Logic-based Verification of JavaScript Programs

Petar Maksimović

Imperial College London

with José Fragoso Santos, Philippa Gardner, Daiva Naudžiūnienė, and Thomas Wood

Formal Methods meet JavaScript

Imperial, March 2018

JaVerT: JavaScript Verification Toolchain (POPL' 18)

What is JaVerT?

JaVerT is a semi-automatic verification toolchain for JavaScript based on separation logic

What is its purpose?

JaVerT is aimed at the specialist developer wanting rich, mechanically verified specifications of critical JavaScript code

JaVerT: The Challenges

Specification Challenge: To design specifications readable by developers

(S1) Abstractions that capture key JavaScript concepts

Prototype inheritance, variable scoping, function closures

Property iteration (for-in)

(S2) Abstractions that hide JavaScript internals

Verification Challenge:

To handle the complex nature of JavaScript without simplification

(V1) Complexity of JavaScript statements

(V2) Fundamental dynamic behaviour of JavaScript

Extensible objects, dynamic property access, dynamic function calls

(V3) JavaScript internal functions

Validation Challenge

To understand what it means for the verification to be trusted

JaVerT: Specification Challenge

Specification Challenge: To design specifications readable by developers

(S1) Abstractions that capture key JavaScript concepts

Prototype inheritance, variable scoping, function closures

Property iteration (for-in)

(S2) Abstractions that hide JavaScript internals

Verification Challenge:

To handle the complex nature of JavaScript without simplification

(V1) Complexity of JavaScript statements

(V2) Fundamental dynamic behaviour of JavaScript

Extensible objects, dynamic property access, dynamic function calls

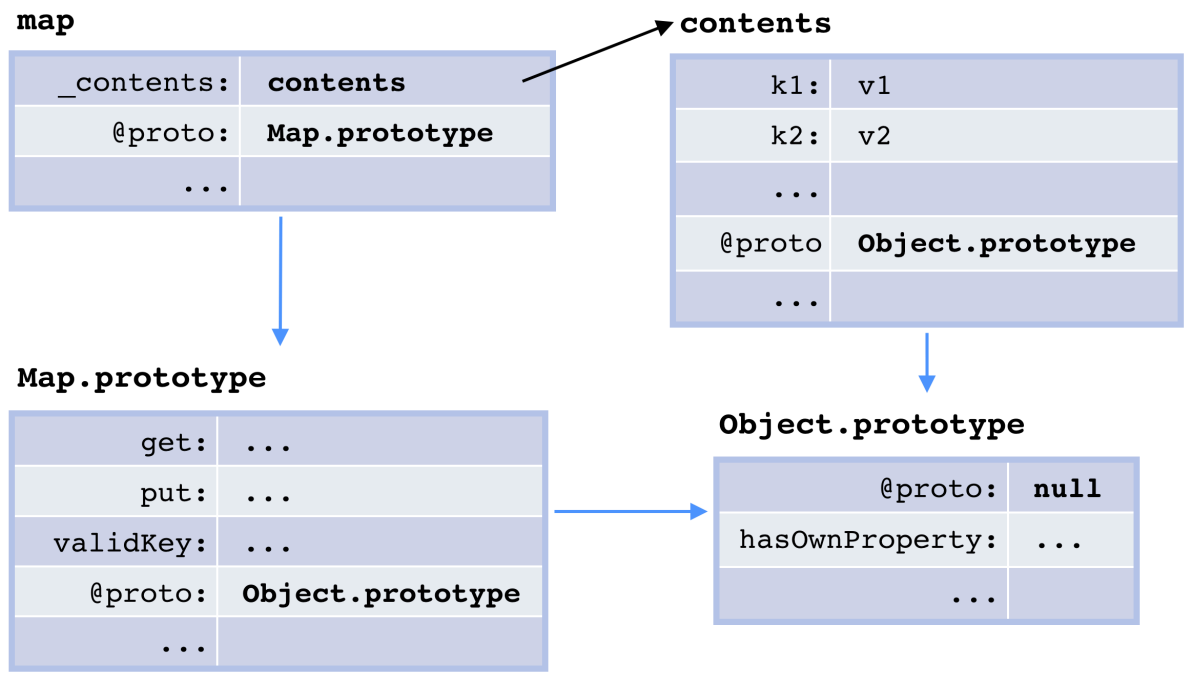
(V3) JavaScript internal functions

Validation Challenge

To understand what it means for the verification to be trusted

JavaScript: Key-Value Map

```
1  function Map () { this._contents = {} }
2
3  Map.prototype.get = function (k) {
4      if (this._contents.hasOwnProperty(k)) {
5          return this._contents[k]
6      } else { return null }
7  }
8
9  Map.prototype.put = function (k, v) {
10     var contents = this._contents;
11     if (this.validKey(k)) {
12         contents[k] = v;
13     } else { throw new Error("Invalid Key") }
14 }
15
16 Map.prototype.validKey = function (k) { ... }
```

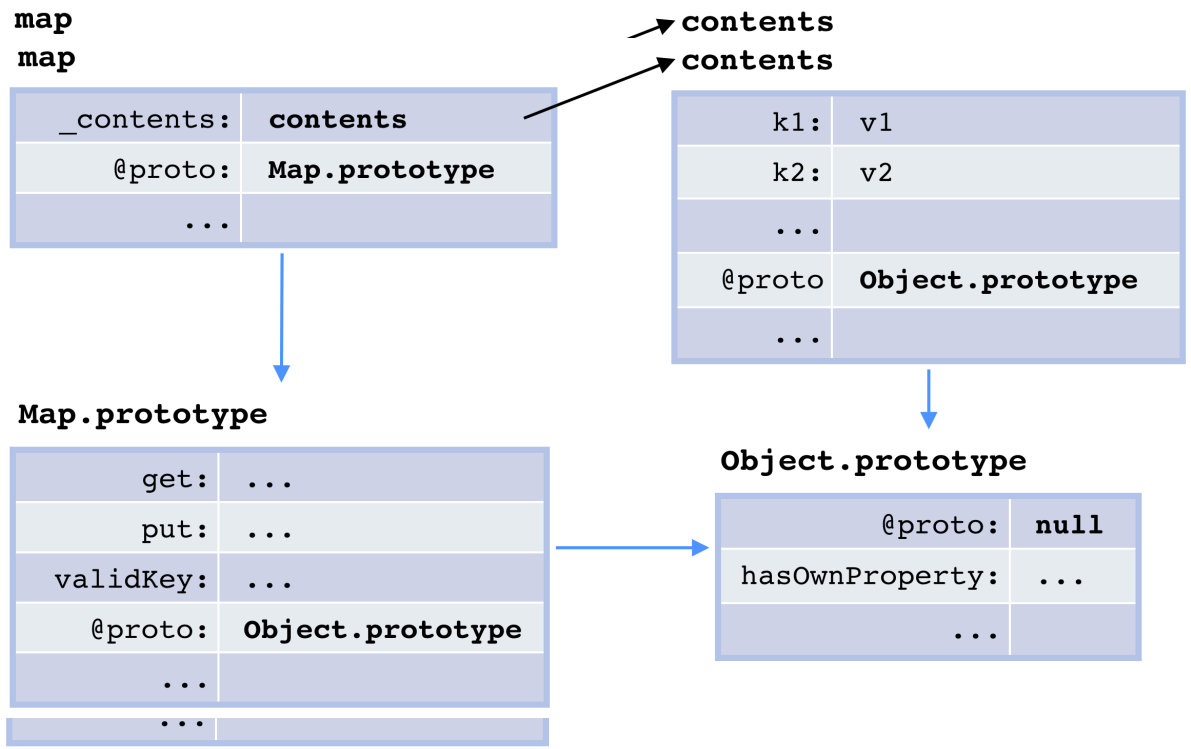


JavaScript: Key-Value Map

```
1  function Map () { this._contents = {} }
2
3  Map.prototype.get = function (k) {
4      if (this._contents.hasOwnProperty(k)) {
5          return this._contents[k]
6      } else { return null }
7  }
8
9  Map.prototype.put = function (k, v) {
10     var contents = this._contents;
11     if (this.validKey(k)) {
12         contents[k] = v;
13     } else { throw new Error("Invalid Key") }
14 }
15
16 Map.prototype.validKey = function (k) { ... }
```

