Determining Dynamic Coupling in JavaScript Using Object Type Inference

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*—Coupling in an object-oriented context is often defined in terms of access to instance variables and methods of other classes. JavaScript, however, lacks static type information and classes, and instead features a flexible object system with prototypal inheritance. In order to determine coupling in JavaScript, we infer object types based on abstract interpretation of a program. Type inference depends on both structure and behavior of objects, and common patterns for expressing classes and modules are supported. We approximate a set of accessed types per function, and classify every access as either local or foreign. Examples demonstrate that our object type inference, together with some additional heuristics concerning property access, enable determining coupling in JavaScript in a meaningful way.**

I. Introduction

JavaScript applications are becoming increasingly complex, and are no longer confined to the browser. As a result, various stakeholders would benefit from tools that support them in assessing the quality of JavaScript code. Extensive research has resulted in quality assessment tools for traditional, class-based, object-oriented languages [1], [2], [3]. JavaScript, in contrast, has received far less attention in this regard. One possible reason is that static reasoning over JavaScript programs is challenging due to the dynamic nature of the language. Static reasoning forms the foundation for advanced tooling.

One important aspect that determines the quality of a program is coupling. It is considered a good software engineering principle to keep coupling low, as this makes code easier to understand and maintain. In this paper, we describe our approach to determining the dynamic coupling of functions to object types. We define dynamic coupling for a function as the set of object types of which that function accesses properties. As JavaScript has no classes nor static types, we infer object types in such a way that we can measure coupling meaningfully, even in the presence of dynamic features and the commonly used class and module patterns. We demonstrate the usefulness of our approach by applying it to the detection of coupling-related code smells.

A. Approach

We start by approximating the set of all property accesses that might occur in a running program. To this end, an abstract interpreter collects tuples that capture the details of each property access (Section II). Next, we compute a type for the receiver (Section III), enabling us to determine the locality of each access based on the type of the base object and that of the `this` binding that is in effect. By collecting this information for all property accesses in a given function, we determine the dynamic coupling for that function (Section IV). The initially computed types are based only on information available in the tuples returned by the interpreter. When considering the entire set of property accesses, we are able to enhance these types by discovering relations between them based on their flow to base expressions (Section V-A) and their interfaces (Section V-B).

978-1-4673-5739-5/13/$31.00 © 2013 IEEE

B. Contributions

- We describe an abstract interpretation required for approximating all property accesses in a running program.
- We examine how to infer types for objects so that module and class patterns are supported.
- We show how to compute dynamic coupling in JavaScript based on approximations of property access and inferred object types.

II. Abstract Interpretation

At the heart of our approach to determining dynamic coupling is an abstract interpreter [4]. An abstract interpreter executes the program using abstract semantics that approximate the concrete semantics of JavaScript program execution. These abstractions simplify the semantic domain that models the runtime, and guarantees finite execution in both time and space. In this paper we define coupling in terms of types inferred from property access within functions. Therefore, our abstract interpreter is configured to collect a set of tuples that contain information about the property accesses it encounters. This set of tuples is then used to infer object types, which in turn determine coupling of functions.

In what follows, we detail the syntactic elements of JavaScript that are relevant to the abstract interpretation, the abstract domain used to analyse property accesses, and the manner in which property access information is collected during interpretation.

A. Syntactic elements

Figure 1 identifies the syntactic elements of JavaScript that are relevant for reasoning over property accesses with the

$n \in \mathsf{Node}$ = finite set of AST nodes

$f \in \mathsf{Fun} \subseteq \mathsf{Node}$ = function expressions and declarations

$v \in \mathsf{Id} \subseteq \mathsf{Node}$ = identifiers

$e \in \mathsf{Exp} \subseteq \mathsf{Node}$ = expressions

$p \in \mathsf{Prop} \subseteq \mathsf{Exp}$ = $e_b\,.\,v$ (dot) $\mid e_b\,[\,e_v\,]$ (bracket)

$t \in \mathsf{This} \subseteq \mathsf{Exp}$ = `this` (this expression)

$u \in \mathsf{Call} \subseteq \mathsf{Exp}$ = $e_f(e_{arg}*)$ (call expressions)

$c \in \mathsf{New} \subseteq \mathsf{Exp}$ = `new` $e_f(e_{arg}*)$ (new expressions)

$l \in \mathsf{ObjLit} \subseteq \mathsf{Exp}$ = $\{\langle v : e_{init}\rangle * \}$ (object literals)

$r \in \mathsf{ArrLit} \subseteq \mathsf{Exp}$ = $[\,e_{init} * \,]$ (array literals)

Fig. 1. Overview of relevant syntactic elements of JavaScript.

$Store = Addr \mapsto Val$

$Val = (Prim \times \wp(Addr)) + Benv$

$Benv = Str \mapsto \wp(Addr)$

$Addr$ = finite set of addresses

$Prim$ = finite set of abstract primitive values

$Prim \supseteq Str$ = set of abstract names

Fig. 2. The abstract domain used in our static analysis.

purpose of inferring object types and determining coupling. Besides property access itself, we also distinguish between function expressions and declarations, identifiers, call and `new` expressions, and array and object literals. The set Prop contains two kinds of property access nodes. Dot-based accesses identify the accessed property of base expression e_b through the identifier v, while bracket-based accesses identify the accessed property through the value of another indexation expression e_v. In the second case, JavaScript always coerces base expressions into object types and indexation expressions into strings. We define a function $base : \mathsf{Prop} \mapsto \mathsf{Exp}$ to access the base expression of either kind of property access.

B. Abstract domain

The abstract interpreter models the runtime of the target program with the abstract domain shown in Figure 2. Addresses in $Addr$ are references to values and generated whenever the interpreter allocates values. An abstract object in $Benv$ is a one-to-many mapping from names to addresses. Abstract objects are used to represent objects, arrays, closures, and environments. A store in $Store$ maps addresses onto a value. It mimics the heap and is a key component in performing points-to analysis. Values are either first-class values (primitives and object references) or objects.

