Static Analysis with Demand-Driven Value Refinement

Benno Stein, **Benjamin Barslev Nielsen,** Bor-Yuh Evan Chang & Anders Møller

AARHUS UNIVERSITET

CUPLV
Programming Languages and Verification
University of Colorado Boulder

Sound static analysis for JavaScript

- Static analysis for JavaScript is very challenging

o[m]()

Sound static analysis for JavaScript

- Static analysis for JavaScript is very challenging

o[m]()

Dynamic object structure

Sound static analysis for JavaScript

- Static analysis for JavaScript is very challenging

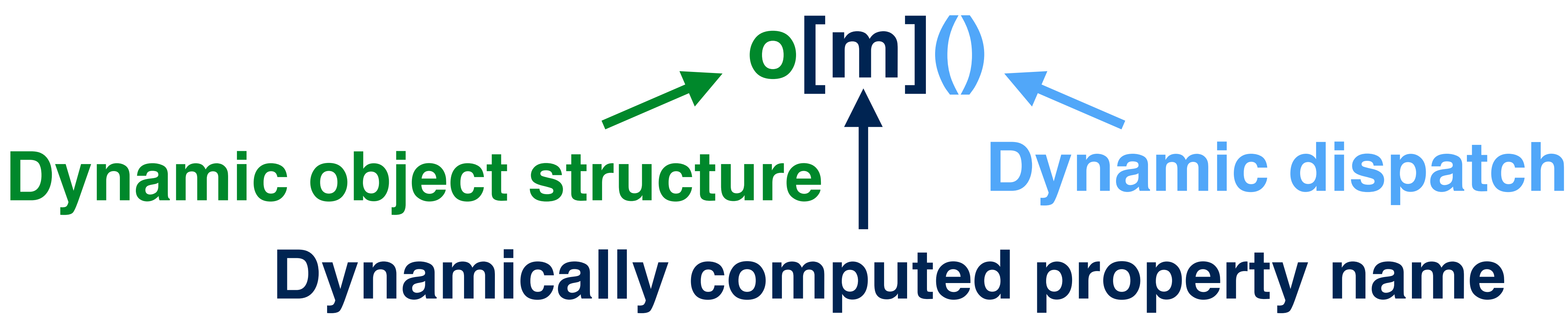
o[m]()