Breaking the library: 1/2

```
1 var m = new Map();
2 m.get = "foo"
```

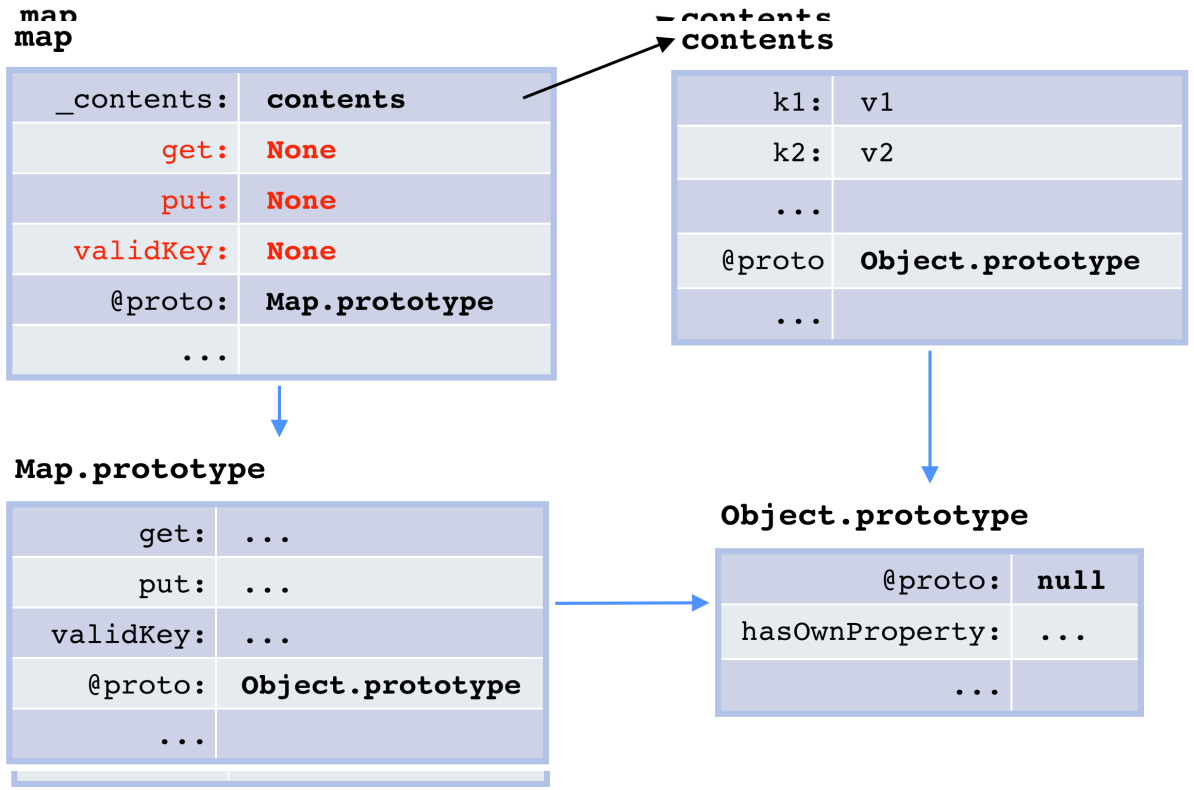


JavaScript: Key-Value Map

```
1  function Map () { this._contents = {} }
2
3  Map.prototype.get = function (k) {
4      if (this._contents.hasOwnProperty(k)) {
5          return this._contents[k]
6      } else { return null }
7  }
8
9  Map.prototype.put = function (k, v) {
10     var contents = this._contents;
11     if (this.validKey(k)) {
12         contents[k] = v;
13     } else { throw new Error("Invalid Key") }
14 }
15
16 Map.prototype.validKey = function (k) { ... }
```

Breaking the library: 2/2

```
1 var mp = Map.prototype;
2 var desc = { value: 0, writable: false };
3 Object.defineProperty(mp, "_contents", desc)
```



Prototype Safety

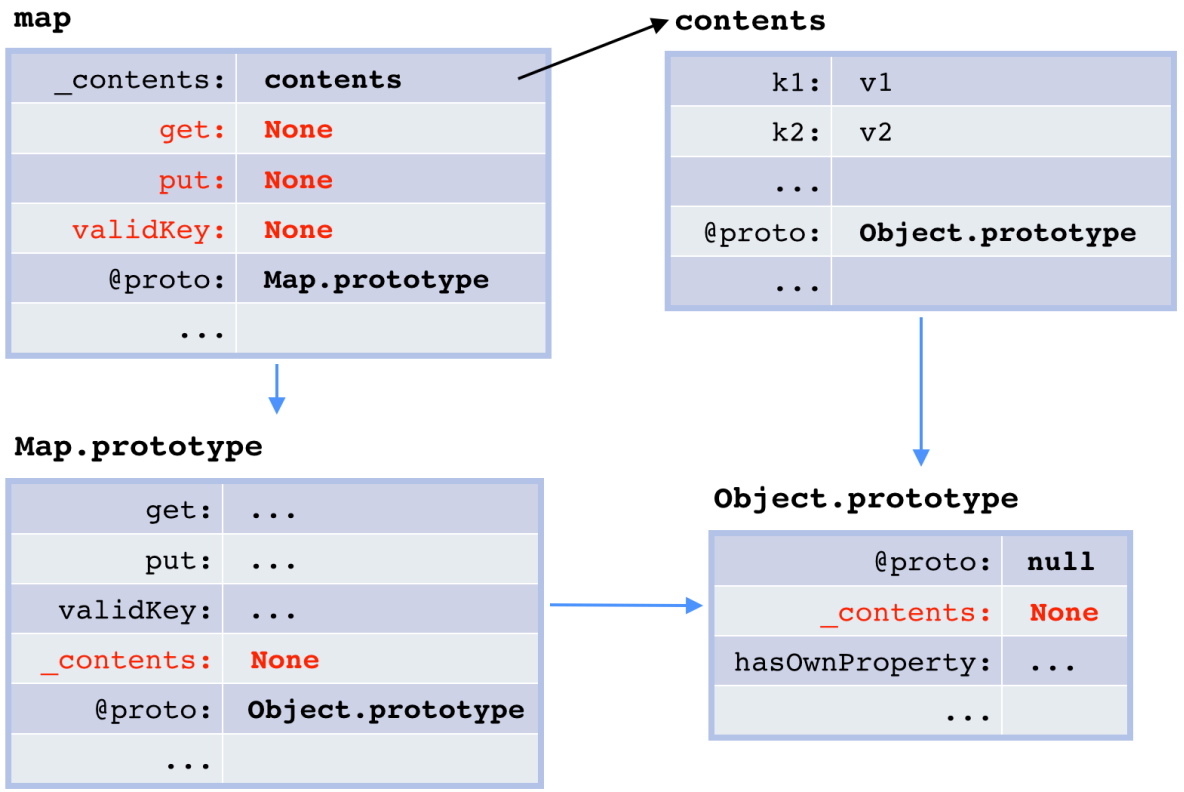
- Constructed objects cannot redefine properties that are to be found in their prototypes
- Prototypes cannot have non-writable properties that are to be present in their instances

