freshness.

$$\begin{aligned}\mathit{isFresh}([\![v]\!], \hat{\kappa}, \mathit{fresh}) &= v_{\text{decl}} \in \mathit{fresh}\\ &\text{where } v_{\text{decl}} = \mathit{declNode}(v)\end{aligned}$$

$$\mathit{isFresh}([\![\mathtt{new}\ v\,\mathtt{(}s\mathtt{)}]\!], \hat{\kappa}, \mathit{fresh}) = \mathit{true}$$

$$\mathit{isFresh}([\![\mathtt{this}]\!], ([\![\mathtt{new}\ v\,\mathtt{(}s\mathtt{)}]\!], \ldots), \mathit{fresh}) = \mathit{true}$$

$$\mathit{isFresh}(e, \hat{\kappa}, \mathit{fresh}) = \mathit{false}$$

Propagation of fresh references happens through variable assignment only, making freshness analysis a limited intraprocedural analysis. We do not track freshness through for example function calls or property loading and storing: for these kinds of object flow we rely entirely on the underlying abstract interpretation and the addresses it allocates.

Like purity analysis, freshness analysis piggybacks on the underlying flow graph for control flow. We define it as a triple of the following components: a set of seen states (S), a current state ($\hat{\varsigma}$), and a mapping from calling contexts to a set of fresh references (F_κ).

For a variable assignment expression, freshness propagates from the right hand side to the reference on the left hand side. We use predicate *isFresh* to determine whether the right hand side is fresh. If this is the case, then we add the left hand side variable to the set of fresh references for the calling context at that state. Else, the variable is removed from this set.

$$\begin{aligned}(S, \overbrace{\mathbf{ev}([\![v\texttt{=}e]\!], \rho, \sigma, \hat{\kappa}, \Xi)}^{\hat{\varsigma}}, F_\kappa) &\mapsto_{\text{fresh}} (S', \hat{\varsigma}', F'_\kappa)\\ &\text{if } \hat{\varsigma} \notin S\\ \text{where } \mathit{fresh} &= F_\kappa(\hat{\kappa})\\ F'_\kappa &= F_\kappa[\hat{\kappa} \mapsto \mathit{fresh}']\\ \mathit{fresh}' &= \begin{cases}\mathit{fresh} \cup \{v_{\text{decl}}\}\\ \quad \text{if } \mathit{isFresh}(v_{\text{decl}}, \hat{\kappa}, \mathit{fresh})\\ \mathit{fresh} \setminus \{v_{\text{decl}}\} \text{ else}\end{cases}\\ v_{\text{decl}} &= \mathit{declNode}(v)\\ (\hat{\varsigma} \to \hat{\varsigma}') &\in G_e\\ S' &= S \cup \{\hat{\varsigma}\}\end{aligned}$$

In all other cases, F_κ is unchanged.

$$\begin{aligned}(S, \hat{\varsigma}, F_\kappa) &\mapsto_{\text{fresh}} (S', \hat{\varsigma}', F_\kappa)\\ &\text{if } \hat{\varsigma} \notin S\\ \text{where } (\hat{\varsigma} \to \hat{\varsigma}') &\in G_e\\ S' &= S \cup \{\hat{\varsigma}\}\end{aligned}$$

Even as an intraprocedural analysis, relation $\mapsto_{\text{fresh}}$ needs to keep track of fresh references per calling context. When a flow graph transition changes the calling context, then either we are entering a function with a calling context we did not previously encounter, or we exit a function restoring a previously encountered calling context. Because $\mapsto_{\text{fresh}}$ enforces the condition $\hat{\varsigma} \notin S$ on every transition, all other cases are ruled out. Also because of condition $\hat{\varsigma} \notin S$, the transitive closure of $\mapsto_{\text{fresh}}$ is finite if the underlying flow graph is too.

