Static Analysis of JavaScript

Insights and Challenges

Ben Hardekopf

Department of Computer Science
University of California, Santa Barbara

Setting Expectations

What this talk is about

- Brief introduction to current state of JavaScript analysis
- Mostly from perspective of my research lab
  - Lessons we've learned
  - Challenges we've faced
- Some discussion of other groups attacking this problem

What this talk is *not* about

- Comprehensive overview of the entire field
- A tutorial on how exactly to analyze JavaScript

Talk Outline

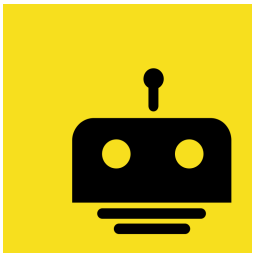
- **Motivation**
- **The JavaScript Language**
- **General Approaches to JavaScript Analysis**
- **The JSAI JavaScript Analyzer**
- **Some Lessons Learned**
- **The Challenges Ahead**

JavaScript is Everywhere









JavaScript is Hard to Get Right

News and Resources on Privacy

Home / How One Missing `var` Ruined our Launch

HOW ONE MISSING `VAR` RUINED OUR LAUNCH

October 31, 2011 · by Geoffrey Hayes · in Startups

Well, that was a veritable shitstorm (sorry for the language). Long story short, Safe Shepherd was featured today on TechCrunch (along with other 500Startups companies, also on VentureBeat, Forbes, …) and everything broke all at once. Every. little. thing. We had rolled out a huge change to MelonCard over the last few days to make our site a seamless "everything just updates" look-good / feel-good product using NodeJS long-polling with a slick KnockoutJS dynamic jQuery Templates front end. We did our due diligence of manual and unit testing, mixed with a full suite of Vows for Node. All systems check, full steam ahead, right? Not so fast.

Source: `http://blog.safeshepherd.com/23/how-one-missing-var-ruined-our-launch/`


JAVASCRIPT
PLEASE JUST WORK

We Need Better Tools for JavaScript

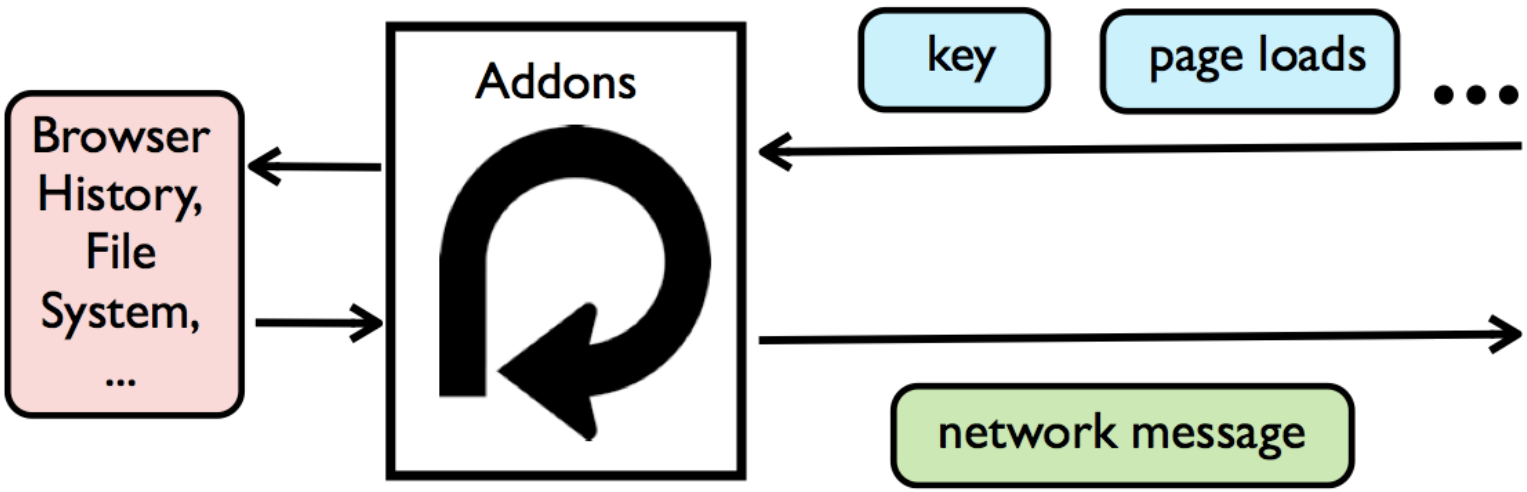
JavaScript program desiderata:

- Fast
- Correct
- Secure
- Maintainable

Static analysis to the rescue?

- Sound?
- Precise?
- Efficient?

Example: Browser Addon Security



- Written in JavaScript by 3rd-party developers
- Complete access to browser information
- No sandboxing or other security restrictions

- **Vulnerabilities** (e.g., arbitrary code execution)
- **Malware** (e.g., key loggers)
- **Proof-of-concept exploits** (e.g., FFSniff)

Kashyap et al, "*Security Signature Inference for JavaScript-based Browser Addons*" CGO 2014

Talk Outline

- Motivation
- **The JavaScript Language**
- General Approaches to JavaScript Analysis
- The JSAI JavaScript Analyzer
- Some Lessons Learned
- The Challenges Ahead

The JavaScript Language

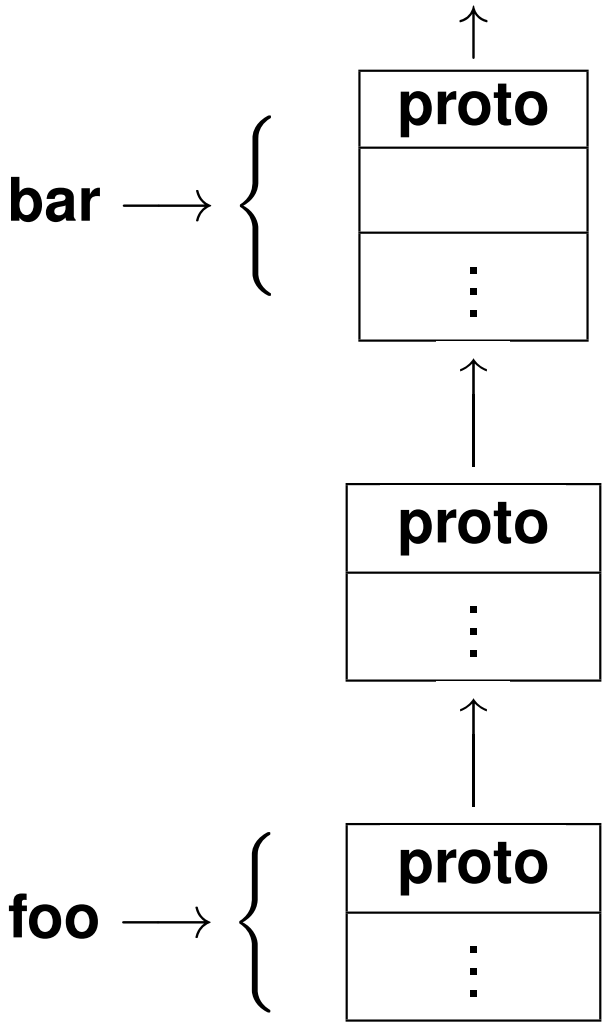
- **Imperative, dynamically-typed language**
  - Objects, prototype-based inheritance, closures, exceptions
- **Objects are the fundamental data structure**
  - Object properties can be dynamically inserted and deleted
  - Property accesses can be computed at runtime
  - Object introspection (runtime reflection)
  - Functions and arrays are just objects
- **Designed to be resilient**
  - Nonsensical actions (accessing a property of a non-object, adding two functions together, etc) are handled using implicit conversions and default behaviors
  - Lots of quirks and edge cases (`with`, `this`, `arguments`, `...`)

Prototype-Based Inheritance

(plus dynamic property computation and insertion.)