Map objects

Must not contain **get**, **put**, and **validKey**

Map.prototype and Object.prototype

Must not contain **_contents** as non-writable



JavaScript: Key-Value Map Specification

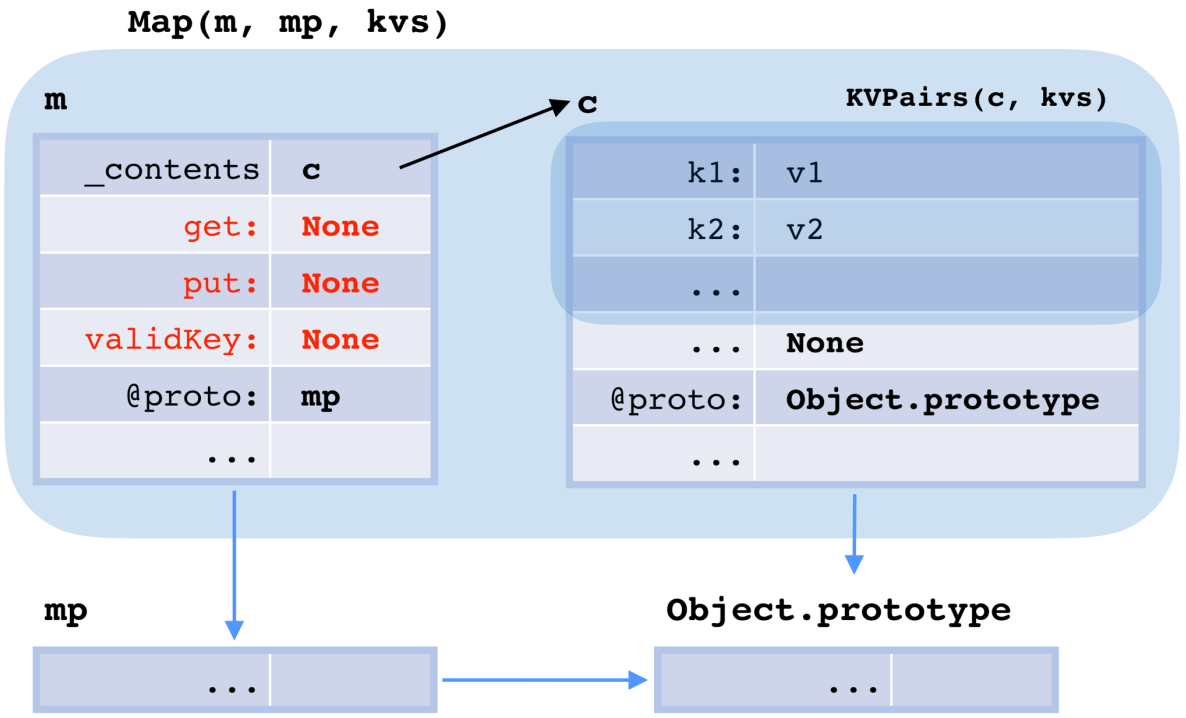
Some Basic Abstractions:

```
                JSObject(o) = (o, “@proto”) -> Object.prototype *
                              (o, “@class) -> “Object” *
                              (o, “@extensible”) -> true

 JSObjectWithProto(o, pr) = (o, “@proto”) -> pr *
                              (o, “@class) -> “Object” *
                              (o, “@extensible”) -> true

         DataProp(o, p, v) = (o, p) -> [ “d”, v, true, true, true ]

FunctionObject(o, id, sc) = (o, “@body”) -> id *
                              (o, “@scope”) -> sc * ...
```

JavaScript: Key-Value Map Specification

What does it mean to be a Map?

```
Map (m, mp, kvs) =
    JSObjectWithProto(m, mp) *
    (m, "get") -> None *
    (m, "put") -> None *
    (m, "validKey") -> None *
    DataProp(m, "_contents", c) *
    JSObject(c) *
    KVPairs(c, kvs) *
    emptyFields(c | first(kvs))
```

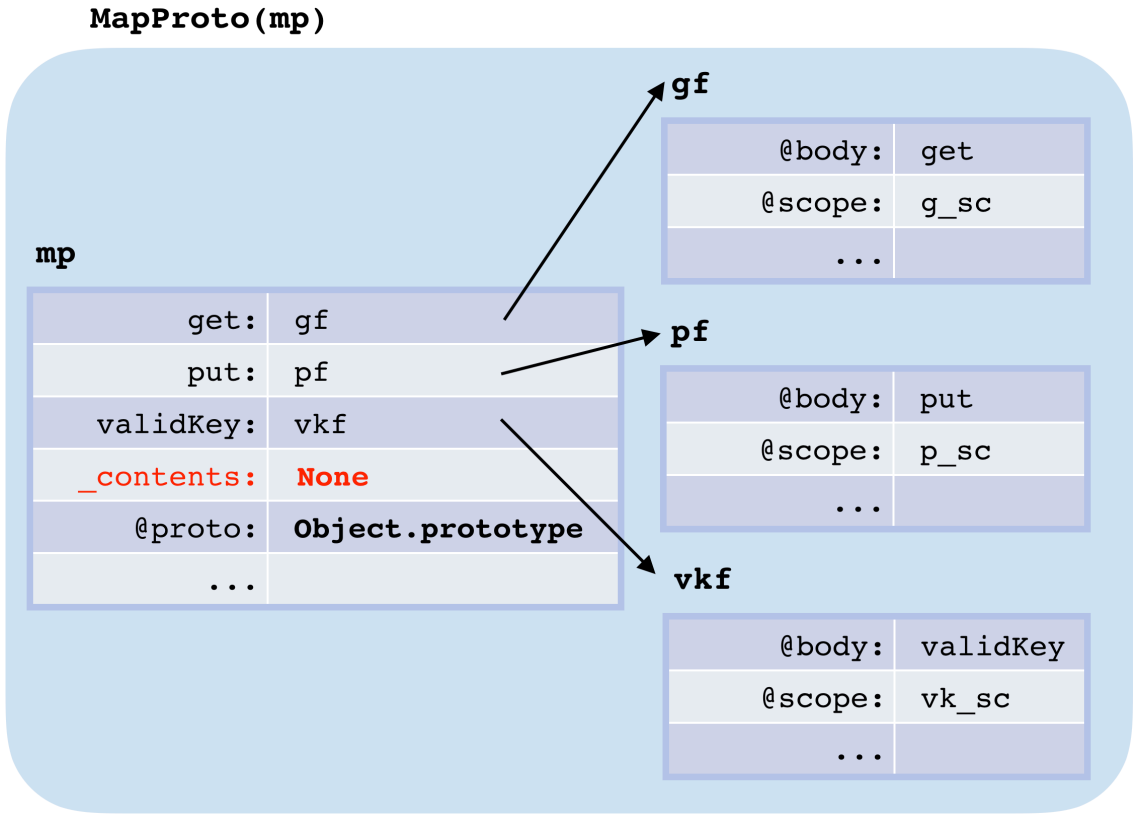


Detail: `KVPairs(c, kvs)` captures the key-value pairs of `c`.