Dynamic object structure

Dynamically computed property name

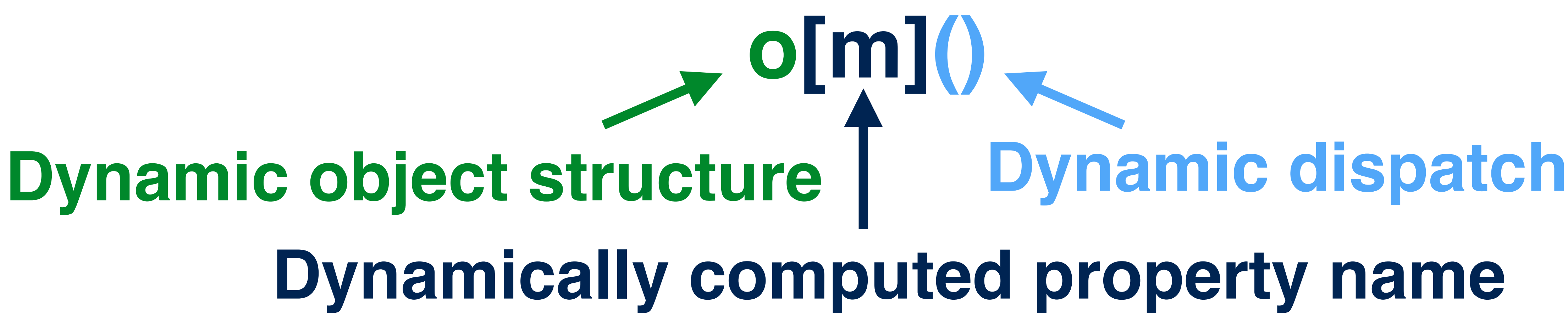
Sound static analysis for JavaScript

- Static analysis for JavaScript is very challenging



Sound static analysis for JavaScript

- Static analysis for JavaScript is very challenging

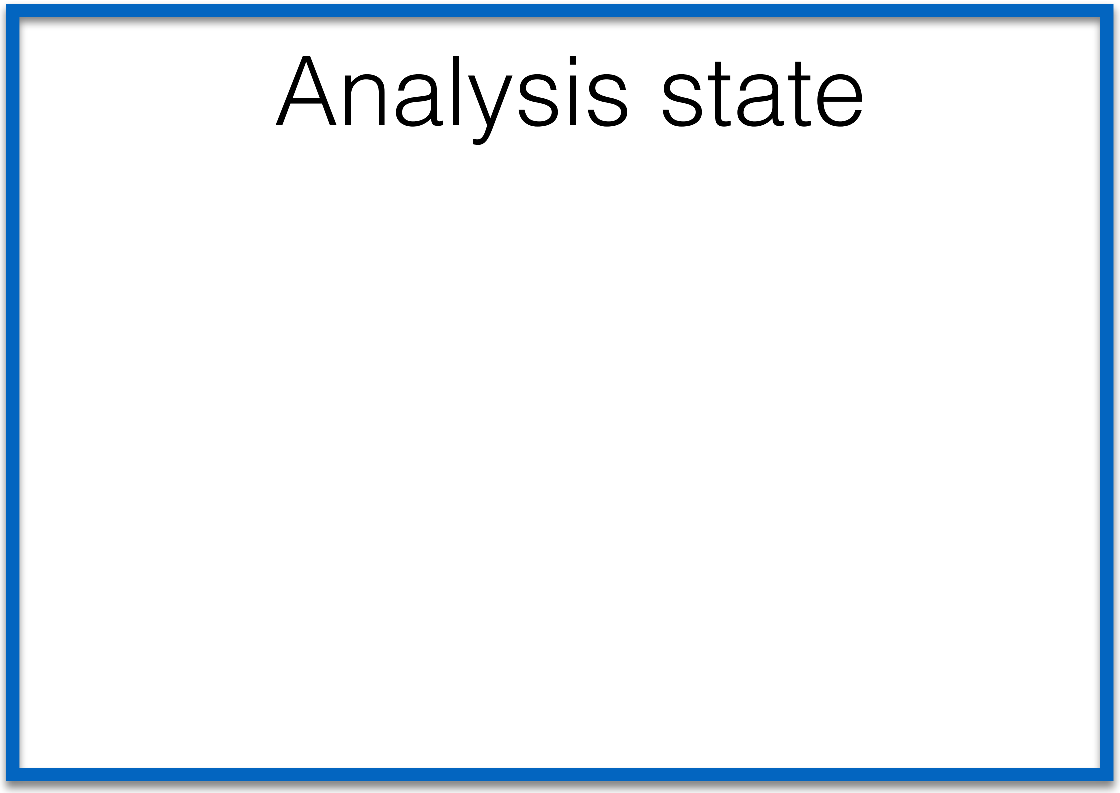


- Critical precision losses renders analysis useless
  - Too much spurious data-flow