In the context of object type inference and dynamic coupling, we are especially interested in those AST nodes that create objects. New expressions, array literals, and object literals are sources of object creation that can be directly traced back to specific nodes. Certain method invocations on built-in objects, like for example `Object.create`, also create objects behind the scenes. Function expressions and declarations give rise to function objects, the evaluation of regular expressions results in `RegExp` objects, and so on. Function nodes also indirectly create a prototype object, accessible through the public `prototype` property on the function object created by the same node. Addresses are generated in the interpreter in such a way that there is a link between the address and the node responsible for creating that address, even if the address in question references an implicitly created object. Thus, we define a function $node : Addr \mapsto \mathsf{Node}$ that establishes this between an address and its corresponding node. Note that this does not limit the number of addresses linked to a particular node, so addresses can be generated in a context-sensitive manner for example.

$Acc = Addr \times Addr \times \mathsf{Prop} \times Addr \times Str \times Store$

$a_f \in Addr$	Address of the function enclosing the property access.
$a_t \in Addr$	Current address of `this` in the active execution context.
$p \in \mathsf{Prop}$	Property access node.
$a_b \in Addr$	Address of $base(p)$.
$s \in Str$	Name of the referenced property in p, either name of v or the result of evaluating e_v.
$\sigma \in Store$	Current store.

Fig. 3. Elements of the active execution context in Acc, generated for each property access the interpreter evaluates.

C. Collecting property access information

Examining property access in a JavaScript program on a syntactic level only is not sufficient: it usually does not reveal any useful information about the objects and properties that might be involved at runtime. We therefore rely on an abstract interpreter that is instrumented to collect information on all property accesses it evaluates during abstract interpretation of a program. For the moment we will only consider *read* access. In Section V-C we explain how to extend our approach to also include *write* access, and how to deal with accessing undefined properties. Information regarding a particular read property access is recorded in a tuple $(a_f, a_t, p, a_b, s, \sigma) \in Acc$ consisting of the elements detailed in Figure 3. The evaluation of one property access can result in multiple tuples, depending on the number of addresses in the points-to sets of the enclosing function, the value of `this`, and the base expression. How close these and other computed sets in the remainder of this paper are to their actual runtime equivalents, depends on the combination of the dynamicity of the program and the precision of the underlying abstract interpretation.

III. Object Type Inference

Our approach determines coupling dynamically by considering the types of accessed objects. To compute the type of an

object, we define a function iof that takes an object reference and a store, and returns a set of addresses that represents the type for that object reference.

$$iof : Addr \mapsto Store \mapsto \wp(Addr)$$

Using addresses to represent types entails that the type of an object is another object. Below we discuss which objects we consider as types. Using objects as types carries the added benefit of allowing us to map an object to abstract syntax tree locations (cf. II-B), which can give further information to type inference algorithms. Defining the function iof this way gives rise to a type hierarchy in the form of a tree of objects computed by the transitive closure $iof*$.

$$iof* : Addr \mapsto Store \mapsto \wp(\wp(Addr))$$
$$iof*(a_0, \sigma) = \{\{a_0, \ldots, a_n\} \mid \forall i \in 1..n : a_i \in iof(a_{i-1})\}$$

Function iof encapsulates different possible object type inference algorithms. If we only look at a single tuple at a time to infer types, there are two straightforward definitions of this function.

1) *Prototype-based*, where the type of an object is its prototype.
2) *Object-based*, where each object is its own type.

Below we explore these definitions, and conclude that we need to combine them in order to support some of the most common JavaScript patterns.

A. Prototype-based type inference

Every object in JavaScript has an *internal* [[Prototype]] property that points to that object's prototype, or that has the value `null` if there is no prototype. Similarly, function objects have a *regular* `prototype` property. When a function object is invoked as a constructor using the `new` operator, the result is a newly created object of which the [[Prototype]] property is set to the `prototype` property of the constructor at the time of construction. Therefore, for any function `F`, the following holds:

```
Object.getPrototypeOf(new F()) === F.prototype
```

We consider a first approximation to object type inference in which the iof function is defined in terms of the [[Prototype]] property. In abstract terms the value of [[Prototype]] for an object is a (possibly empty) *set* representing an over-approximation of object references to prototype objects. By defining a function $proto : Benv \mapsto \wp(Addr)$ that returns this set of possible prototype addresses, we can define a prototype-based definition of iof_p as follows:

$$iof_p : Addr \mapsto Store \mapsto \wp(Addr)$$
$$iof_p(a, \sigma) = proto(\sigma(a))$$

A prototype chain of a JavaScript object starts at the [[Prototype]] of that object and then traces out subsequent [[Prototype]] links. In JavaScript the prototype chain of an object would be obtained by repeatedly applying the built-in function `Object.getPrototypeOf`, until `null` is encountered. The `instanceof` operator of JavaScript uses the prototype chain to determine types: an object `x` is an instance of constructor `F` if the prototype chain of `x` contains `F.prototype`. For approximating the object type of an object, the transitive closure of iof_p*, will compute a prototype *tree*, represented as a set of all possible [[Prototype]] chains. Function iof_p defines types by referring to prototype objects (`Object.getPrototypeOf`) instead of taking prototype objects to constructors as `instanceof` does.