JavaScript source code:

```
bar.func = function() {···}
var name = "fu" + "nc"
foo[name]()
```

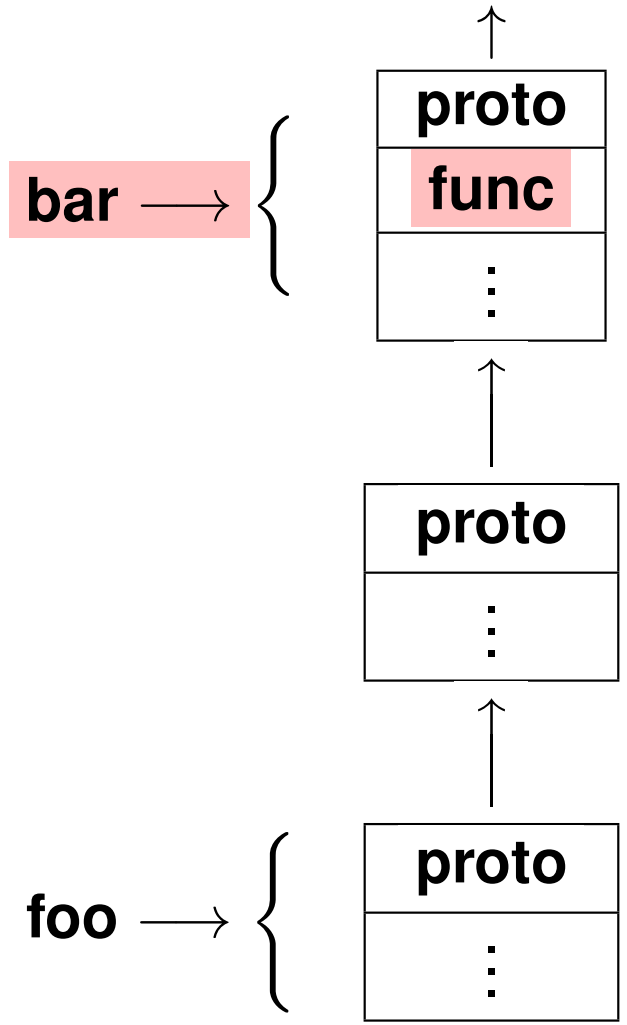


Prototype-Based Inheritance

(plus dynamic property computation and insertion.)

JavaScript source code:

```
bar.func = function() {···}
var name = "fu" + "nc"
foo[name]()
```

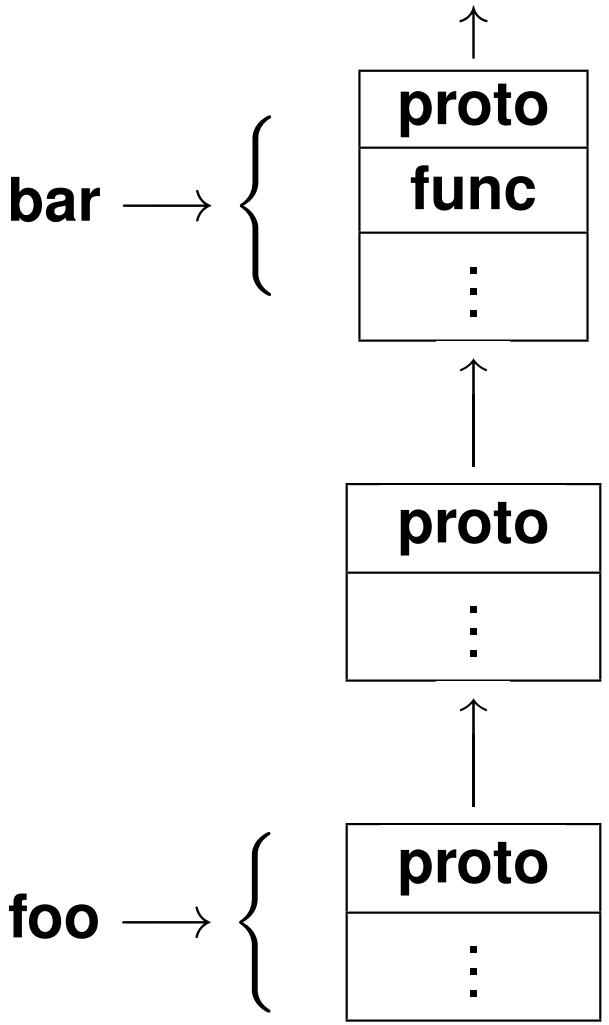


Prototype-Based Inheritance

(plus dynamic property computation and insertion.)

JavaScript source code:

```
bar.func = function() {···}
var name = "fu" + "nc"
foo[name]()
```

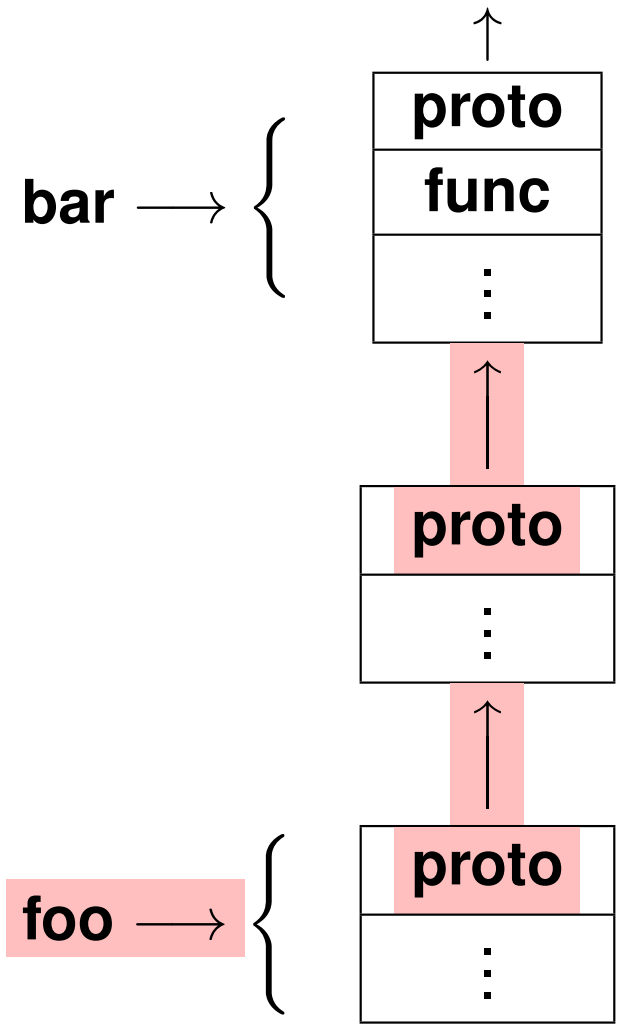


Prototype-Based Inheritance

(plus dynamic property computation and insertion.)