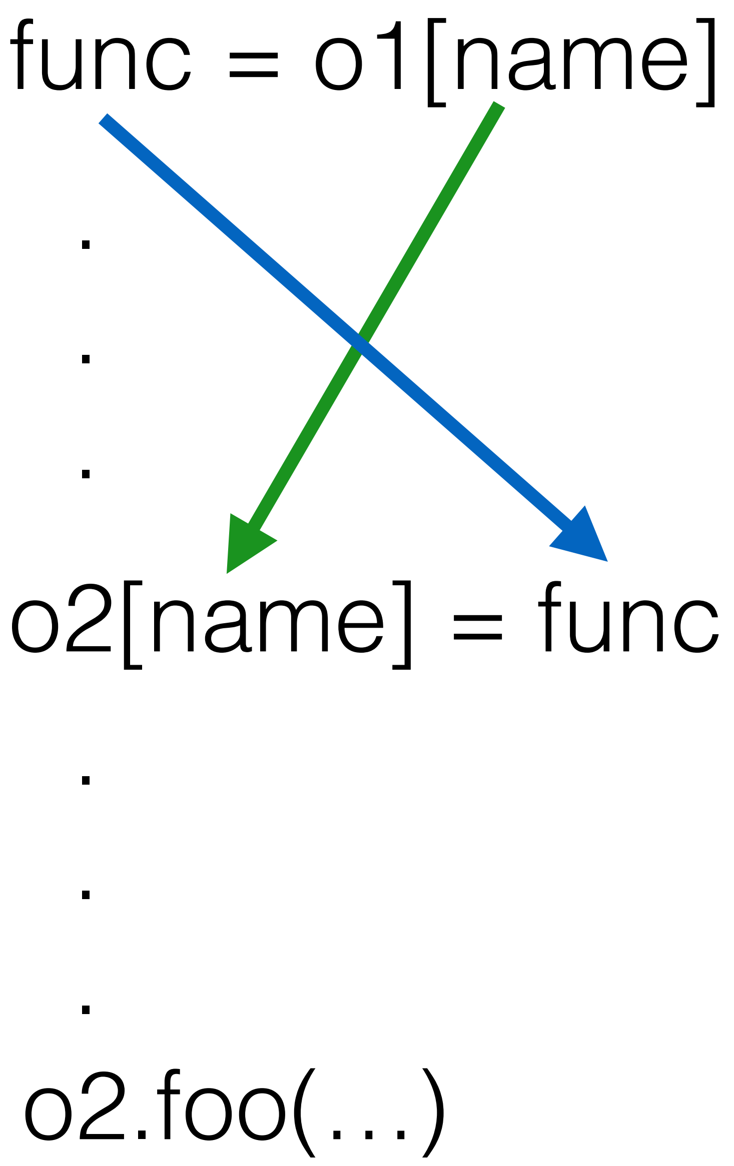
State-of-the-art data-flow analyzers

- Fail to analyze load of some very popular libraries
  - Critical precision losses occur
- Common characteristics
  - Forwards whole program analysis
  - Tracks data-flow, e.g., strings, functions and other objects
  - Non-relational
  - Aims to mitigate critical precision losses by:
    - Context sensitivity
    - Syntactic patterns and special-case techniques

Critical code example



Example program



Critical code example

Analysis state

o1 = {foo: f1, bar: f2}

name = T_{str}

o2 = {}

Example program

```
func = o1[name]
.
.
.
o2[name] = func
.
.
.
o2.foo(…)
```

Critical code example

Analysis state

o1 = {foo: f1, bar: f2}

name = T_{str}

o2 = {}

func = f1|f2

Example program

```
func = o1[name]
.
.
.
o2[name] = func
.
.
.
o2.foo(…)
```

Critical code example

Analysis state

o1 = {foo: f1, bar: f2}

name = T_{str}

o2 = { T_{str} : f1|f2}

func = f1|f2

Example program

```
func = o1[name]
.
.
.
o2[name] = func
.
.
.
o2.foo(…)
```

Critical code example

Example program

Analysis state

o1 = {foo: f1, bar: f2}

name = T_{str}

o2 = { T_{str} : f1|f2}

func = f1|f2

```
func = o1[name]
.
.
.
o2[name] = func
.
.
.
o2.foo(…)
```

Resolves both f1 and f2

The Lodash library

```
function baseFor(object, iteratee) {
    ...
    while (length--) {
        var key = props[++index];
        iteratee(object[key], key)
    }
}

mixin(lodash, (function() {
    var source = {};
    baseFor(lodash, function(func, methodName) {
        if (!hasOwnProperty.call(lodash.prototype, methodName)) {
            source[methodName] = func;
        }
    });
    return source;
}()));
```

Critical code example

Example program

Analysis state

o1 = {foo: f1, bar: f2}

name = T_{str}

o2 = {}

func = f1|f2

```
func = o1[name]
.
.
.
o2[name] = func
.
.
.
o2.foo(…)
```

Critical code example

Analysis state

o1 = {foo: f1, bar: f2}

name = T_{str}

o2 = {}

func = f1|f2

Example program

```
func = o1[name]
.
.
.
o2[name] = func
.
.
.
o2.foo(…)
```

Demand-driven value refinement

Regain relational information through refinement queries

Without modifying base analysis domain

Refinement query: What is x, when y $\mapsto \hat{v}$?

What value can variable x have,
given that y has value $\hat{v}$?

Critical code example

Analysis state

o1 = {foo: f1, bar: f2}

name = $\top_{str}$

o2 = {}

func = f1|f2

Example program

```
func = o1[name]
.
.
.
o2[name] = func
.
.
.
o2.foo(…)
```

Critical code example

Analysis state

o1 = {foo: f1, bar: f2}

name = T_{str}

o2 = {}

func = f1|f2

What is name, when func ↦ f1? →

What is name, when func ↦ f2?