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

function Point(x, y) {
  Circle.call(this, x, y, 0);
}
Point.prototype =
  Object.create(Circle.prototype);

var p = new Point(90, 90);
p.x
```

Fig. 4. Example program P_1.

Convention: In the code examples that follow we will translate addresses into equivalent source code expressions, i.e. expressions that evaluate to the same addresses. If necessary, the evaluation context for these expressions will be made clear in the example.

Example 1: In the JavaScript program P_1 (Fig. 4) the following holds:

```
Object.getPrototypeOf(p)
  === Point.prototype
⇒ true
Object.getPrototypeOf(Point.prototype)
  === Circle.prototype
⇒ true
Object.getPrototypeOf(Circle.prototype)
  === Object.prototype
⇒ true
Object.getPrototypeOf(Object.prototype)
  === null
⇒ true
```

Therefore, if σ is the store when evaluating property access `p.x`, then $iof_p(\mathtt{p}, \sigma)$ = {`Point.prototype`} and $iof_p*(\mathtt{p}, \sigma)$ = {{`Point.prototype`, `Circle.prototype`, `Object.prototype`}}. □

Example 2: Example program P_2 (Fig. 5) shows the essence of the module pattern in JavaScript. Functions `pub` and `priv` are defined in the local scope of the immediately invoked function expression (IIFE) enclosing the two functions. Because they are confined to the local scope of the IIFE, they are not directly accessible from outside it. The object literal the IIFE returns serves as the interface of the module, so that the access `module.pub` returns function `pub`. Using iof_p, the type computed for the base object in property access `module.pub` is {`Object.prototype`}. The reason is that object literals are evaluated to objects that are created as if by the expression `new Object()`. The type

```
var module = (function () {
    function pub(x) {return Math.sqrt(x)};
    function priv() {return "bar"};
    return {pub:pub};
  })();

function f() {
  print(module.pub(4));
}
f()
```

Fig. 5. Example program P_2.

```
for (var i = 0; i < 1000; i++) {
  var o = {x:i};
  print(o.x);
}
```

Fig. 6. Example program P_3.

of base object `Math` in property access `Math.sqrt` is also {`Object.prototype`}, because the prototype of this object is `Object.prototype` as well. □
Example 2 shows the limits of iof_p when inferring the types of object literals and built-in objects such as `Math`.

B. Object-based type inference

The finest granularity at which we can assign object types, is by assigning each object its own type. Therefore, we define an object-based version of iof_o:

$$iof_o : Addr \mapsto Store \mapsto \wp(Addr)$$
$$iof_o(a, \_) = \{a\}$$

The type hierarchy computed by the transitive closure iof_o* consists of a single chain containing the object address: $iof_o * (a, \_) = \{\{a\}\}$. In a concrete setting, each object can be allocated at a unique address. In our abstract setting, the set of addresses is finite and typically much smaller than the set of concrete addresses. The address allocation strategy used during abstract interpretation will greatly influence to which degree we can distinguish different objects. Although different configurations and implementation strategies may therefore impact the speed and precision (and therefore quality) of computed results, those results should always be sound.

Example 3: Taking the same module definition as in Example 2, but now using object-based type inference, we compute {`module`} as type of the base object in property access `module.pub`, and {`Math`} as type of the base object in property access `Math.sqrt`.

Example 4:

Consider the example program P_3 (Fig. 6) consisting of a `for` loop: A concrete interpretation will allocate 1000 different objects, but abstract interpretation might distinguish between far less *abstract* objects, usually depending on how quickly it reaches a fixpoint (which in turn might depend on things like context-sensitivity, strong updating, widening, etc.). For example, our abstract interpreter is equipped with a "k last call sites" strategy for distinguishing between different contexts (Section VII), and entering a loop body is considered to be equivalent to a function call in this regard. With $k = 0$, i.e with context-sensitivity turned off, only one address is used for allocating objects resulting from evaluating the object literal, and the property access is evaluated three times before a fixpoint is reached. We therefore have $iof_o(\texttt{o}, \_) = \{\{\texttt{x:i}\}\}$ for all property accesses.

When $k = 5$, the abstract interpreter returns seven tuples and generates three distinct addresses for allocating the object literal. In this case we compute three different types for `o` instead of one single type as was the case with $k = 0$.

In both cases the computed types are sound, altough in the latter case they are more precise. □

C. Combining prototypes and object-based inference

Examining the results obtained by typing base objects using either iof_p or iof_o, we observe the following:

- Example 1 shows that prototype-based object type inference works well when objects are explicitly created through constructors using `new`, and we are dealing with "instance" data. However, when accessing properties on non-instance objects, like the property access `Math.sqrt` in Example 2, prototype-based inference is not useful. Example 2 also demonstrates that prototype-based inference is unable to distinguish between accesses to different modules. Objects created from object literals are considered to be instances of the standard `Object` constructor, greatly diminishing the usefulness of the type inference in these cases.
- Examples 3 and 4 shows that object-based object type inference is able to distinguish between different object literals, and, as a consequence, different modules. On the other hand, as in Example 4, object-based inference may be overkill in situations where objects from one or more object literals intuitively represent different instances of the same type.

A better definition for iof therefore would be one that differentiates objects based on their creation. If an object comes into existence as a result of evaluating a constructor invocation (New), an array literal (ArrLit), or the invocation of a built-in method like for example `Object.create` (Call), then the type of that object is its [[Prototype]]. For all other objects, we take the address of those objects as their type. Built-in objects (like `Math` and the global object) will have unique addresses, as will objects created at different source code locations[1].

$$iof_n : Addr \mapsto Store \mapsto \wp(Addr)$$
$$iof_n(a, \sigma) = \begin{cases} iof_p(a, \sigma) & \text{if } n = node(a) \\ & \wedge n \in \mathsf{New} \cup \mathsf{ArrLit} \cup \mathsf{Call} \\ iof_o(a, \sigma) & \text{else} \end{cases}$$

[1]Up to a certain limit because the set of addresses is finite.

IV. Computing coupling

As we have previously explained (cf. I), we are interested in statically assessing the dynamic coupling of JavaScript functions. Thus, we require, for each property access appearing in a given function, the type of the base object (represented by its address $Addr$), and a constant indicating whether that type is local or foreign ($\{L, F\}$) at the time of access. The function *coupling* computes typed accesses specific to a function, starting from the set of tuples collected by the abstract interpreter.

$$coupling : Addr \mapsto \wp(Acc) \mapsto (Addr \times \{L, F\})$$

Whether a type is local or foreign is decided in function of the value of `this` in the function at the time of access. If the property is accessed on a type in the type hierarchy of `this`, then the type is marked as local (L), otherwise, the type is deemed foreign (F).

Our approach can be instantiated with any definition of the function iof that returns a set of addresses representing objects. In what follows, we will use ${iof}_n$ and its transitive closure from the previous section. Concretely, when the function ${iof}_n*$ is applied to an address a_t representing `this`, we obtain all possible [[Prototype]] chains consisting of types that are considered local to the active function invocation. Then, the function ${iof}_n$ is used to compute a type a_o for each object appearing as a base object in a property access. The base object type is local when it is a member of a chain returned by the application of ${iof}_n*$, else it is foreign.

$$locality : Addr \mapsto Addr \mapsto Store \mapsto \wp(\{L, F\})$$

$$locality(a_o, a_t, \sigma) = \bigcup_{\overrightarrow{a} \in {iof}_n*(a_t,\sigma)} \begin{cases} L & \text{if } a_o \in \overrightarrow{a} \\ F & \text{else} \end{cases}$$

Function *locality* ensures that every access to `this` in functions is always considered to be a local access, since in that case $a_o = a_t$.

We then define a helper function that will filter the information on property access collected in a set $\mathcal{T}$, keeping only tuples pertaining to the specified function.

$$filterFun : \mathsf{Fun} \mapsto \wp(Acc) \mapsto \wp(Acc)$$
$$filterFun(f, \mathcal{T}) = \{(a_f, \_, \_, \_, \_, \_) \in Acc \mid node(a_f) = f\}$$

Filtering the collected coupling information, and applying *iof* and *locality*, we can determine the dynamic coupling for a function f as follows:

$$coupling(f, \mathcal{T}) = \{(a, \ell) \mid a \in {iof}_n(a_b, \sigma) \wedge \ell \in locality(a, a_t, \sigma) \wedge (a_f, a_t, p, a_b, s, \sigma) \in filterFun(f, \mathcal{T})\}$$

Example 5: The code in Figure 7 defines two types, `Circle` and `Point`. In the module, a `Point` is made to be a `Circle` with radius 0.

Suppose we have the following client application:

```
function Circle(x, y, r) { ... }

Circle.MP = 0.5;

Circle.prototype.midpoint =
  function (c) {
    return new Point((this.x+c.x) * Circle.MP,
      (this.y+c.y) * Circle.MP);
  }

function Point(x, y) {
  Circle.call(this, x, y, 0);
}
Point.prototype = Object.create(Circle.prototype);
```

Fig. 7. Example program P_4.

```
var c = new Circle(10, 20, 50);
var p = new Point(90, 90);
p.midpoint(c)
```

The abstract interpreter returns the following tuples regarding property access in method `midpoint`:

(`midpoint`,`p`,`this.x`,`p`,`"x"`,σ)
(`midpoint`,`p`,`c.x`,`c`,`"x"`,σ)
(`midpoint`,`p`,`Circle.MP`,`Circle`,`"MP"`,σ)
(`midpoint`,`p`,`this.y`,`p`,`"y"`,σ)
(`midpoint`,`p`,`c.y`,`p`,`"y"`,σ)

We compute the following coupling for `midpoint` using ${iof}_n$:

(`Point.prototype`, L)
(`Circle.prototype`, L)
(`Circle`, F)

The coupling to `Point.Prototype` is generated by accesses on `this` bound to an instance of `Point`, and accesses of `this` are local by definition. Access to parameter `c` inside method `midpoint` generates coupling to `Circle`, which is also local since `Circle.prototype` is a member of all chains in the type hierarchy ${iof}_n$ * (`p`). The access `Circle.MP` is considered to be a foreign coupling to `Circle`, since ${iof}_n$(`c`) = `Circle`, which is not a member of the type hierarchy of `this`. The constructor invocation `new Point(...)` does not generate any coupling, since `Point` is looked up in the lexical scope of function `midpoint`, and we do not track coupling to environment records[2]. □

Example 6: We again start with the code in Figure 7, but now define the following client program:

```
var c2 = {x:10, y:20, r:50};
var p = new Point(90, 90);
p.midpoint(cl);
```

The difference between this client program and the one in Example 5 is that we don't create a circle object using the constructor `Circle`, but instead use an object literal that

[2]We are deliberately glossing over some details here, like the top-level environment being the global object in JavaScript, and the `with` statement blurring the line between objects and environment records, but in our approach *only* member access (at the syntactical level) generates dynamic coupling.

defines the same properties. The abstract interpreter returns the same tuples regarding property access, but now we compute the following coupling for `midpoint` using iof_n:

$$(\texttt{Point.prototype}, L)$$
$$(\texttt{c2}, F)$$
$$(\texttt{Circle}, F)$$

Access to parameter `c` inside method `midpoint` generates coupling to type `c2`, which is foreign since it is not a member of the type hierarchy $iof_n * (\texttt{p})$. □

V. Enhancing Object Types

Having explained how we calculate the types to which a JavaScript function is coupled to, we now explore how to enhance the typing information produced by our object type inferencing for the purpose of coupling assessment. Previous definitions of iof (cf. III) are derived from looking at a single tuple generated by the abstract interpretation of the program, while in what follows, we consider the whole set of tuples. We propose two ways of relating equivalent object types, the first one relates object types by looking at what instances are used together (i.e., flow to the same base objects in a property access); while the second one relates object types by considering which property names are accessed together (i.e., interface types). Finally, as type inference is calculated using property reads alone, we discuss what further information can be obtained by including property write operations in the analysis.

A. Flow of types

The definition of iof_n solves some of the problems we observed at the beginning of Section III-C, but it still does not adequately deal with the situation in which objects resulting from the evaluation of one or more object literals intuitively represent different instances of the same type. One way of relating types is by looking at where instances end up being used as base object. To capture this notion, we first define the function $flowsTo$ that for every type computed using iof_n, yields all the base expression nodes that type flows to. Unlike the previous type inferences, we now take the entire set of tuples returned by the abstract interpreter into consideration.

$$flowsTo : Addr \mapsto \wp(Acc) \mapsto \wp(\mathsf{Exp})$$
$$flowsTo(a, \mathcal{T}) = \{e_b \mid a \in iof_n(a_b, \sigma) \wedge (\_, \_, p, a_b, \_, \sigma) \in \mathcal{T} \wedge base(p) = e_b\}$$

Then we can define a reflexive and symmetric relation $\approx_f$ stating that "two types are equivalent iff they flow to the same base expression node".

$$a_1 \approx_f a_2 \iff flowsTo(a_1, \mathcal{T}) \cap flowsTo(a_2, \mathcal{T}) \neq \varnothing$$

The transitive closure of $\approx_f^*$ partitions the set of types in the range of iof_n into equivalence classes. All types returned by a single application of iof_n are equivalent by this definition.

Example 7: We compute the following coupling for function `area` in example program P_5 (Fig. 8).

```
function Circle(x, y, r) { ... }

Circle.prototype.circumference =
  function () {
    return 2 * Math_a.PI * this_a.r;
  }

function area(c) {
  return Math_b.PI * c_a.r * c_b.r;
}

var c1 = {x:10, y:20, r:30};
var c2 = new Circle(30, -5, 10);
var c3 = new Circle(50, 50, 20);
area(c1)
area(c2)
c2.circumference()
area(c3)
```

Fig. 8. Example program P_5.

$$(\texttt{Math}, F)$$
$$(\texttt{c1}, F)$$
$$(\texttt{Circle.prototype}, F)$$

And for method `circumference` we obtain the following coupling:

$$(\texttt{Math}, F)$$
$$(\texttt{Circle.prototype}, L)$$

For the set of tuples $\mathcal{T}$, we have the following flow of types:

$$flowsTo(\texttt{Math}, \mathcal{T}) = \{\texttt{Math}_a, \texttt{Math}_b\}$$
$$flowsTo(\texttt{c1}, \mathcal{T}) = \{\texttt{c}_a, \texttt{c}_b\}$$
$$flowsTo(\texttt{Circle.prototype}, \mathcal{T}) = \{\texttt{c}_a, \texttt{c}_b, \texttt{this}_a\}$$

Consequently, the equivalence relation $\approx_f^*$ partitions the set of computed types as follows:

$$\{\{\texttt{Math}\}, \{\texttt{c1}, \texttt{Circle.prototype}\}\}$$

This means that types `c1` and `Circle.prototype` are equivalent under $\approx_f^*$, and we could substitute the equivalence class for the computed types when determining coupling. If we denote the equivalence class of type a as $[a]$, we have:

$$[\texttt{Math}] = \{\texttt{Math}\}$$
$$[\texttt{c1}] = [\texttt{Circle.prototype}] = \{\texttt{c1}, \texttt{Circle.prototype}\}$$

This however can have consequences for the locality of the access. The equivalence class of prototype `Circle.prototype` in our example also contains an object-based type `c1`, the latter which does not belong to the hierarchy of receiver `c2`. If the type hierarchy of an equivalence class is the join of the type hierarchies computed for every type in that equivalence class, the locality for the coupling to `Circle.prototype` both $\{L, F\}$. This also breaks the invariant that an access to `this` inside a method is a purely local access. One solution could be to replace types by their equivalence class, but keep the locality computed using the original type. Also, if an equivalence class contains a prototype, one could assume that "the programmer knows

best" and that these types, most often constructed by `new`, should somehow take precedence over other types in that equivalence class. We regard clarifying these issues as future work.

Relating types by their flow may make types too coarse-grained. Two types may flow to the same base expression(s), thereby sharing a very small part of their interface, but actually may be unrelated for all other intents, including in the context of coupling. We'll have more to say on this when we discuss interface types next.

B. Interface types

Just like *flowsTo* induces a partitioning on the types computed by iof_n based on the flow of types, we can also partition the set of types based on the properties that are accessed. Intuitively, this follows the notion of duck typing. We define the *interface* of a type as the set of names that are accessed on it. As with *flowsTo*, we consider the entire set of property access information returned by abstract interpretation.

$$\begin{aligned} interface &: Addr \mapsto \wp(Acc) \mapsto \wp(Str) \\ interface(a, \mathcal{T}) &= \{s \mid a \in iof_n(a_b, \sigma) \\ &\qquad \wedge (\_, \_, \_, a_b, s, \sigma) \in \mathcal{T}\} \end{aligned}$$

We define a reflexive and symmetric relation $\approx_i$ stating that "two types are equivalent iff they have the same interface".

$$a_1 \approx_i a_2 \iff interface(a_1, \mathcal{T}) = interface(a_2, \mathcal{T})$$

The transitive closure of $\approx_i^*$ partitions the set of types in the range of iof_n into equivalence classes, and here too all types returned by a single application of iof_n are equivalent by this definition.

Relating types only by interface is usually not sufficient, since it may relate types that should be considered unrelated. Two types can have the same interface, say {`"x"`, `"y"`, `"z"`}, but one type might express a 3D coordinate, while the other represents the code for a 3-cipher lock. As was previously the case with the definition of iof_n, where we combined prototype inference with object-based type inference, here too a combination of flow information and interfaces may be the best approach.

Example 8: In program P_6 (Fig. 9) the relation $\approx_f^*$ induces the following partition on the set of types:

{{`Math`}, {`Label.prototype`, `Circle.prototype`}}

Types `Label.prototype` and `Circle.prototype` are related because they flow to function `hash`. However, intuitively they represent different types and have different interfaces, even though they share {`"x"`, `"y"`} in this respect. Based on interfaces, we may split the equivalence class based on flow, thereby obtaining a set of unrelated types again:

{{`Math`}, {`Label.prototype`}, {`Circle.prototype`}}

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

function Circle(x, y, r) { ... }

function hash(p) {
  return ((p.x + p.y)*(p.x + p.y + 1)/2) + p.y;
}

Circle.prototype.circumference =
  function () {
    return 2 * Math.PI * this.r;
  }

function display(label) {
  return label.x + label.y + label.n;
}

var c = new Circle(30, -5, 10);
var l =  {x: 50, y:50, n:"Destination"};
hash(c);
c.circumference();
hash(l);
display(l);
```