JavaScript source code:

```
bar.func = function() {···}
var name = "fu" + "nc"
foo[name]()
```



Implicit Conversion: `var x = myArray[idx]`

Implicit Conversion: `var x = myArray[idx]`

What happens in the interpreter:

```
if myArray is null or undefined then raise type-error
if myArray is primitive then obj = toObject(myArray)
else obj = myArray
if idx is primitive then property = toString(idx)
else if idx.toString is callable then
    tmp = idx.toString()
    if tmp is primitive then property = toString(tmp)
    else
      VAL:
      if idx.valueOf is callable then
        tmp = idx.valueOf()
        if tmp is primitive then property = toString(tmp)
        else raise type-error
      else raise type-error
else goto VAL
x = obj.property
```

Implicit Conversion: `var x = myArray[idx]`

What happens in the interpreter:

```
if myArray is null or undefined then raise type-error
if myArray is primitive then obj = toObject(myArray)
else obj = myArray
if idx is primitive then property = toString(idx)
else if idx.toString is callable then
    tmp = idx.toString()
    if tmp is primitive then property = toString(tmp)
    else
      VAL:
      if idx.valueOf is callable then
        tmp = idx.valueOf()
        if tmp is primitive then property = toString(tmp)
        else raise type-error
      else raise type-error
else goto VAL
x = obj.property
```

Talk Outline

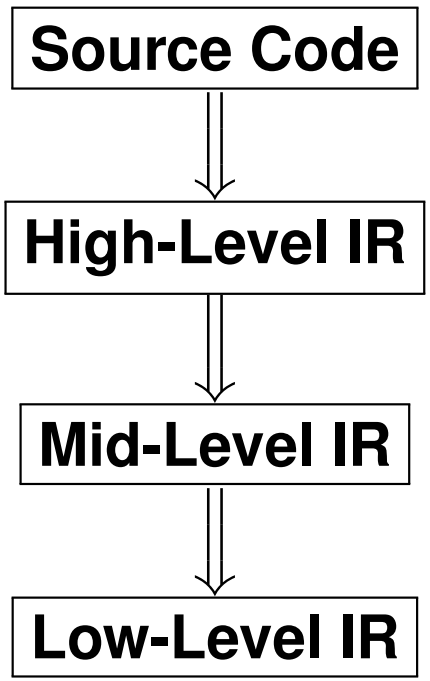
- Motivation
- The JavaScript Language
- **General Approaches to JavaScript Analysis**
- **The JSAI JavaScript Analyzer**
- **Some Lessons Learned**
- **The Challenges Ahead**

To be Sound or Not to be Sound

- It is hard to be simultaneously **sound**, **precise**, and **efficient**
  - This is always true, but for JavaScript achieving soundness is especially difficult
- Most JavaScript analyses give up on soundness, and for some domains this is perfectly OK, e.g., IDEs
  - Code completion *(Feldthaus et al, OOPSLA'13)*
  - Approximate callgraph construction *(Feldthaus et al, ICSE'13)*
- Other domains require soundness, e.g., security
  - Addon security vetting *(Kashyap et al, CGO'14)*
- In general, it's easier to start with soundness and remove features than to start with unsoundness and add features

Intermediate Representations

What level of IR should we analyze?



- **Lower-Level IR**
  - ***Pro:*** Simple, regular expressions; implicit operations made explicit
  - ***Con:*** Complex translation; results hard to map to the source code
- **Higher-Level IR**
  - ***Pro:*** Simple translation (if any); easy to map results to source code
  - ***Con:*** Complex, irregular expressions; implicit semantics

Type of Analysis

What analysis method should we use?

- ~~**Constraint-Based** (Flow-Insensitive)~~
  - Non-starter!
  - We need flow-sensitivity at a minimum
- **Dataflow Analysis** (CFG-based)
  - A popular choice, but (in my opinion) flawed
  - We need complex analysis to compute control-flow
- **State Reachability** (Abstract Interpretation-based)
  - Abstracting abstract machines *(Van Horn and Might, ICFP'10)*
  - Widening for control-flow *(Hardekopf et al, VMCAI'14)*

Other Research Groups

There are other research groups doing excellent work on JavaScript static analysis who have explored in different directions.

- Anders Møller's group, Aarhus University, Denmark
- WALA group, IBM T.J. Watson
- Sukyoung Ryu's group, KAIST, Korea
- And others...

Talk Outline

- Motivation
- The JavaScript Language
- General Approaches to JavaScript Analysis
- **The JSAI JavaScript Analyzer**
- **Some Lessons Learned**
- **The Challenges Ahead**

