# Systematic Approaches for Increasing Soundness and Precision of Static Analyzers

Anders Møller

Aarhus University

Joint work with Esben Sparre Andreasen and Benjamin Barslev Nielsen


http://casa.au.dk/

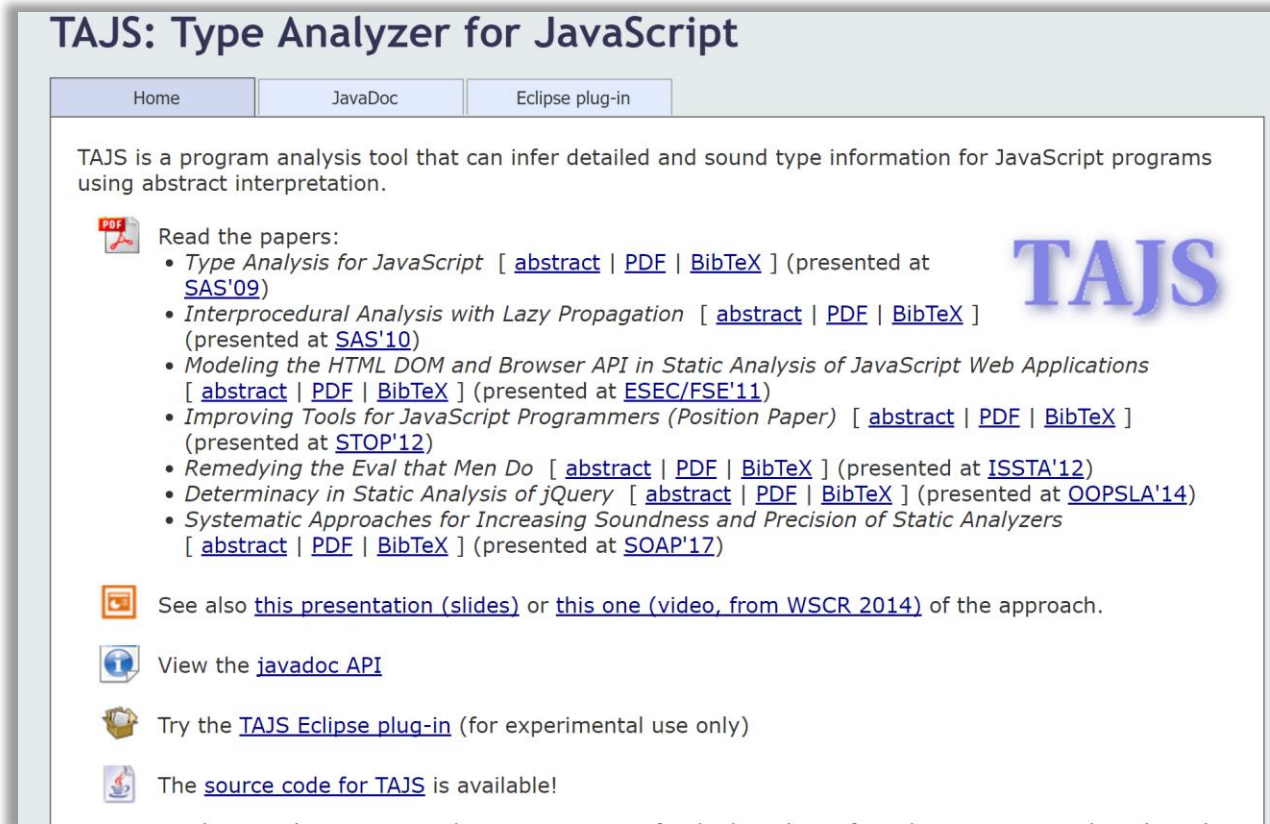

European Research Council

Established by the European Commission

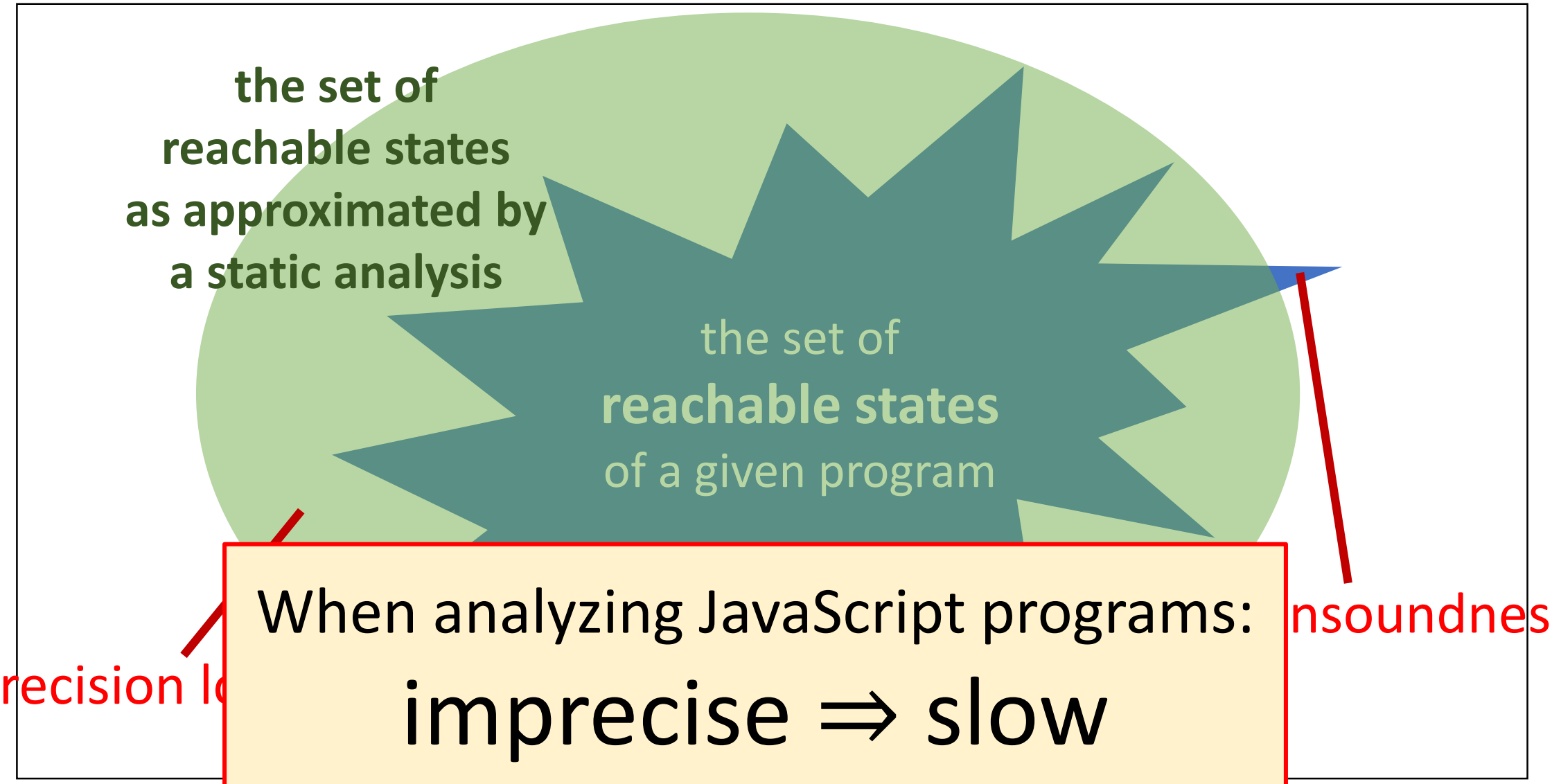
# Some lessons learned from developing a static analyzer for JavaScript

- How to detect **soundness** bugs that matter?
- How to isolate **precision** bottlenecks?

… in a large static analyzer for a complex language and a massive platform API



# Soundness and precision in static analysis



# Outline

- ***Soundness testing***
- ***Blended analysis***[1]
- ***Delta debugging***[2] (or, *cause reduction*)
- Combining the techniques

1) Dufour, Ryder, and Sevitsky, *Blended Analysis for Performance Understanding of Framework-Based Applications*, ISSTA'07
2) Zeller and Hildebrandt, *Simplifying and Isolating Failure-Inducing Input*, STE 2002

# Soundness testing

## ***Provably*** sound

- sound with respect to all concrete executions
- infeasible for an analyzer as complex as TAJS

vs.

## ***Probably*** sound

- sound with respect to a finite set of concrete executions
- very easy to test
- over 1 million soundness checks in TAJS's test suite

# Soundness testing – example

A JavaScript program:

```
1  var o = { p: 'foo,bar' };
2  var s = o.p;
3  var a = s.split(',');
```

A value log fr

```
Soundness testing failed for 1/5
 - CALLEE on pro
Soundness testing succeeded for 5/5 checks
f.js:2                                  type.split)
f.js:2                                  ined}
f.js:3:9 BASE STRING("foo,bar")
f.js:3:9 CALLEE BUILTIN(String.prototype.split)
f.js:3:9 ARGO STRING(",")
```

# Artificially increasing precision with blended analysis

Filter abstract values based on concrete values:

```
876  ...
877  eval(code); // code is unknown
878  ...
```