Example program

```
func = o1[name]
.
.
.
o2[name] = func
.
.
.
o2.foo(…)
```

Backwards abstract interpreter for value refinement

- Backwards goal-directed from the query location
- Separation logic based abstract domain
  - Intuitionistic - constraints hold for all extensions
  - Special symbolic variable RES represents value being refined

symbolic variables	$\hat{x}, \hat{y}, \hat{z}$, RES	$\in \widehat{Var}$
symbolic stores	$\varphi \in \widehat{Store}$	$::= \hat{h} \wedge \pi \mid \varphi_1 \vee \varphi_2$
heap constraints	$\hat{h}$	$::=$ true $\mid$ unalloc$(\hat{x}) \mid x \mapsto \hat{x} \mid \hat{x}_1[\hat{x}_2] \mapsto \hat{x}_3 \mid \hat{h}_1 * \hat{h}_2$
pure constraints	π	$::=$ true $\mid \hat{e} \mid \pi_1 \wedge \pi_2$
symbolic expressions	$\hat{e} \in \widehat{Expr}$	$::= \hat{x} \mid \hat{v} \mid \hat{e}_1 \oplus \hat{e}_2$

Backwards abstract interpreter for value refinement

- Based on refutation sound Hoare triples $\langle \varphi \rangle \; s \; \langle \varphi' \rangle$
- Refutation soundness:

  For all concrete runs where φ' holds after s, the state before s must satisfy φ.

- Encoding refinement queries:

What is x, when y $\mapsto \hat{v}$? $\rightsquigarrow \left\langle \text{x} \mapsto \text{RES} * \text{y} \mapsto \hat{y} \wedge \hat{y} = \hat{v} \right\rangle$

Critical code example

```
func = o1[name]

→ o2[name] = func
```

Critical code example

Refinement query: What is name, when func $\mapsto$ f1?

```
func = o1[name]
```

→

```
o2[name] = func
```

Critical code example

Refinement query: What is name, when func $\mapsto$ f1?

func = o1[name]

$$\left\langle \text{name} \mapsto \text{RES} * \text{func} \mapsto \widehat{\text{func}} \wedge \widehat{\text{func}} = \text{f1} \right\rangle$$

$\longrightarrow$ o2[name] = func

Critical code example

Refinement query: What is name, when func $\mapsto$ f1?

$$\left\langle \text{name} \mapsto \text{RES} * \text{o1} \mapsto \widehat{\text{o1}} * \widehat{\text{o1}}[\text{RES}] \mapsto \widehat{\text{func}} \wedge \widehat{\text{func}} = \text{f1} \right\rangle$$

func = o1[name]

$$\left\langle \text{name} \mapsto \text{RES} * \text{func} \mapsto \widehat{\text{func}} \wedge \widehat{\text{func}} = \text{f1} \right\rangle$$

$\longrightarrow$ o2[name] = func

Leveraging forwards analysis state

Analysis state

o1 = {foo: f1, bar: f2}

$$\left\langle \text{name} \mapsto \text{RES} * \text{o1} \mapsto \widehat{\text{o1}} * \widehat{\text{o1}}[\text{RES}] \mapsto \widehat{\text{func}} \wedge \widehat{\text{func}} = \text{f1} \right\rangle$$

Refinement result is the values of RES satisfying:

$$\text{o1}[\text{RES}] = \text{f1}$$

Refinement result: “foo”

Critical code example

Analysis state

o1 = {foo: f1, bar: f2}

name = T_{str}

o2 = {}

func = f1|f2

Example program

```
func = o1[name]
.
.
.
o2[name] = func   ←
.
.
.
o2.foo(…)
```

Critical code example

Analysis state

$o1 = \{foo: f1, bar: f2\}$

$name = \top_{str}$

$o2 = \{\}$

$func = f1|f2$

What is name, when func $\mapsto$ f1? →

What is name, when func $\mapsto$ f2?

Example program

```
func = o1[name]
.
.
.
o2[name] = func
.
.
.
o2.foo(…)
```

Critical code example

Analysis state

o1 = {foo: f1, bar: f2}

name = $\top_{str}$

o2 = {}

func = f1|f2

What is name, when func ↦ f1?

"foo"

What is name, when func ↦ f2?

"bar"

Example program

```
func = o1[name]
.
.
.
o2[name] = func
.
.
.
o2.foo(…)
```

Critical code example

Analysis state

o1 = {foo: f1, bar: f2}

name = $\top_{str}$

o2 = {foo: f1, bar: f2}

func = f1|f2

What is name, when func ↦ f1?