JSAI Challenges

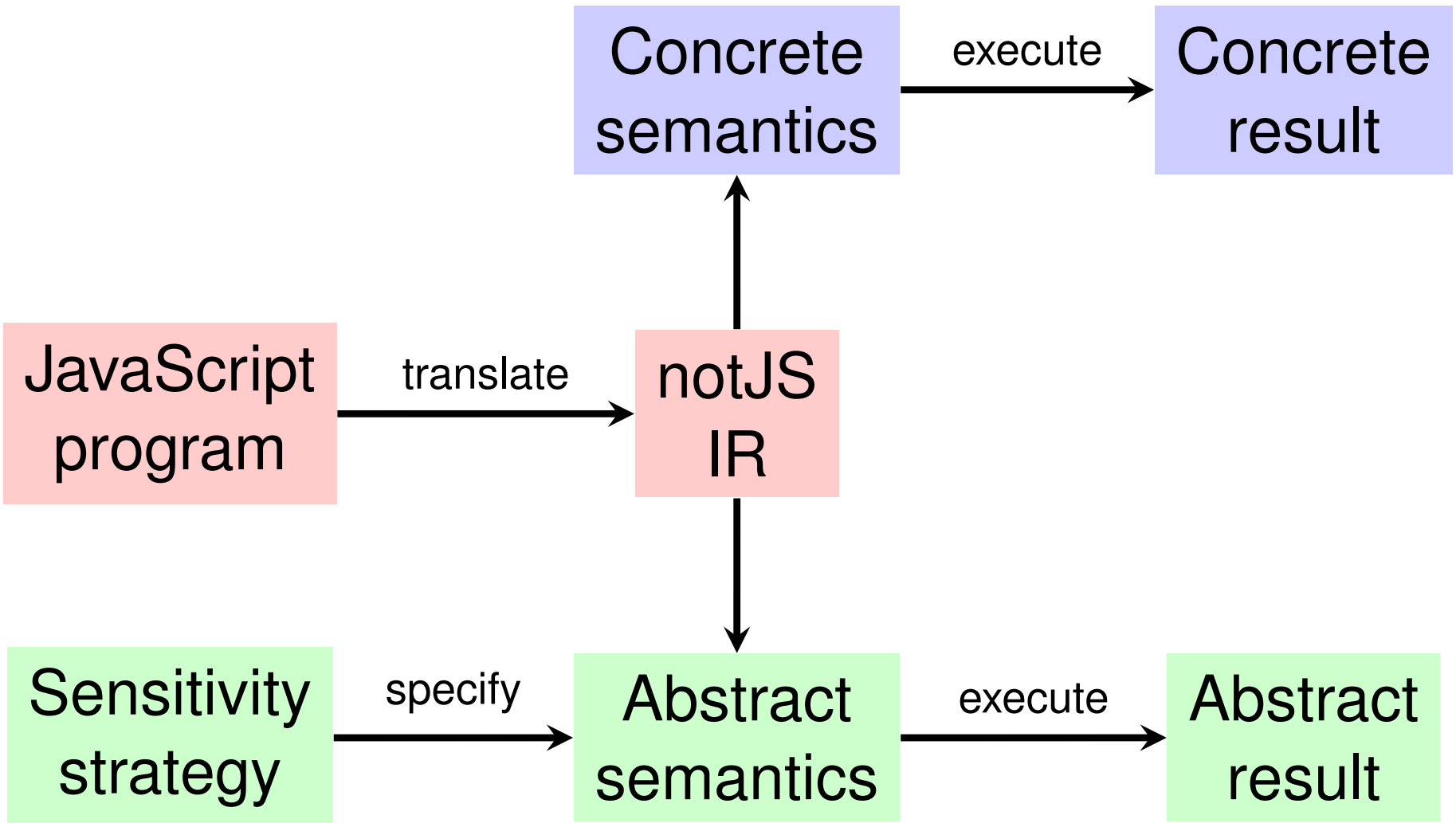
- **How to guarantee soundess?**
  - ✦ People have tried “best effort” and it doesn’t work
  - ✦ Need formalisms, abstract interpretation
- **How to define static analysis?**
  - ✦ Standard dataflow analysis doesn’t work (no CFG available)
  - ✦ Need different formulation of static analysis
- **What abstractions/sensitivities should be used?**
  - ✦ No one knows what abstractions and sensitivities work best
  - ✦ Need to easily experiment with different possibilities

JSAI Features

- **Features of JSAI**
  - First provably sound static analysis for JavaScript
  - Extensively tested against commercial JavaScript engines
  - Configurable control-flow sensitivity and abstract domains
  - Novel abstract domains for objects and strings

- **Publically available to research community**
  - Build client analyses
  - Experiment with abstract domains
  - Experiment with sensitivities

JSAI Architecture



The notJS IR

We designed the IR with static analysis in mind in order to make the analysis simpler, more efficient, and more precise.

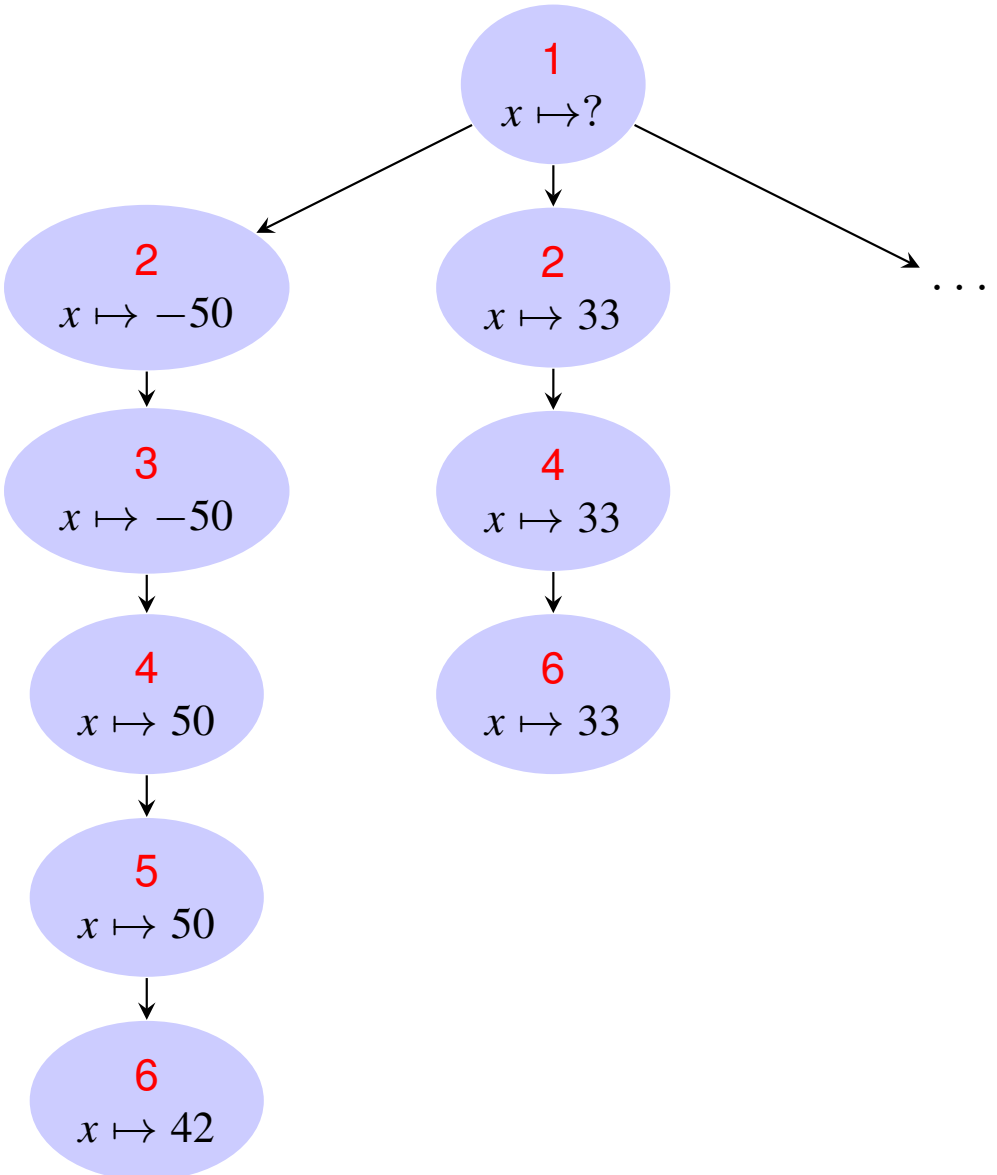
Selected IR Features

- Separate pure expressions from impure statements
- Translate implicit conversions into explicit operations
- Make the `this` and `arguments` parameters explicit
- And more...