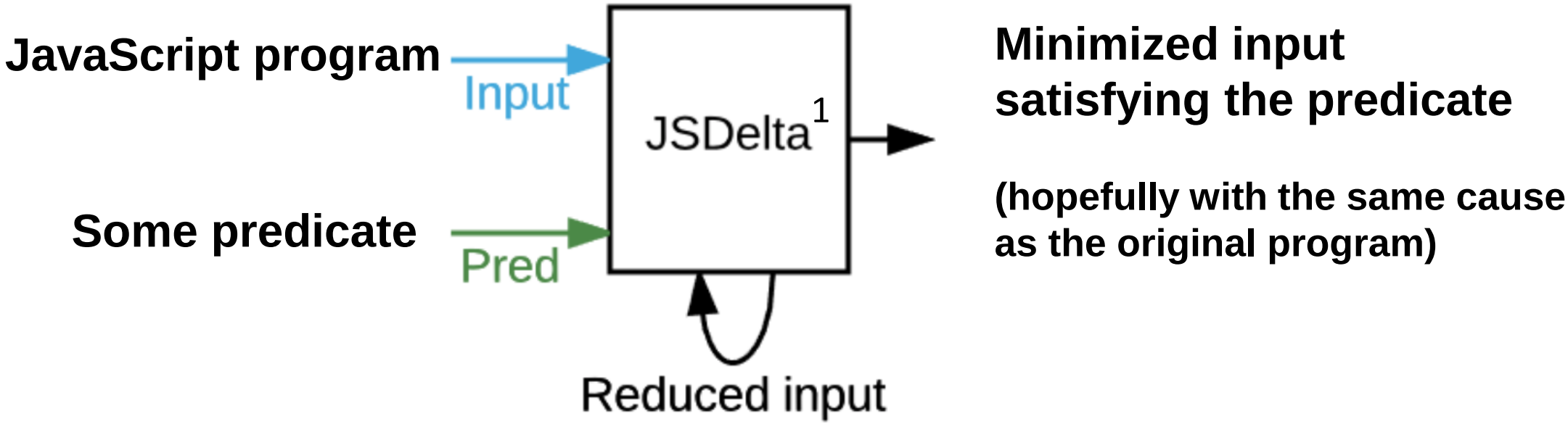
```
f.js:877:1 ARGO STRING("print('Same')")
```

Dufour, Ryder, and Sevitsky, *Blended Analysis for Performance Understanding of Framework-Based Applications*, ISSTA'07

# Delta debugging

Systematically minimizes input while preserving a target behavior

Typical inputs: Programs to analyze



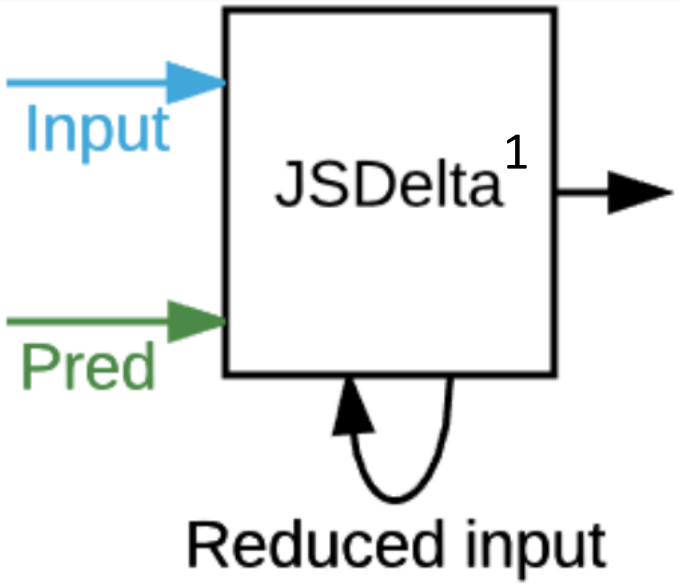

1) https://github.com/wala/jsdelta

# Delta debugging – precision example

Limitations:
- Generally only finds one problem at a time
- May introduce spurious behaviors

**underscore.js**
**1548 lines**

**Analysis times out after 3 minutes (“unanalyzable”)**




```
2  a = {p: 0, q: 0};
3  b = [];
4  for (var p in a)
5   b.push(p);
6  x = b[0]
7  a[x] = b[x];
8  a.p();
```