JavaScript: Key-Value Map Specification

What does it mean to be a Map prototype?

```
MapProto (mp)  =
    JSObject(mp) *
    (mp, "_contents") -> None) *
    DataProp(mp, "get", gf) *
    FunctionObject(gf, “get”, g_sc) *
    DataProp(mp, "put", pf) *
    FunctionObject(pf, “put”, p_sc) *
    DataProp(mp, "validKey", vkf) *
    FunctionObject(vkf, “validKey”, vk_sc)
```



Caveat: The definition of MapProto cannot be part of the Map predicate because of shared resource. All maps share the same prototype.

JavaScript: Key-Value Map Specification

Specification of the get function

```
[ Map(this, mp, kvs) * MapProto(mp) *
    (k, v) in kvs * ObjProto() ]

get(k)

[ Precondition * (ret = v) ]
```

```
[ Map(this, mp, kvs) * MapProto(mp) *
!(k in first(kvs)) * ValidKey(k) * ObjProto() ]

get(k)

[ Precondition * (ret = null) ]
```

```
3 Map.prototype.get = function (k) {
4     if (this._contents.hasOwnProperty(k)) {
5         return this._contents[k]
6     } else { return null }
7 }
```

JaVerT: The Challenges

Specification Challenge: To design specifications readable by developers

(S1) Abstractions that capture key JavaScript concepts

Prototype inheritance, variable scoping, function closures

Property iteration (for-in)

(S2) Abstractions that hide JavaScript internals

Verification Challenge:

To handle the complex nature of JavaScript without simplification

(V1) Complexity of JavaScript statements

(V2) Fundamental dynamic behaviour of JavaScript

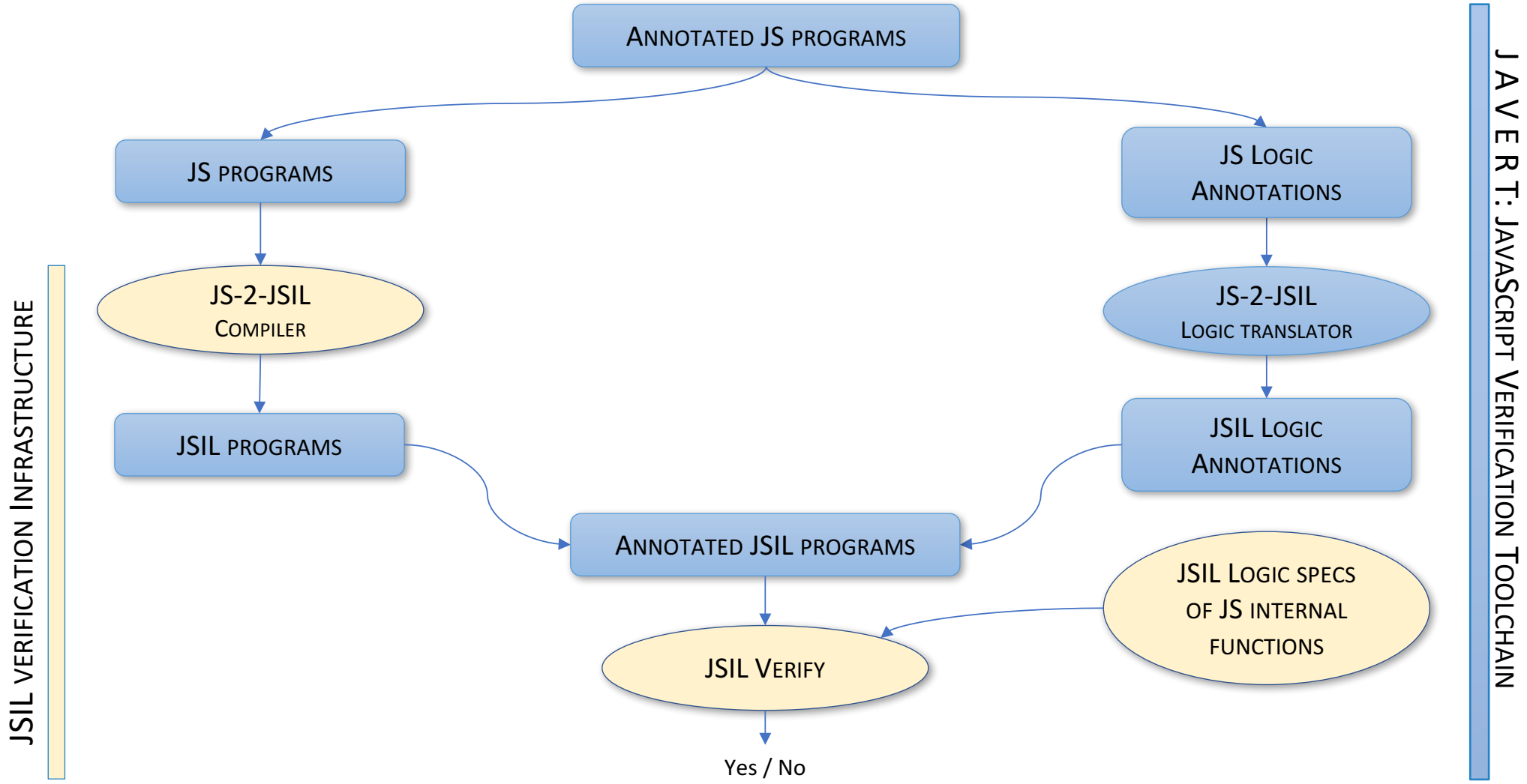
Extensible objects, dynamic property access, dynamic function calls