Concrete Semantics

- **Concrete semantics specifies actual program behavior**
  - Define as a state transition system
  - Technically, an abstract machine smallstep operational semantics
- **Sound analysis $\Rightarrow$ formal semantics**
  - Forces us to precisely specify behavior
  - Amenable to proofs
- **Reality-check on our understanding of JavaScript behavior**
  - "Ground truth" for our static analysis
  - Heavily tested on over 1 million JavaScript programs, using Spidermonkey as a reference

Concrete State Transition System



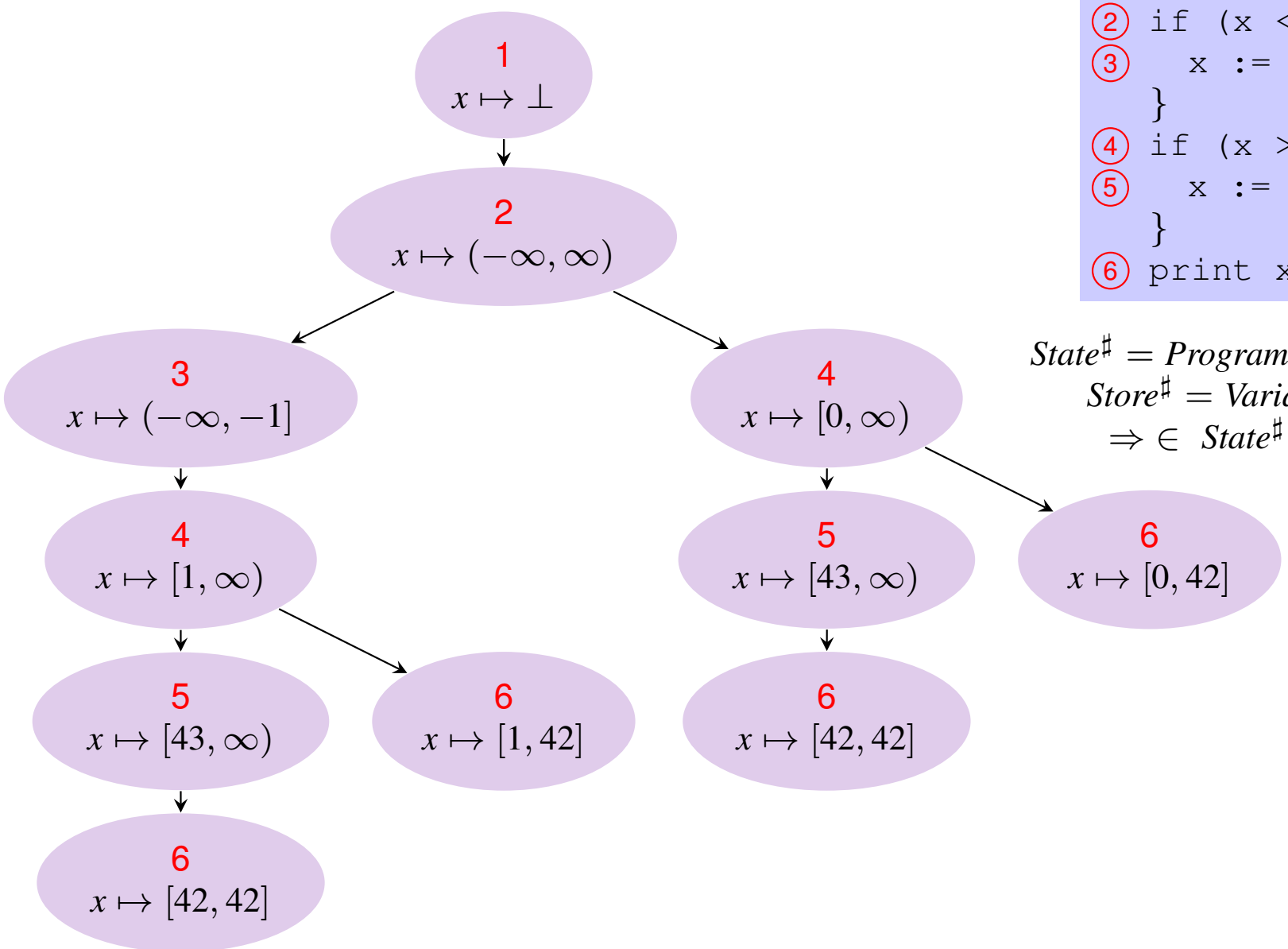

```
1 x := random_int()
2 if (x < 0) {
3   x := -x
  }
4 if (x > 42) {
5   x := 42
  }
6 print x
```

$$State = ProgramPoint \times Store$$
$$Store = Variable \to \mathbb{Z}$$
$$\Rightarrow \in State \times State$$

Abstract Semantics

- **Also in the form of a state transition system**
  - Think of an abstract state as representing a (potentially infinite) set of possible concrete states
  - There is no control-flow graph; the analysis computes the set of reachable abstract states using the state transition system
- **Specifies the actual static analysis**
  - Combines type inference, pointer analysis, control-flow analysis, string analysis, and boolean and number constant propagation
  - Novel abstract domains to represent objects and strings
- **Sound wrt the concrete semantics**

Abstract State Transition System



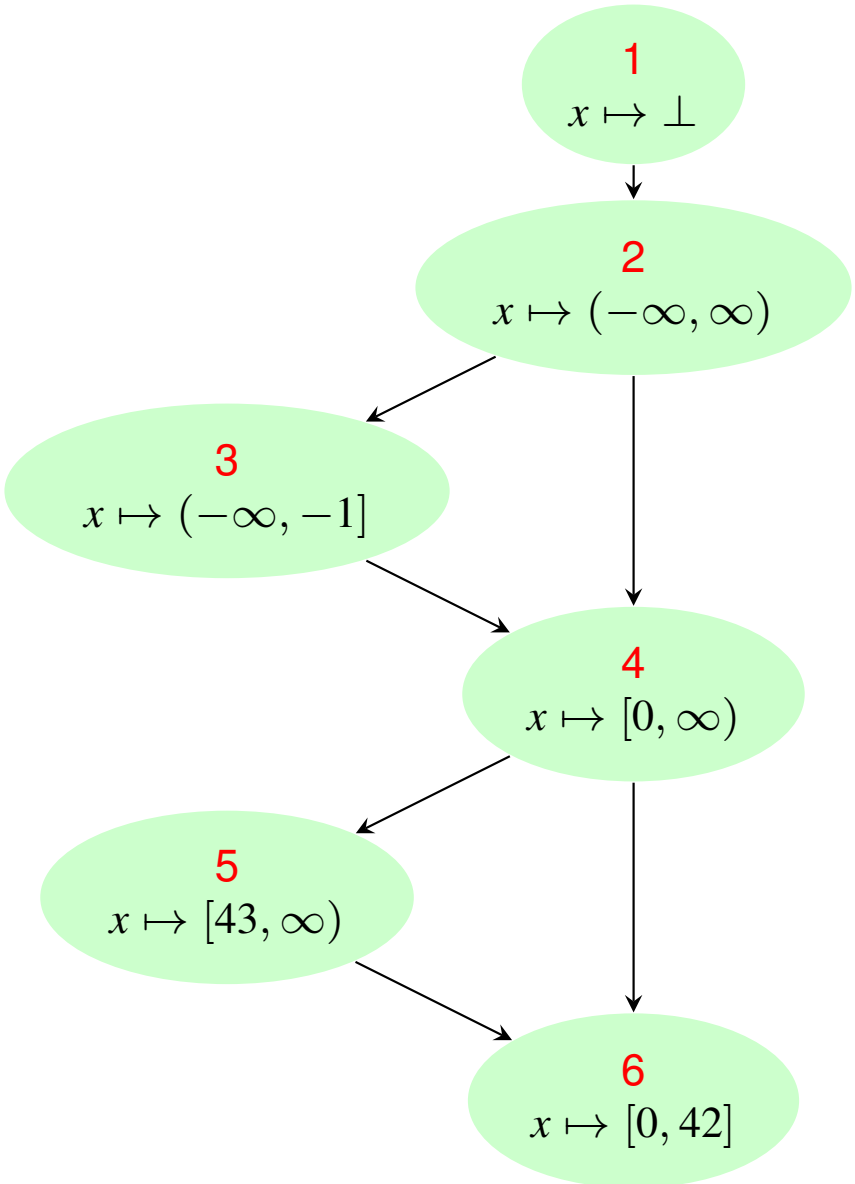

```
1 x := random_int()
2 if (x < 0) {
3   x := -x
  }
4 if (x > 42) {
5   x := 42
  }
6 print x
```