While having a mapping from calling contexts to fresh object references is required while traversing the flow graph, our purity analysis needs to determine freshness of object references per state. We therefore introduce mapping F_ς from states to their set of fresh references, by associating each state with the list of fresh references for the state's calling context obtained from F_κ as follows:

$$\begin{aligned}F_\varsigma = \{[\hat{\varsigma} \mapsto \mathit{fresh}] \mid\ & (\varnothing, \hat{\varsigma}_0, []) \mapsto^*_{\text{fresh}} (\_, \hat{\varsigma}, F_\kappa)\\ &\wedge (\ldots, \hat{\kappa}, \_) = \hat{\varsigma}\\ &\wedge \mathit{fresh} = F_\kappa(\hat{\kappa})\}\end{aligned}$$

We can now combine purity analysis with freshness analysis by defining handlers for property write effects that do not mark functions as impure when a fresh reference is involved. Property writes happen in two instances: explicit property store, or when assigning to a top-level variable:

$$\begin{aligned}\mathit{handle}(\overbrace{\mathbf{ko}(d, \mathbf{st}(s, v, \rho) : \iota, \hat{\kappa}, \Xi)}^{\hat{\varsigma}}, \mathbf{Wp}(a, v)) &= []\\ \text{if } \mathit{isFresh}(s, \hat{\kappa}, F_\varsigma(\hat{\varsigma}))\end{aligned}$$

$$\begin{aligned}\mathit{handle}(\overbrace{\mathbf{ko}(d, \mathbf{as}(v, \rho) : \iota, \hat{\kappa}, \Xi)}^{\hat{\varsigma}}, \mathbf{Wp}(a, v)) &= []\\ \text{if } \mathit{isFresh}(v, \hat{\kappa}, F_\varsigma(\hat{\varsigma}))\end{aligned}$$

Installing these handlers *before* the property write effect handlers from Section III ensures that freshness analysis for object references improves precision of the purity analysis.

V. Implementation

We implemented the purity analysis and freshness analysis discussed in this paper as a proof of concept[1]. Our implementation significantly extends the input language and semantics presented in this paper. Notably, our implementation adds support for computed properties. We also added many of the built-in JavaScript functions and objects required to run our benchmarks.

Our prototype implementation uses abstract garbage collection (AGC) as a technique to increase performance and precision of abstract interpretation [5]. Abstract garbage collection reclaims unused addresses, and so 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] https://github.com/jensnicolay/jipda/tree/scam2015/protopurity

			— Without fresh —		— With fresh —	
Benchmark	Flow time	#Func	#Pure	Purity time	#Pure	Purity time
`access-nbody`[2]	ε	11	2	ε	7	ε
`controlflow-recursive`[2]	ε	3	3	ε	3	ε
`crypto-sha1`[2]	ε	17	6	ε	7	ε
`math-spectral-norm`[2]	ε	5	2	ε	2	ε
`tree-add`[4]	0'01"	8	1	ε	4	ε
`navier-stokes`[3]	1'33"	36	4	0'20"	4	1'05"
`richards`[3]	0'19"	38	5	0'02"	9	0'05"

Fig. 4. Purity analysis results. *Flow time* is the running time of the flow analysis creating a flow graph. *#Func* is the number of functions in the benchmark. *#Pure* is the number of functions that our analysis determines pure. *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.

VI. Preliminary Experiments

We used our prototype implementation to analyze several JavaScript benchmarks using monovariant allocation. The table depicted in Figure 4 reports, for each benchmark, the number of pure functions detected, for both our purity analysis without freshness analysis, and for our purity analysis combined with freshness analysis. We also report the time the underlying flow analysis and the subsequent purity analysis took.

A. Observations

a) Our purity analysis is able to detect pure functions: We manually verified each function our analysis reported as pure, and found no false positives. This also confirms that our purity analysis is conservative. Absence of false positives is especially important in the context of security, where pure functions increase the confidence that there are no unintended side-effects when running sensitive code. `crypto-sha1` is a Sunspider benchmark[2] that tests cryptographic functions, and our prototype is able to detect that the majority of the functions that are actually called in that benchmark are indeed pure functions. `navier-stokes` is an octane benchmark[3] that passes around arrays between functions that update these arrays in place. Our analysis correctly predicted that almost all functions in this benchmark are impure.

b) The number of false negatives is low: In addition to running with standard abstract semantics using a monovariant allocation scheme, we also configured our flow analysis to execute programs with concrete semantics, i.e. with concrete values and concrete allocation. In this configuration, our definition of purity based on checking address membership in caller stores (Section III) is maximally precise. We found that, for all benchmarks except `navier-stokes` which ran out of memory (> 12GB) when executed with concrete semantics, our abstract purity analysis (with freshness) detected every pure function that the concrete semantics (without freshness) identified as pure.