Fig. 9. Example program P_6.

C. Writing properties

So far we have only addressed coupling as a consequence of reading properties. When considering writing properties' impact on coupling, we observe that:

1) Writing properties can extend or modify an object. When the property is not present, the property is added, else the value of the property is changed.
2) Writing of properties does not involve prototype lookup. The base object is the object that is extended or modified.

In order to properly reason about property writes, the abstract interpreter must be extended to keep track of extra information for each property access. We therefore extend the definition of our tuples in Acc by adding a tag indicating whether a property write (W) or read (R) occurred.

$$Acc_{RW} = Addr \times Addr \times \mathsf{Prop} \times \{R, W\} \times Addr \times Str \times Store$$

If the abstract interpreter takes a snapshot of the store before the property access itself is effectuated, it becomes possible to determine whether the accessed property exists or not. That way we can discern between the following scenarios with regards to the coupling induced by the property access:

1) Read of an existing property: this is the scenario that we have considered so far in the paper.
2) Read of a non-existing property, or property with an `undefined` value: programmers sometimes use the presence or absence of a certain property as a type test. It may be argued that reading an undefined property does not generate coupling by itself. Programmers might try to read undefined properties as a kind of "instance of"

check, in which case the check is normally followed by uses of the object, which in turn will be detected as coupling.

3) Update of an existing property: this scenario generates coupling in the same way as reading an existing property, thus it is detected in the same manner as scenario 1 above.
4) Adding a property: again we argue that this does not generate coupling on the level of types, since adding properties always "works", regardless of the inferred type of the receiving object.

VI. Applications

We now present how, using the coupling detection approach we have outlined above we can detect the Feature Envy bad smell as defined by Lanza, Marinescu and Ducasse in [5], adapted to functions in JavaScript.

A. Detecting Feature Envy

Feature Envy is a bad smell related to coupling. Informally, it can be described as "a design disharmony [that] refers to methods that seem more interested in the data of other classes than that of their own class" [5]. To detect feature envious functions, we implement the three metrics that Lanza et. al. use to detect Feature Envy: Access To Foreign Data ($atfd$), Locality of Attribute Access (laa), and Foreign Data Providers (fdp). Given those functions, they propose the following detection strategy for Feature Envy, given a function f:

$$\begin{aligned} featureEnvy(f) \iff & \ atfd(f) > FEW \\ & \wedge\ laa(f) < \frac{1}{3} \\ & \wedge\ fdp(f) \leq FEW \end{aligned}$$

where FEW stands for a small positive integer like 2 or 3.

We use the set of tuples computed by function $coupling$ from Section IV to distinguish between different data providers, and between foreign and local attributes, Note however, that a type is returned only once by the $coupling$ function regardless the amount of times that type is referred to in the function. When detecting feature envy, this distinction becomes relevant. In order to approximate the different accesses to the same type, we include the name of the accessed property s in the coupling information computed by $coupling_s$:

$$\begin{aligned} coupling_s &: Addr \mapsto \wp(Acc) \mapsto (Addr \times Str \times \{L, F\}) \\ coupling_s(f, \mathcal{T}) &= \{(a, s, \ell) \\ &\quad \mid \tau \in iof(a_b, \sigma) \wedge \ell \in locality(a, a_t, \sigma) \\ &\quad \wedge\ (a_f, a_t, p, a_b, s, \sigma) \in filterFun(f, \mathcal{T})\} \end{aligned}$$

B. Calculating the metrics

It is straightforward to compute the proposed coupling metrics using the results from function $coupling_s$. In what follows, $\#S$ denotes cardinality of S.

Function atd counts all distinct property accesses contained in function f, given a set of tuples $\mathcal{T}$.

$$\begin{aligned} atd &: \mathsf{Fun} \mapsto \wp(Acc) \mapsto \mathsf{Int} \\ atd(f, \mathcal{T}) &= \#coupling_s(f, \mathcal{T}) \end{aligned}$$

Function $atfd$ only counts the foreign property accesses.

$$\begin{aligned} atfd &: \mathsf{Fun} \mapsto \wp(Acc) \mapsto \mathsf{Int} \\ atfd(f, \mathcal{T}) &= \#\{(\_, \_, F) \in coupling_s(f, \mathcal{T})\} \end{aligned}$$

Function laa is the ratio of local property accesses to total number of property accesses.

$$\begin{aligned} laa &: \mathsf{Fun} \mapsto \wp(Acc) \mapsto \mathsf{Int} \\ laa(f, \mathcal{T}) &= \frac{atd(f, \mathcal{T}) - atfd(f, \mathcal{T})}{atd(f, \mathcal{T})} \end{aligned}$$

Finally, the function fdp counts the number of distinct foreign types that are accessed.

$$\begin{aligned} fdp &: \mathsf{Fun} \mapsto \wp(Acc) \mapsto \mathsf{Int} \\ fdp(f, \mathcal{T}) &= \#\{a \mid (a, \_, F) \in coupling_s(f, \mathcal{T})\} \end{aligned}$$

To illustrate how feature envy can be detected, consider the following example.