$$State^\sharp = ProgramPoint \times Store^\sharp$$
$$Store^\sharp = Variable \to \mathbb{Z}^\sharp$$
$$\Rightarrow \in State^\sharp \times State^\sharp$$

Configurable Sensitivity

- The previous abstract semantics is **exponential** in the number of nondeterministic transitions
- **Control-flow sensitivity** bounds the state space
  - Flow-sensitivity, context-sensitivity, path-sensitivity
  - Enables trade-offs between **precision** versus **performance**
  - An analysis usually bakes in a specific sensitivity
- **Theoretical insight:** Completely separate sensitivity strategy from abstract semantics
  - Define and implement abstract semantics independently from the sensitivity strategy
  - Plug in sensitivity strategies *a posteriori*, modularly tuning the analysis sensitivity

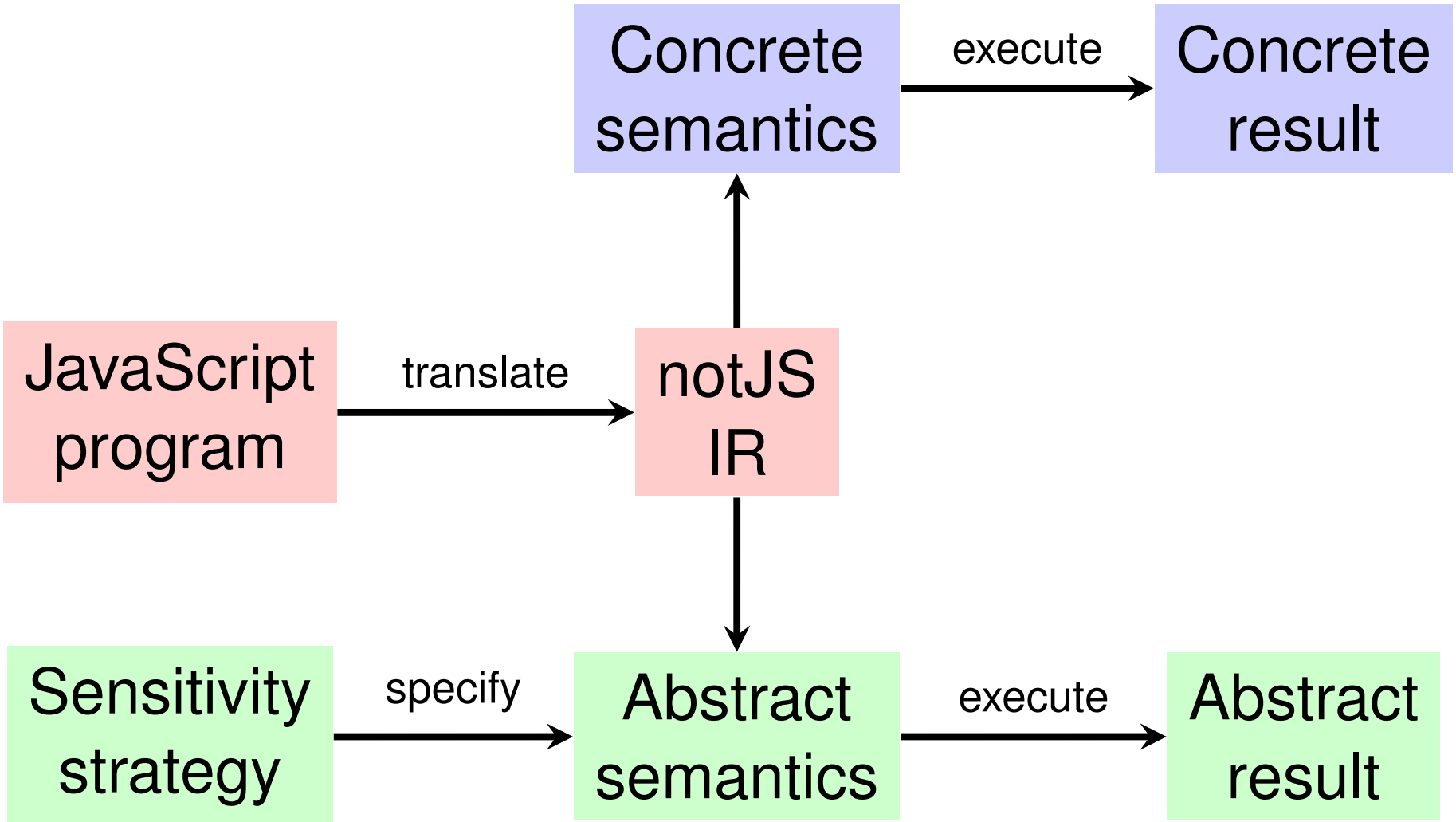
Widened Abstract State Transition System




```
1 x := random_int()
2 if (x < 0) {
3   x := -x
  }
4 if (x > 42) {
5   x := 42
  }
6 print x
```

$$State^\sharp = ProgramPoint \times Store^\sharp$$
$$Store^\sharp = Variable \to \mathbb{Z}^\sharp$$
$$\Rightarrow \in State^\sharp \times State^\sharp$$