[2]https://www.webkit.org/perf/sunspider/sunspider.html

[3]https://developers.google.com/octane

c) Freshness analysis improves precision, at the expense of performance: From the benchmark results we see that incorporating freshness analysis significantly improves precision. Freshness analysis does increase the overall running time of the analysis, but in our view adding it represents a good trade-off between speed and precision. `tree-add` is a JOlden benchmark[4] that we converted from Java into JavaScript. Although it is a relatively small benchmark, it exhibited poor precision when analyzed without freshness: only 1 pure function was detected out of 4 functions that are determined pure with freshness analysis enabled. The pattern in Figure 3, illustrating some of the weaknesses of address-only purity analysis, was distilled from this benchmark.

B. Comparison to Existing Work

Comparing our approach to existing approaches in terms of results is difficult, since to the best of our knowledge this is the first purity analysis that specifically targets JavaScript. There is related work, which we discuss in the next section, that focuses on method purity for Java, and analyzes benchmarks from the JOlden suite. `tree-add` is such a benchmark that we manually converted to JavaScript. While the comparison is not ideal, we observe that for this benchmark, according to [6] both JPPA [7] and JPure [8] find 1 pure method out of 10 methods in total. ReImInfer [6] finds 6 pure methods out of 10. In our JavaScript version there are 8 methods (functions), and our prototype implementation correctly infers that 4 of these are pure.

VII. Related work

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

Salcianu and Rinard [7] 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

[4]ftp://ftp.cs.umass.edu/pub/osl/benchmarks

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 is very lightweight compared to that of Salcianu and Rinard, and most of the work is performed by the underlying abstract interpreter computing interprocedural properties of a program. Like our approach, JPPA requires a whole-program analysis.

Pearce introduces JPure [8], 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 [9] present the pointer analysis used in Salcianu and Rinard [7] as an abstract interpretation. 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 [6] 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 calling contexts are more precise than any approach discussed in [6], but at the expense of running time and scalability. ReImInfer does not handle higher-order language constructs.

Rytz et. al. [10] 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 [11] treat purity in the setting of a functional language (Erlang), and they take the stricter definition of purity where a function also may not depend on external side-effects. Higher-order functions are supported, but the analysis is based on a "pretty simple" dataflow analysis, while in this work we employ a state-of-the-art flow analysis that offers full call/return precision.

Finifter et. al. [1] 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, but would not apply in our setting of JavaScript, which has closures.

Extending our approach to the stricter definition of purity of [1], [11] would involve tracking read effects in a manner similar to how we already track write effects, and handling them appropriately. This extension remains the focus of ongoing work.

VIII. Conclusion

We present 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 write effects when variables and object properties are written. By comparing addresses in effects with addresses in caller stores, we distinguish observable from unobservable side-effects. Doing this for all write effects in all applications of a function enables us to classify functions as pure or impure. 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 freshness analysis that can identify unobservable write effects that would be considered observable by the address-based purity analysis. Adding freshness analysis improves precision of our purity analysis considerably, as demonstrated in our preliminary experiments.

Acknowledgments

Jens Nicolay is funded by IWT Vlaanderen. Coen De Roover is funded by the *Cha-Q* project also sponsored by IWT Vlaanderen. Carlos Noguera is funded by the AIRCO project of the "Fonds Wetenschappelijk Onderzoek".

References

[1] 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*. ACM, 2008, pp. 161–174.

[2] J. I. Johnson and D. Van Horn, "Abstracting abstract control," in *Proceedings of the 10th ACM Symposium on Dynamic languages*. ACM, 2014, pp. 11–22.

[3] 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*. ACM, 1987, pp. 314–.

[4] 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.

[5] 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.

[6] 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*, vol. 47, no. 10. ACM, 2012, pp. 879–896.

[7] A. Salcianu and M. Rinard, "Purity and side effect analysis for java programs," *Lecture notes in computer science*, pp. 199–215, 2005.

[8] D. J. Pearce, "Jpure: a modular purity system for java," in *Compiler Construction*. Springer, 2011, pp. 104–123.

[9] R. Madhavan, G. Ramalingam, and K. Vaswani, "Purity analysis: An abstract interpretation formulation," in *Static Analysis*. Springer, 2011, pp. 7–24.

[10] 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*. ACM, 2013, p. 4.

[11] M. Pitidis and K. Sagonas, "Purity in erlang," in *Implementation and Application of Functional Languages*. Springer, 2011, pp. 137–152.