**8 lines!**

1) https://github.com/wala/jsdelta

# Combining the techniques

① Soundness testing + delta debugging

② Blended analysis + delta debugging

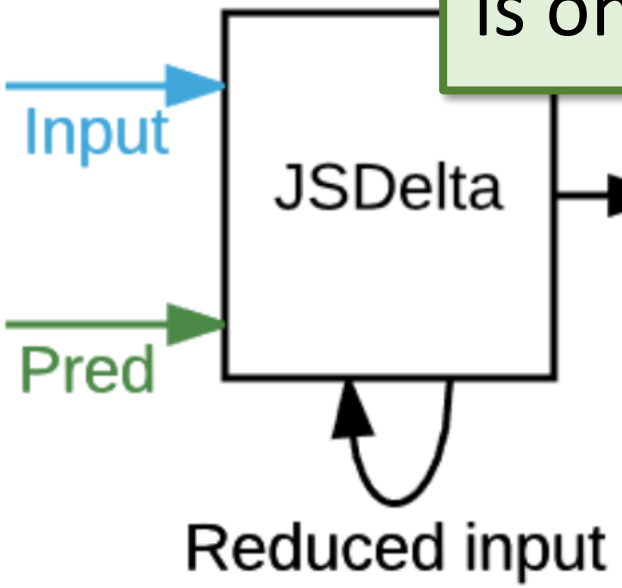
③ Soundness testing + blended analysis

④ Soundness testing + blended analysis + delta debugging

# ① Soundness testing + delta debugging

Goal: Isolate a soundness bug

Debugging is easier when there is only one soundness test failing




**Program to analyze**

**Analysis result is unsound**

**Minimized unsound program**

```
1  var i, s;
2  i = "0";
3  s = i++;
```

```
Soundness testing failed:
 - VAR 's' on program line 3:
  - concrete: NUMBER(0)
  - abstract: {STRING("0")}
```

# ② Blended analysis + delta debugging

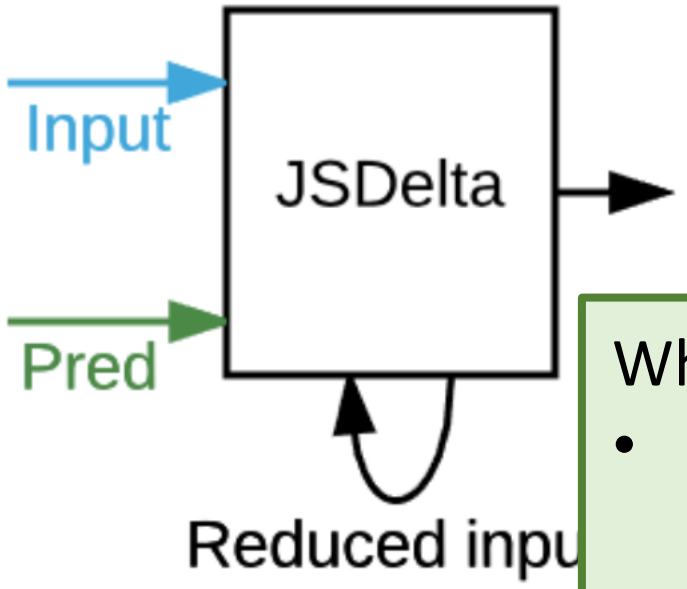
Goal: find precision bottlenecks

Idea: *bottlenecks are program locations that benefit from blended analysis*

– what is the minimal set of such locations?

**All locations in program to analyze**

**Program analyzable in 3 minutes?**




**Minimized set of locations satisfying predicate**

Why better than reducing the program?

- Finds *all* the critical locations in the program
- Avoids spurious behaviors introduced by delta debugger

# Example: finding precision bottlenecks

```
underscore-1.8.3.js needs more precision at:
- PROPERTY WRITE at line 1492
- CALL at line 1494
```

```
...
1490 _.mixin = function(obj) {
1491  _.each(_.functions(obj), function(name) {
1492   var func = _[name] = obj[name];
1493   _.prototype[name] = function() {
1494    func.apply(_, arguments);
1495   };});};
1496 _.mixin(_);
...
```

critical that `name` is not "any string"

critical that `func` is not *any* function from `obj`

Useful information for analysis designers!
Tells us where we need to improve the analysis abstractions

# ③ Soundness testing + blended analysis

Soundness testing is possible even with unanalyzable programs!

(where “unanalyzable” means “cannot be analyzed within 3 minutes”)

*Blended analysis does not affect the soundness tests when using the same concrete executions*

```
Soundness testing failed for 43/3932 check
 - PROP on program line 542:
  - concrete: BUILTIN(Symbol.unscopables)
  - abstract: {undefined}
```

Our model of `Symbol` was inadequate

# ④ Soundness testing + blended analysis + delta debugging

Automatically find a **minimal unsound program** from an **unanalyzable** program:

```
1  function f(){
2   return arguments;
3  }
4  f().p;
```

```
Soundness testing failed:
- PROP on program line 4:
   - concrete: UNDEFINED
   - abstract: {}
```

# Recommendations to static analysis developers

1. Implement a dynamic analysis to record **value logs** from concrete executions
2. Use **soundness testing** systematically
   – When soundness bugs are detected, use delta debugging
3. When critical precision problems appear, use **blended analysis**
   – Use delta debugging to find the critical program locations
4. Soundness bugs can be found, even with programs that are unanalyzable due to insufficient precision

# A workflow for static analysis developers