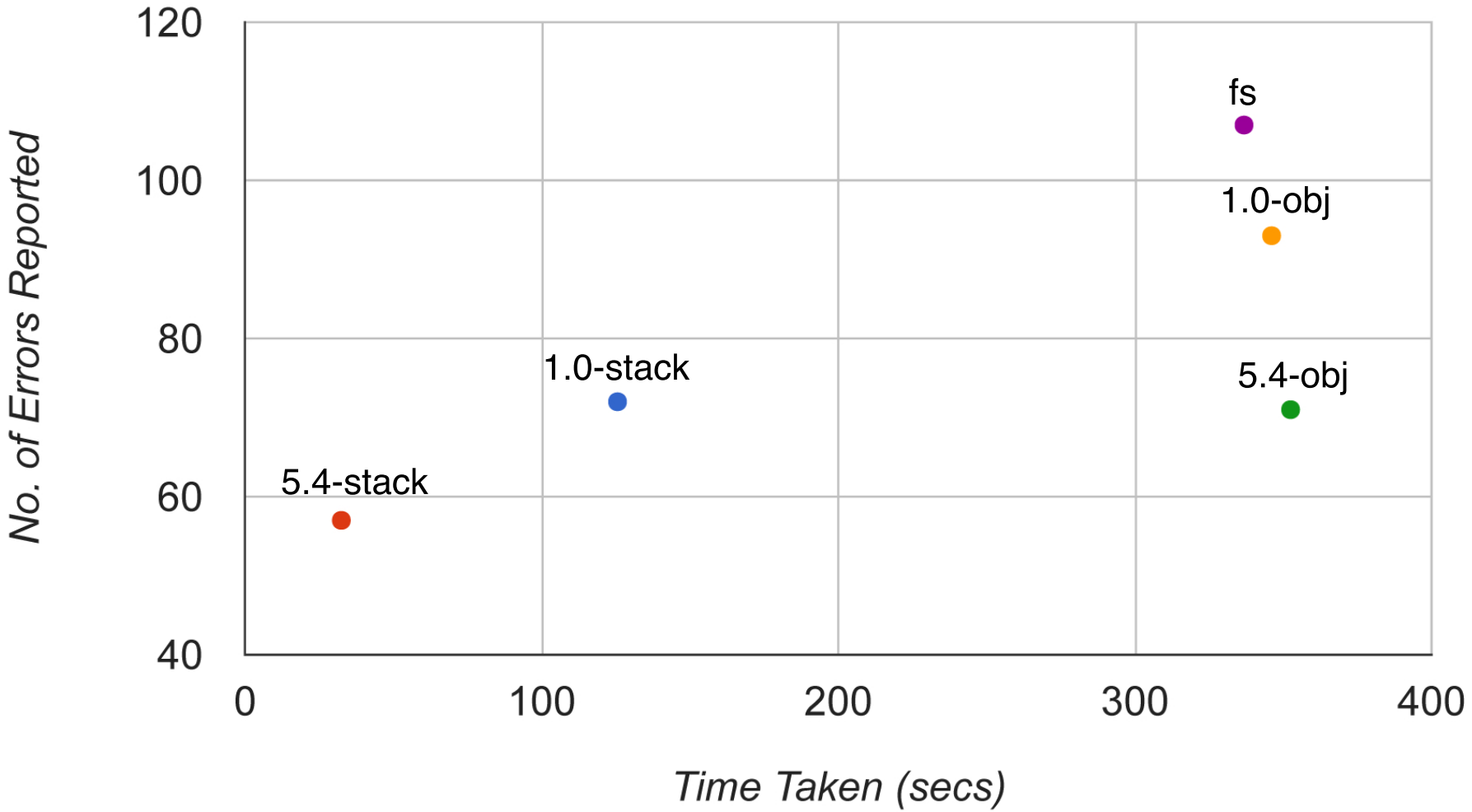
JSAI Architecture Review



JSAI Evaluation

- We evaluate JSAI for both **performance** and **precision**
  - We use a type-error analysis to measure relative precision: because the analysis is sound, fewer potential type errors means more precise
- **28 benchmarks, 4 different categories** of JavaScript programs
  (prior work mostly used just the first category)
  - Standard benchmarks (Sunspider, Octane)
  - Browser addons
  - Real-world open-source programs from Github
  - Generated JavaScript via Emscripten
- **Tested 56 different sensitivities**
  - Largest such study ever done, due to our configurable sensitivity

Selected Results



Talk Outline

- Motivation
- The JavaScript Language
- General Approaches to JavaScript Analysis
- The JSAI JavaScript Analyzer
- **Some Lessons Learned**
- **The Challenges Ahead**

Reduced Product is Essential

Computing control-flow and data-flow requires:

- Type inference
- Pointer analysis
- Control-flow analysis
- String analysis
- Number analysis
- Boolean constant propagation

All of these need to work together in carefully designed harmony in order to get useful results.

String and **Object** Abstract Domains Very Important

- **Object classes.** Objects come from different pre-defined classes, e.g., *Array*, *Function*, *Number*, etc. An object's class affects its semantics.
  - ***Example:*** assignment to `length` property for Array vs non-Array
- **Property names.** The names of properties are just strings; looking up an unknown string as a property can lose tremendous amounts of precision.
  - ***Example:*** Prototype-based inheritance means that the results merge all properties of all objects in the prototype chain

Type Refinement

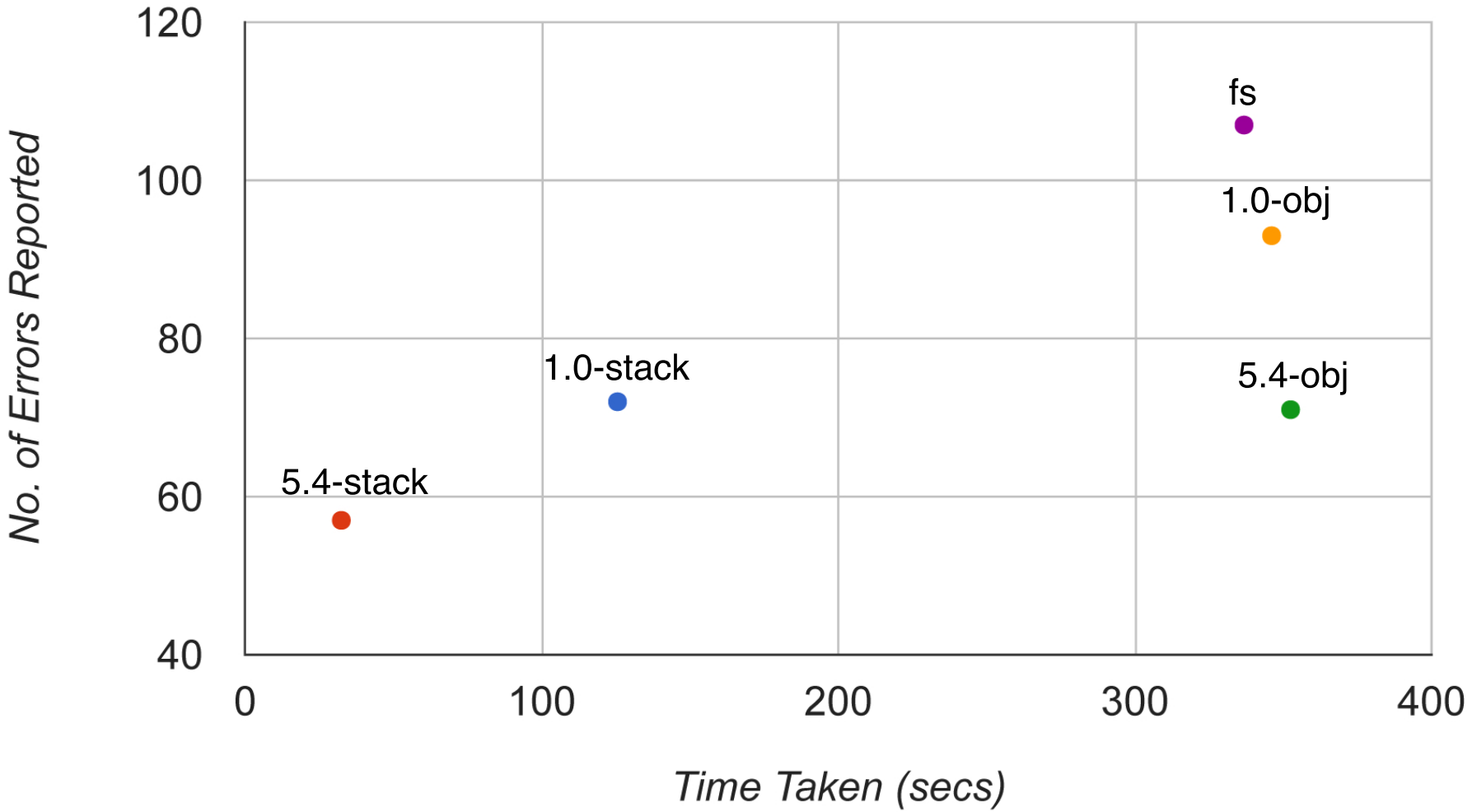
Old idea: refine abstract values based on branch conditions.