“foo”

What is name, when func ↦ f2?

“bar”

Example program

```
func = o1[name]
.
.
.
o2[name] = func
.
.
.
→ o2.foo(…)
```

Critical code example

Analysis state

o1 = {foo: f1, bar: f2}

name = T_{str}

o2 = {foo: f1, bar: f2}

func = f1|f2

Example program

```
func = o1[name]
.
.
.
o2[name] = func
.
o2.foo(...)
```

What is name, when func ↦ f1?

"foo"

What is nam

"bar"

Resolves only f1

Implementation for JavaScript

- **TAJS$_{VR}$**: **TAJS** extended with demand-driven value refinement
- **TAJS** is a state-of-the-art analyzer for JavaScript
  - Implemented in Java
  - Active research since 2009
- **VR$_{JS}$**: Backwards abstract interpreter for JavaScript for answering refinement queries
  - Implemented in Scala from scratch

AARHUS UNIVERSITET

Compared to state-of-the-art

	#tests	TAJS	CompAbs	$TAJS_{VR}$
Underscore[1]	182	**0 %**	**0 %**	**95% (2.9s)**
Lodash3[1]	176	**0 %**	**0 %**	**98% (5.5s)**
Lodash4[1]	306	**0 %**	**0 %**	**87% (24.7s)**
Prototype[2]	6	**0 %**	**33% (23.1s)**	**83% (97.7s)**
Scriptaculous[2]	1	**0 %**	**100% (62.0s)**	**100% (236.9s)**
JQuery[3]	71	**7% (14.4s)**	**0 %**	**7% (17.2s)**
JSAI tests[4]	29	**86% (12.3s)**	**34% (32.4s)**	**86% (14.3s)**

"x% (y)" means succeeded x% of test cases with average time y

[1]: Most popular functional utility libraries
[2]: Wei et al. [2016]
[3]: Andreasen and Møller [2014]
[4]: Kashyap et al. [2014] & Dewey et al. [2015]

Compared to state-of-the-art

	#tests	TAJS	CompAbs	TAJS$_{VR}$
Underscore[1]	182	0 %	0 %	95% (2.9s)
Lodash3[1]	176	0 %	0 %	98% (5.5s)
Lodash4[1]	306	0 %	0 %	87% (24.7s)
Prototype[2]	6	0 %	33% (23.1s)	[illegible]
Scriptaculous[2]	1	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible] (17.2s)
[illegible]	[illegible]	[illegible] (12.3s)	34% (32.4s)	86% (14.3s)

TAJS$_{VR}$ succeeds analyzing 92% of Underscore and Lodash tests, which all are unanalyzable by existing analyzers

"x% (y)" means succeeded x% of test cases with average time y

[1]: Most popular functional utility libraries
[2]: Wei et al. [2016]
[3]: Andreasen and Møller [2014]
[4]: Kashyap et al. [2014] & Dewey et al. [2015]

Value refinement insights

- Value refinement is triggered in few locations
  - In Lodash4, it is triggered in 7 locations in >17000 LoC
- Almost all queries are solved successfully (>99%)
- Queries are answered efficiently (Avg. ~10ms)
- Answering a query requires visiting few locations
  - Typically below 40
- Many queries requires interprocedural reasoning

Conclusion

- New technique: Demand-Driven Value Refinement
  - Relational reasoning on top of non-relational analysis
  - Eliminates critical precision loss on-the-fly
  - Uses backwards analysis for gaining relational precision
  - Exploiting forwards analysis state allows efficient refinements
- Experimental evaluation
  - First analysis capable of analyzing most popular JavaScript library
  - No significant overhead for incorporating backwards analyzer
  - Open-source: https://www.brics.dk/TAJS/VR/

Value refinement statistics

	Ref locs	Avg # queries	Succ (%)	Refiner time (%)	Avg query time (ms)	Avg. locs visited	Inter (%)
Underscore	5	268	99.98	22.4	2.43	5.05	0.10
Lodash3	12	475	99.99	47.2	5.46	10.47	40.22
Lodash4	7	1284	99.97	52.0	10.01	10.09	25.75
Prototype	4	188	100	2.5	13.08	39.98	48.10
Scriptaculous	2	601	100	3.4	13.21	36.91	42.26
JQuery	5	1	87.5	0.1	13.57	7.1	2.86
JSAI tests	0	-	-	-	-	-	-