(V3) JavaScript internal functions

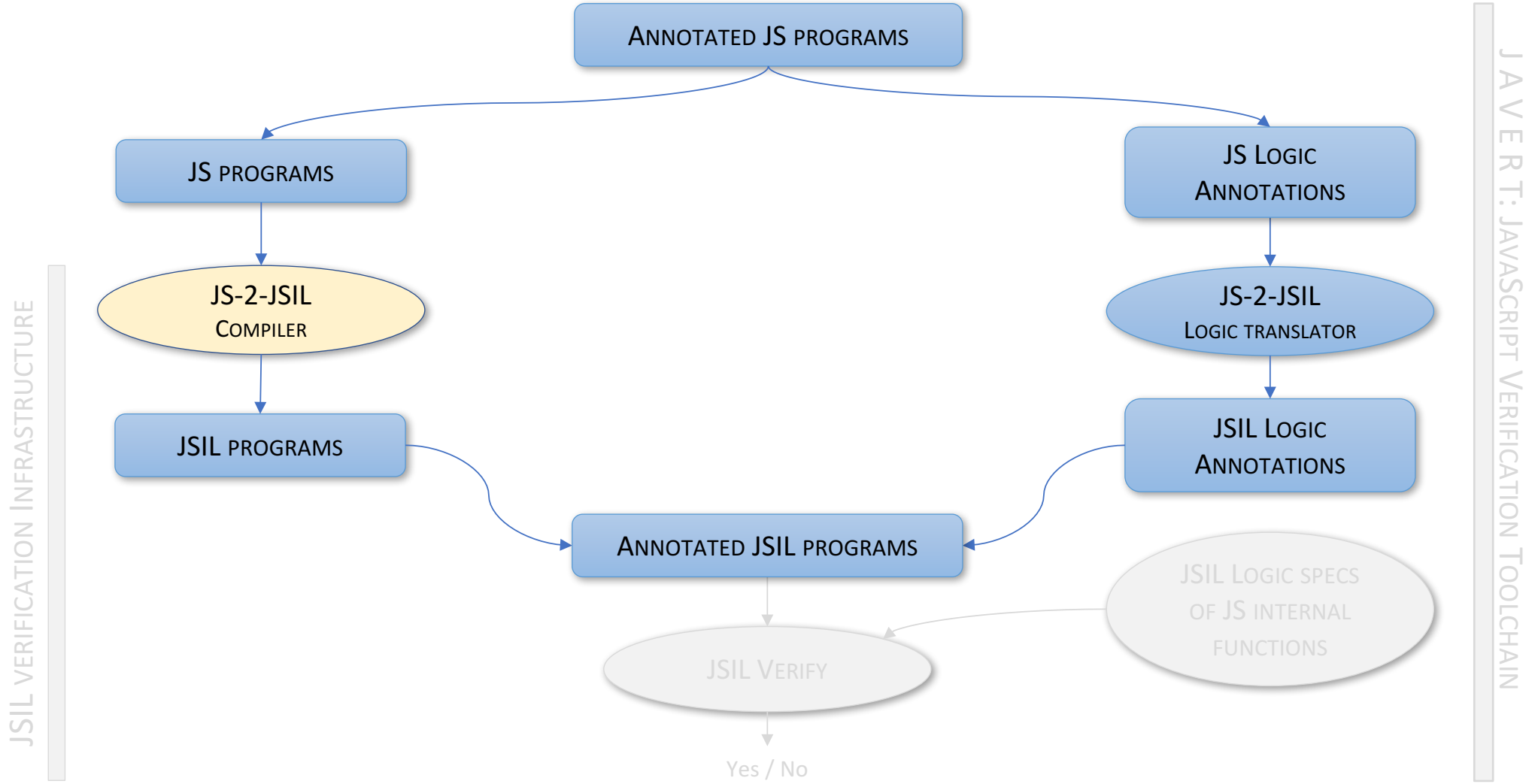
Validation Challenge

To understand what it means for the verification to be trusted

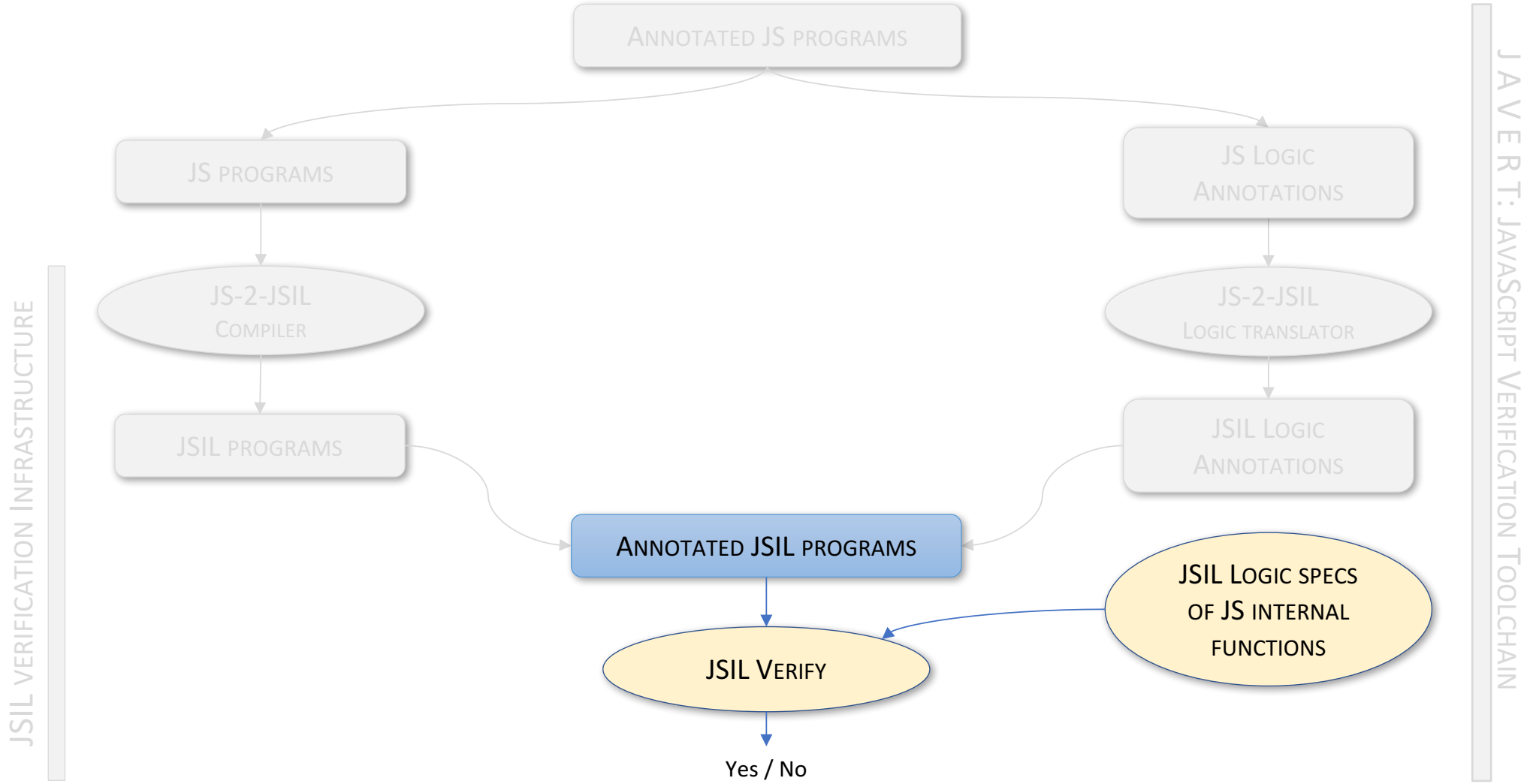
JaVerT: Overall structure



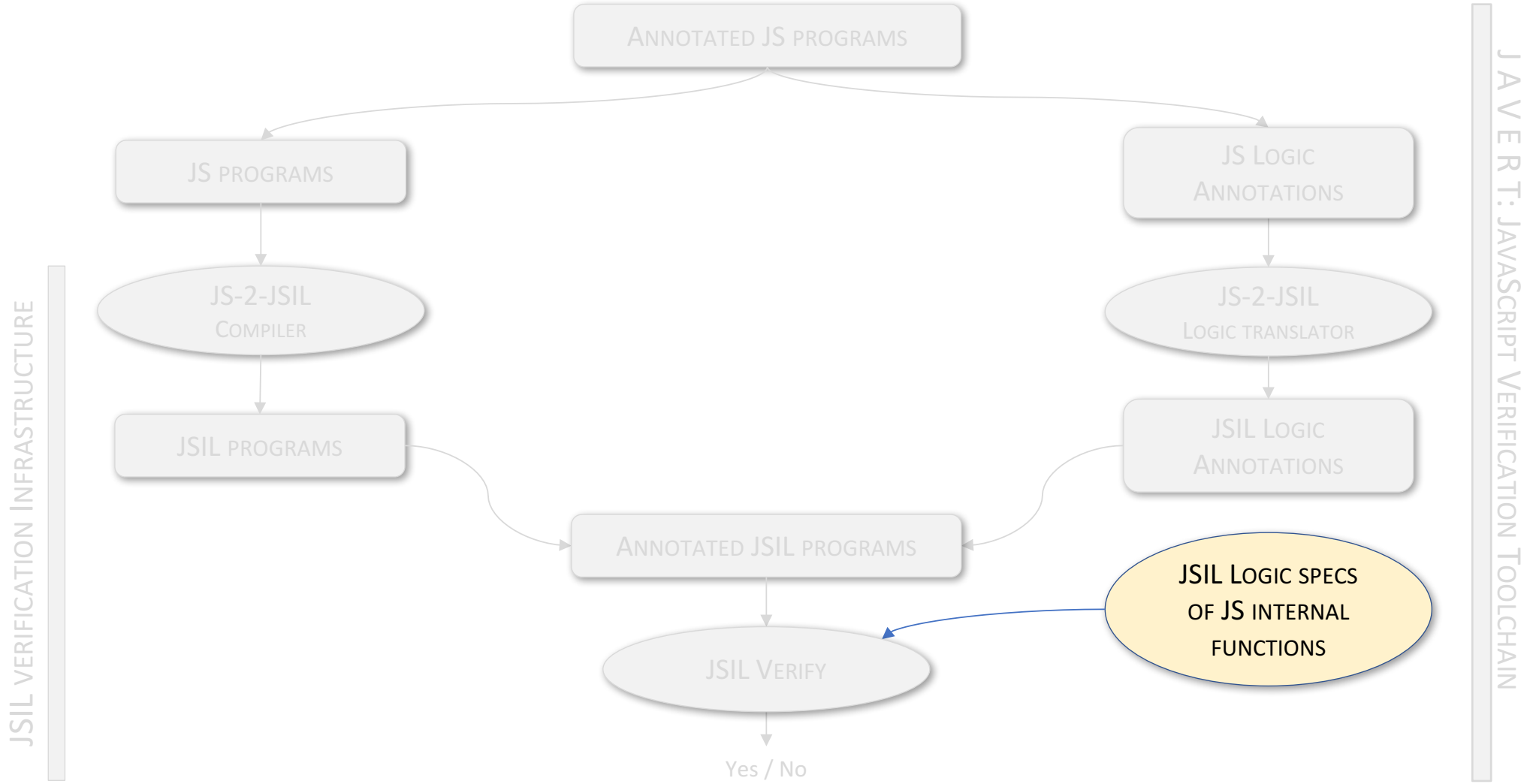
(V1) Complexity of JavaScript Statements



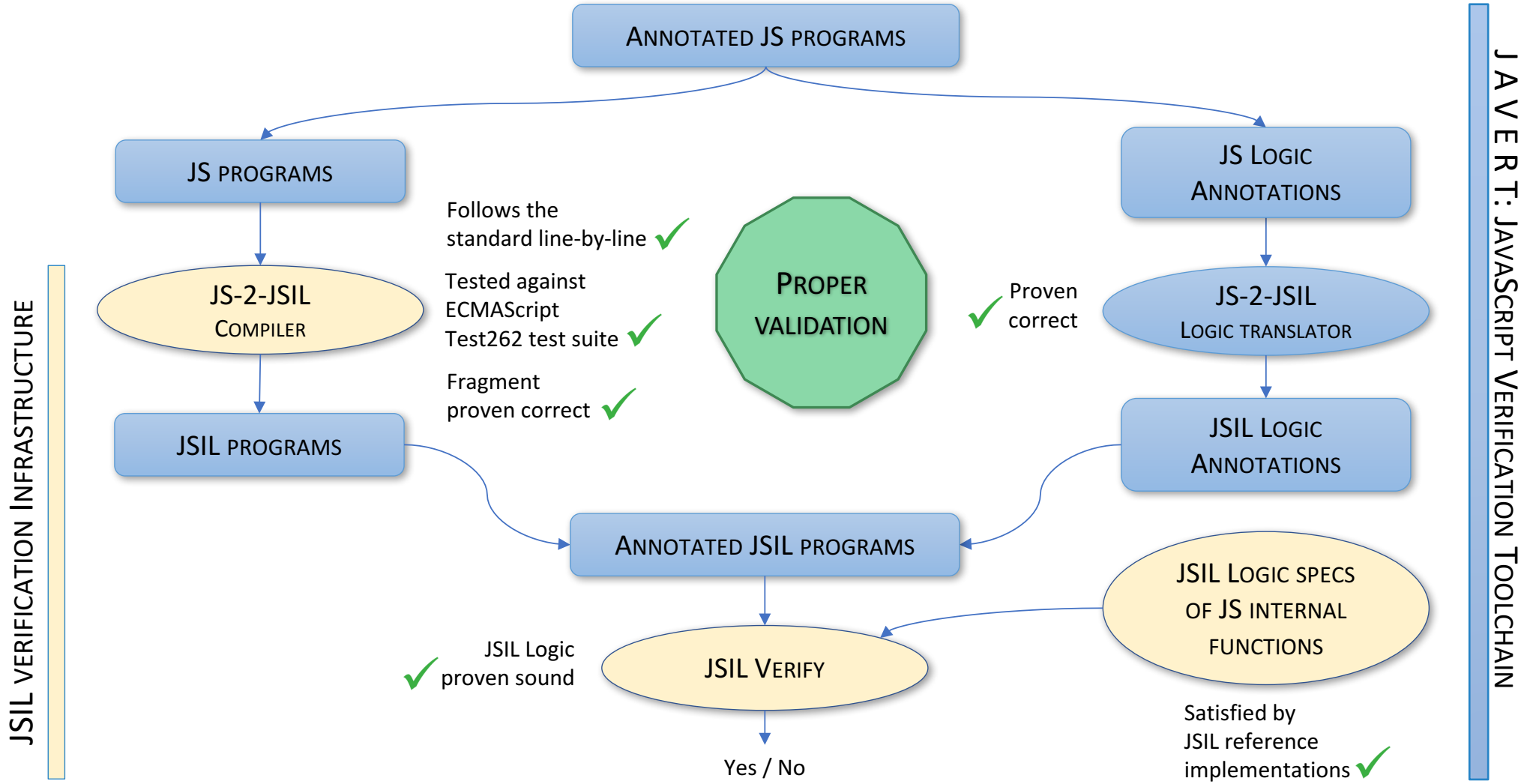
(V2) Fundamental Dynamic Behaviour of JavaScript



(V3) JavaScript Internal Functions



JaVerT: Trusted Verification



That went well...

Our specification of Map.get failed – how can we find the error?

- We assume that JaVerT is working correctly
- The specifications seem reasonable, there is no obvious error
- Lifting meaningful error messages from JSIL to JavaScript is difficult
- JaVerT's debugging proof trace for this example is 346,786 lines long