Example 9: Suppose we have the following client program (identical to the one in Example 5) that uses the code in Figure 7.

```
var c = new Circle(10, 20, 50);
var p = new Point(90, 90);
p.midpoint(c);
```

Using function $coupling_s$ and a set of property access tuples $\mathcal{T}$, we compute the following dynamic coupling for method `midpoint`:

$$\begin{aligned} &(\texttt{Point.prototype}, \texttt{"x"}, L) \\ &(\texttt{Point.prototype}, \texttt{"y"}, L) \\ &(\texttt{Circle.prototype}, \texttt{"x"}, L) \\ &(\texttt{Circle.prototype}, \texttt{"y"}, L) \\ &(\texttt{Circle}, \texttt{"MP"}, F) \end{aligned}$$

The set of foreign data providers is $\{\texttt{Circle}\}$ being accessed once. The remaining types are local, with in total 4 distinct data accesses. The metrics would then be calculates as:

$$\begin{aligned} atfd(\texttt{midpoint}, \mathcal{T}) &= 1 \\ laa(\texttt{midpoint}, \mathcal{T}) &= \frac{4}{5} \\ fdp(\texttt{midpoint}, \mathcal{T}) &= 1 \end{aligned}$$

From this we conclude that method `midpoint` does not exhibit Feature Envy.

C. Dealing with arrays

As explained in Section VI-A, function $coupling_s$ takes the name of the accessed properties into account in its output. JavaScript, unlike Java for example, treats indexing into an array as any other property access. A property name s is an array index if it is the string representation of an integer between 0 and $2^{32} - 1$ [6]. This can have undesirable consequences,

since functions accessing arrays can have very strong coupling to array objects if every array access is counted as a distinct property access.

To address this, we could choose to disregard array index access altogether or treat all array indexing for a particular base array object as one distinct access. Instead, we define a predicate function idx that discriminates between property names that are indexes and those that are not. If the name of the access is computed, loss of precision in the abstract domain can lead to a situation where the concrete representation of an abstract name can be both an array index name and not. Instead of reflecting this by returning a set of booleans, we simplify and idx returns `false` in this situation.

$$idx : Str \mapsto \mathsf{Bool}$$

The behavior of generic JavaScript methods on `Array.prototype` assume that any object accessed using array indexes is in fact array-like. We therefore do not check in any way whether the base object actually is of type `Array.prototype`. This has as an advantage that we can identify array index access on the level of tuples in Acc returned by the abstract interpreter. We can then remove these accesses, or we can define a function $mapIdx$ that maps the property name of every index access onto a unique global symbol IDX so that array indexing is reflected in the metrics at most once for every base object.

$$\begin{aligned} mapIdx &: \wp(Acc) \mapsto \wp(Acc) \\ mapIdx(\mathcal{T}) &= \{(a_f, a_t, p, a_b, s', \sigma) \\ &\quad \mid (a_f, a_t, p, a_b, s, \sigma) \in \mathcal{T} \\ &\quad \wedge s' = \begin{cases} IDX & \text{if } idx(s) \\ s & \text{else} \end{cases} \end{aligned}$$

VII. Implementation

We implemented our approach on top of a generic abstract interpreter called JIPDA, which was influenced by work of Might et al. [7]. JIPDA is capable of handling a large subset of JavaScript semantics, and allows for a large range of configuration options, allowing us to plug-in the abstract domain depicted in Fig. 2 which generates addresses as described in Section II-B. To generate and collect the tuples central to our approach, we instrumented the evaluator by overriding the two methods that deal with member expressions: one method for member expressions that are in operator position, and the other method for all other cases. To determine the enclosing function, we implemented a function that walks the stack to find the most recent invocation.

Our current implementation computes the functions that calculate coupling ($coupling$ and $coupling_s$), and the $flowsTo$ and $interface$ relations defined in Section V. We consider the development of the algorithms that enhance the type inference based on these partitions the subject of future work.

The generic abstract interpreter, its configuration, and the code for implementing our approach, are written in JavaScript. The implementation is publicly available on the project's webpage (`https://code.google.com/p/jipda/`); the configuration of the abstract interpreter and the supporting code used in this paper can be found in folder `instances/mod`.

Limitations: Most limitations in the implementation of our approach are a direct consequence of limitations in the underlying abstract interpreter.

Some of these limitations are inherent to abstract interpretation itself. By using overapproximation, we are sure that we do not exclude any objects and relations between objects that may exist at runtime, but false positives (computed results that will never actually occur in a running program) will diminish the usefulness of the type inference. Therefore, the abstract interpreter that underlies an implementation of our approach should be precise enough to be useful in practice, while being able to compute results in an acceptable amount of space and time, where "useful" and "acceptable" depend on the context. Such a good trade-off between speed and precision is difficult to achieve.

Our implementation of the abstract interpreter supports a set of ECMAScript 5 features [6] large enough, and models this with sufficient precision, to be able to experiment with interesting examples. However, we are currently not able to handle real-world JavaScript frameworks and applications. The main shortcoming is that many global objects and primitives are not yet modeled in the abstract domain. Another limitation is that we do not yet support getter/setter properties and strict mode. These limitations must be addressed before we can attempt to validate the usefulness of our approach on real-life JavaScript programs.

VIII. Related Work

A. Static and dynamic coupling metrics

We have found no previous work that concerns itself with coupling assessment of JavaScript functions, as most existing work has been carried out in the context of object-oriented systems with a static type-system. Within this body of work, two main approaches have been explored: those that rely on the structure of the *source code* (static) and those that rely on (dynamic) *runtime* information, of which the former have received most of the attention in literature. Most of them measure the manner in which classes are linked to each other by observing method invocations or attribute references. This information is readably available in statically-typed languages such as Java or C++. In more dynamic languages as is the case with JavaScript, calculating static coupling metrics is more difficult since obtaining the receiver type of a method invocation requires static analyses. This difficulty is further compounded with the fact that no single way of implementing classes is imposed by JavaScript, so the metric calculation must accommodate for different patterns used to construct objects.

Dynamic coupling metrics, first introduced by Yacoub et al. [8], measure the degree of interaction between objects from a dynamic rather than static context. Beszedes et al. [9] Dynamic Function Coupling (DFC) considers two functions

as coupled if their behaviors are "close" to each other. This is in contrast to our approach, where we consider the coupling between functions and (local or foreign) types. Hassoun et al.'s Dynamic Coupling Measure (DCM) considers the coupling between objects as varying in time [10]. Parallels can be drawn between DCM's time-variance and the set of tuples generated by our abstract interpreter. Both DFC and DCM are metrics calculated from actual runs of a program. The work of Harman et. al. [11] is the only instance we have found that uses static analysis to calculate coupling, using program slicing to measure the coupling of a system. The intuition behind their work is that code fragments that share a slice are coupled.

B. Abstract Interpretation and type inference of JavaScript

Hackett and Guo developed a fast type inference for JavaScript that uses abstract interpretation [12]. Similar to our approach, they assign types to objects according to their prototype, except for plain `Object` and `Array` objects, which have the same type if they were allocated at the same soure location. They do not attempt to relate types based on flow properties or their interfaces, as we do. Logozzo and Venter perform atomic type analysis based on abstract interpretation [13]. Their focus is not on objects, but instead on numerical analysis and domains with the aim of program optimization. Anderson et al. give an operational semantics and static type system using structural types over an idealized version of JavaScript [14]. They consider subtyping for objects based on *implementation* (a subtype must contain all members of its supertype), whereas we formulated equivalence between types based on having the same *interface*. Jensen et al. describe a type analysis for JavaScript based on abstract interpretation, with the goal of checking for the absence of common programming errors, and to provide type information for program comprehension [15]. The abstract domain for TAJS is more complex than ours and models all possible primitive data types, while our approach focuses on object types. Static analysis performed for the Self language is interesting in the context of static analysis for JavaScript, because Self influenced the design of JavaScript. Agesen et al. designed and implemented a constraint-based type inference algorithm for Self, based on implementation types, to guarantee the safety of message sends [16].

IX. Conclusion

We have demonstrated that it is possible to statically determine dynamic coupling of JavaScript functions to object types in a manner that is useful. We also showed that our dynamic coupling can be used to compute metrics with the aim of detecting bad smells related to coupling.

Object type inference is challenging for a dynamic language without classes and static typing. Manual code inspection and even simple AST analysis do not suffice. Value and control flow need to be tracked simultaneously for non-trivial JavaScript programs. Our object type inference therefore is based on information collected by an abstract interpreter that approximates run-time objects and property accesses with sufficient precision. The actual object type inference already takes common JavaScript patterns for classes and modules into account. Relating types based on flow and interfaces are promising ideas to further enhance the type inference, but they need to be further explored to become useful in practice.

Acknowledgments

Jens Nicolay is funded by the "Flemish agency for Innovation by Science and Technology" (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] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *Software, IEEE*, vol. 25, no. 5, pp. 22–29, 2008.

[2] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. IEEE, 2004, pp. 245–256.

[3] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, 2010.

[4] 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*. ACM, 1977, pp. 238–252.

[5] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[6] ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*, 5th ed., June 2011. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[7] M. Might and T. Prabhu, "Interprocedural dependence analysis of higher-order programs via stack reachability," in *Proceedings of the 2009 Workshop on Scheme and Functional Programming, Boston, Massachussetts, USA*, 2009.

[8] S. M. Yacoub, T. Robinson, and H. H. Ammar, "Dynamic metrics for object oriented design," in *Proc. International Symposium on Software Metrics*, 1999, pp. 50–58.

[9] A. Beszedes, T. Gergely, S. Farago, T. Gymothy, and F. Fischer, "The dynamic function coupling metric and its use in software evolution," in *Proc. European Conference on Software Maintenance and Reengineering*, 2007, pp. 103–112.

[10] Y. Hassoun, R. Johnson, and S. Counsell, "A dynamic runtime coupling metric for meta-level architectures," in *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*. IEEE, 2004, pp. 339–346.

[11] M. Harman, M. Okunlawon, B. Sivagurunathan, and S. Danicic, "Slice-based measurement of coupling," in *IEEE/ACM ICSE workshop on Process Modeling and Empirical Studies of Software Evolution*, Boston, Massachsetts, 1997, pp. 28–32.

[12] B. Hackett and S.-y. Guo, "Fast and precise hybrid type inference for javascript," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, 2012, pp. 239–250.

[13] F. Logozzo and H. Venter, "Rata: rapid atomic type analysis by abstract interpretation–application to javascript optimization," in *Compiler Construction*. Springer, 2010, pp. 66–83.

[14] C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for javascript," in *ECOOP 2005-Object-Oriented Programming*. Springer, 2005, pp. 428–452.

[15] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *Static Analysis*. Springer, 2009, pp. 238–255.

[16] O. Agesen, J. Palsberg, and M. I. Schwartzbach, *Type inference of SELF*. Springer, 1993.