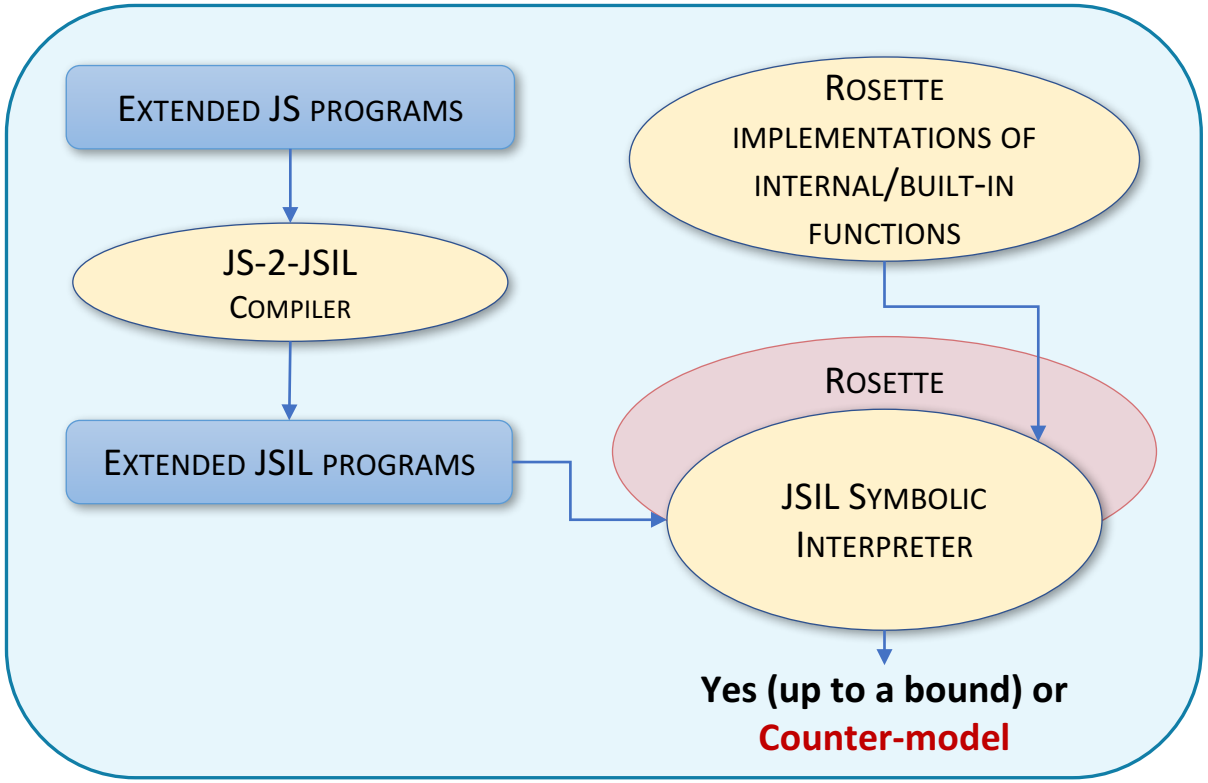
We cannot expect the developer to go through the proof trace.

We need a more robust approach.

COSETTE: SYMBOLIC TESTING FOR JAVASCRIPT

Rosette: solver-aided programming language (first-order logic)

- JS and JSIL extended with simple constructs for creating/reasoning about **symbolic values**
- JSIL **concrete** interpreter written in Rosette
- Concrete interpreter carefully written so that Rosette's solver-aided constructs are lifted, obtaining a **JSIL symbolic interpreter**
- JSIL symbolic execution formalised and proven sound; absence of false positives proven



(joint work with Julian Dolby, IBM)

COSETTE: SIMPLE SYMBOLIC TEST FOR MAP.GET

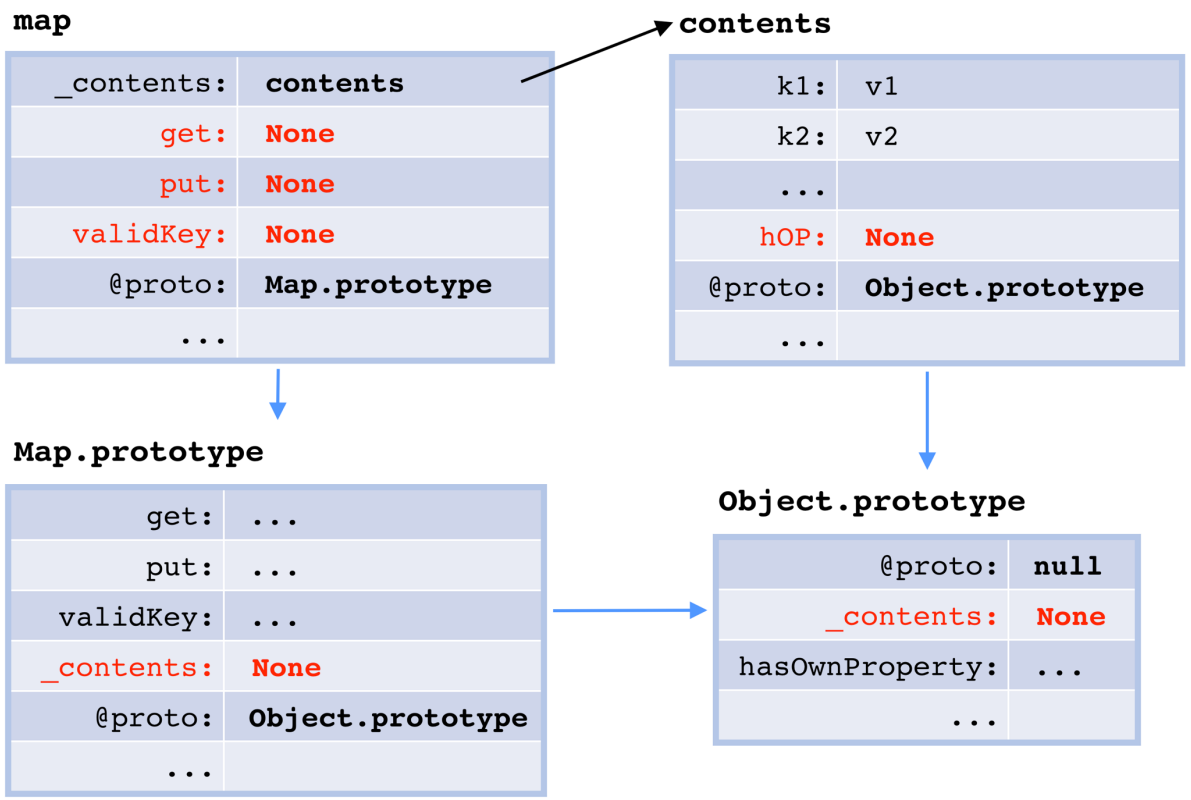
```
var k = __s;                    /* let k be a symbolic string      */
var v = __n;                    /* let v be a symbolic number      */
var m = new Map();              /* let m be an empty key-value map */

if validKey(k) {                /* let k be a valid key                          */
   m.put(k, v);                 /* put the key-value pair (k, v) in the map      */
   var w = m.get(k);            /* get the value corresponding to the key k      */
   assert(v = w);               /* that value must equal the one that we put     */
}
```

JavaScript: Key-Value Map Revisited

```
1  function Map () { this._contents = {} }
2
3  Map.prototype.get = function (k) {
4      if (this._contents.hasOwnProperty(k)) {
5          return this._contents[k]
6      } else { return null }
7  }
8
9  Map.prototype.put = function (k, v) {
10     var contents = this._contents;
11     if (this.validKey(k)) {
12         contents[k] = v;
13     } else { throw new Error("Invalid Key") }
14 }
15
16 Map.prototype.validKey = function (k) { ... }
```

Breaking the library: 3/3

```
1 var m = new Map ();
2 m.put("hasOwnProperty", "bar")
```



Prototype Safety Revisited

- Constructed objects cannot redefine properties that are to be found in their prototypes
- Prototypes cannot have non-writable properties that are to be present in their instances

Map objects

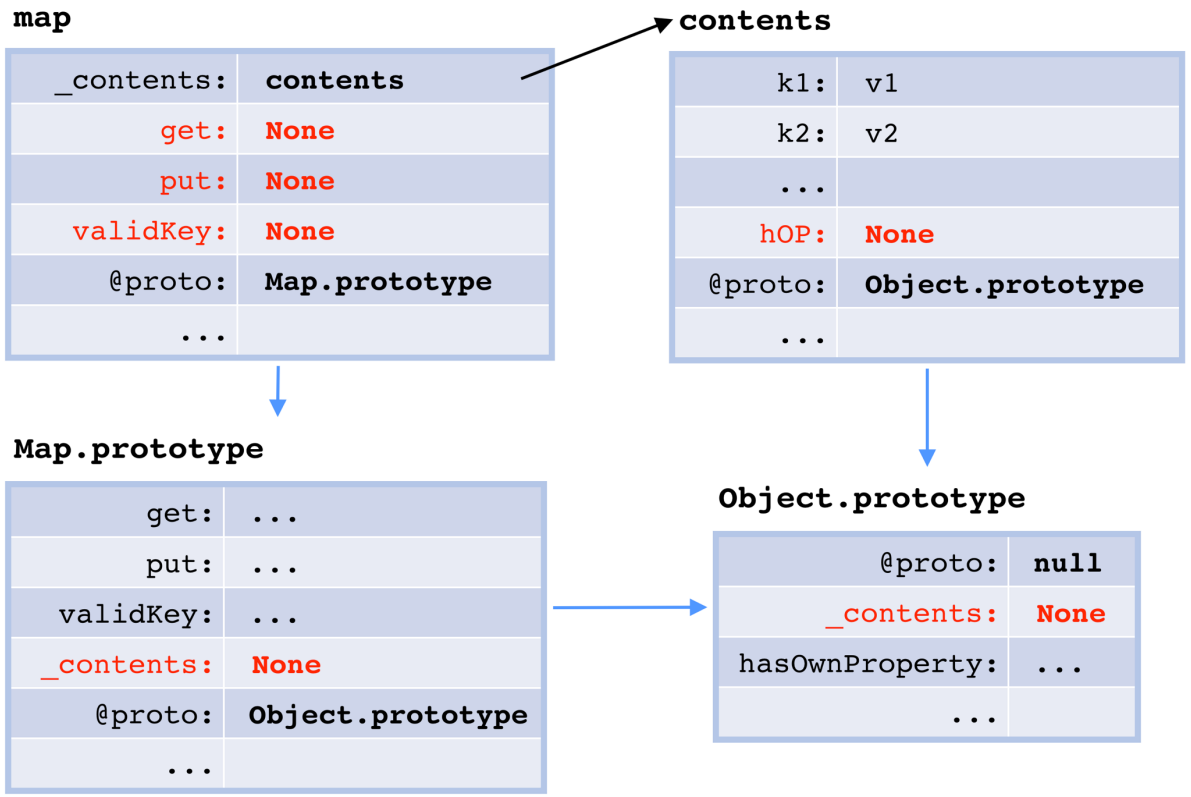
Must not contain **get**, **put**, and **validKey**

Map.prototype and Object.prototype

Must not contain **_contents** as non-writable

Map contents

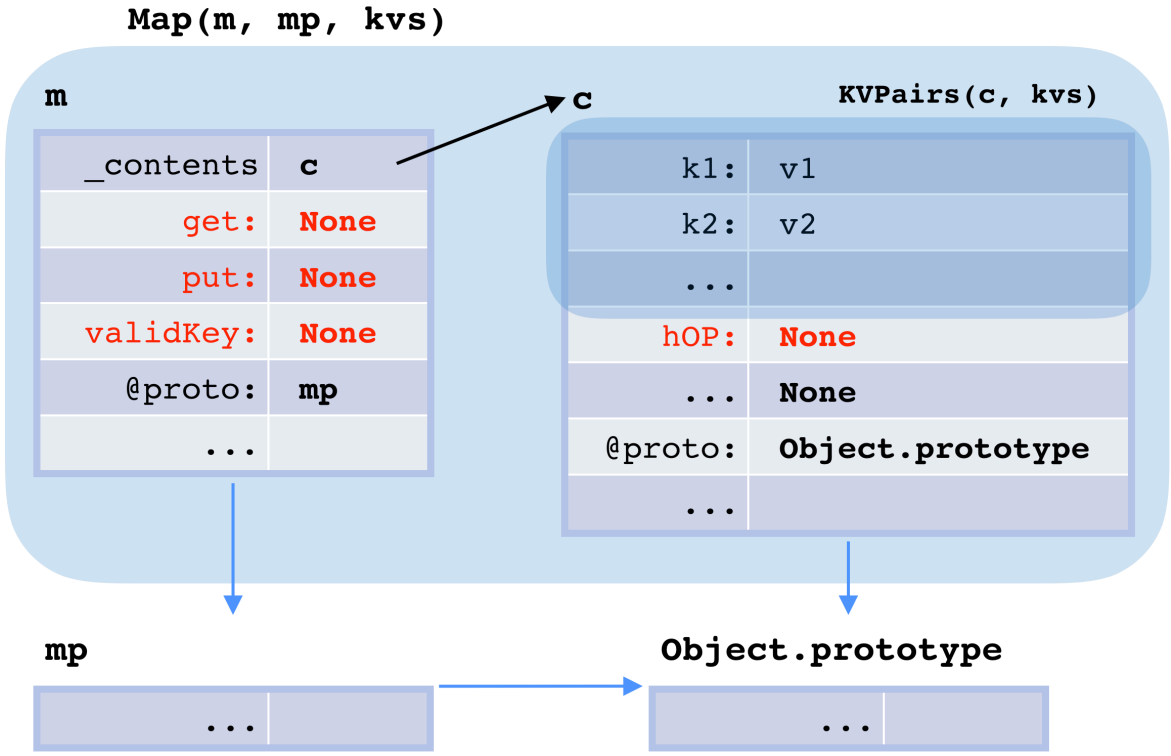
Must not contain **hasOwnProperty** as non-writable



JavaScript: Key-Value Map Specification Revisited

What does it mean to be a Map?

```
Map (m, mp, kvs) :=
    JSObjectWithProto(m, mp) *
    (m, "get") -> None *
    (m, "put") -> None *
    (m, "validKey") -> None *
    DataProp(m, "_contents", c) *
    JSObject(c) *
    (c, "hasOwnProperty") -> None *
    KVPairs(c, kvs) *
    emptyFields(c | first(kvs)
                    U "hasOwnProperty")
```



Detail: `KVPairs(c, kvs)` captures the key-value pairs of `c`.

JaVerT: Summary

Specification Challenges

(S1) Abstractions capturing key JavaScript concepts
- Prototype inheritance, variable scoping, function closures ✓
- Property iteration (for-in) ✗

(S2) Abstractions that hide JavaScript internals ✓

Verification Challenges

(V1) Complexity of JavaScript statements ✓
(V2) Fundamental dynamic behaviour of JavaScript ✓
(V3) Internal functions ✓

Validation Challenges

Correctness of JS-2-JSIL ✓
Correctness of assertion translation ✓
Soundness of JSIL Logic ✓
Correctness of specifications for internal functions ✓

Caveat: No higher-order reasoning yet ✗

JaVerT: Further Verified Examples

Example	#JS	#JSIL	#specs	t(s)
Key-value map	23	523	9	3.37
ID Generator	16	330	4	0.73
Priority queue	46	1003	10	7.14
BST	70	1032	5	7.38
Insertion sort	24	415	2	1.78
Test262 examples	113	1367	16	3.46

ID Generator: function closures

Priority queue: library based on a real-world Node.js library

Binary search trees: set reasoning

Insertion sort: list reasoning

Test262 examples: complex JS statements (switch, try/catch/finally)

Future: The JaVerT Ecosystem