- Often ignored in dataflow analysis
- Especially important for JavaScript
- Most important refinements are to type information
- Most important branches are implicit in the semantics
- A low-level IR helps tremendously

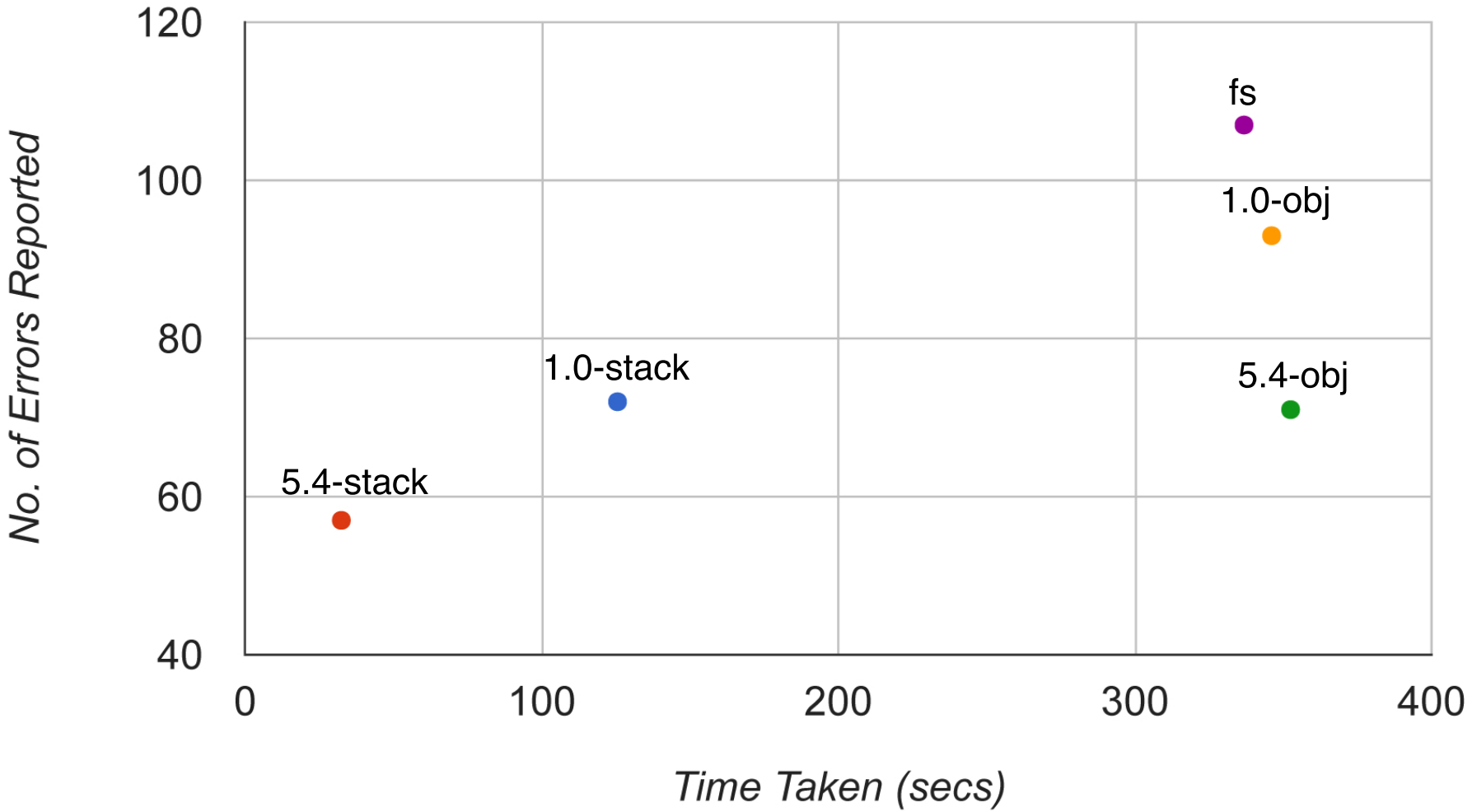
We tried this with JSAI for a type-error client analysis.

- Average 53% reduction in reported type errors
- Maximum 86% reduction in reported type errors

Higher Precision ⊃ Better Performance



Context-Sensitivity: Callstring > Object



State Reachability $\supset$ Good Parallelization

The state reachability method for static analysis turns out to be amenable to parallelization:

- State reachability is embarassingly parallel
- Merging states for sensitivity adds synchronization points
- Different sensitivities tradeoff parallelism for reduced state space

We tried this for JSAI.

- 2–4$\times$ speedup on average, 36$\times$ maximum
- We think it could do even better with more work

Talk Outline

- Motivation
- The JavaScript Language
- General Approaches to JavaScript Analysis
- The JSAI JavaScript Analyzer
- Some Lessons Learned
- **The Challenges Ahead**

Sensitivities and Abstractions

What are the right ones to use?

- Better abstractions for strings and objects
- Better sensitivities for precision and performance

Need to explore more sensitivities to find the sweet-spot (JavaScript's equivalent of object-sensitivity for Java).

JSAI's configurable sensitivity helps make this feasible.

Performance

We can handle 1,000s–10,000s LOC, but we need to handle 100,000s.

- **Parallelism.** We've made a good start, but need more
- **Sparseness.** Traditional SSA won't cut it; what can we do?
  - Complex dependencies, need to consider branch conditions
  - Some progress. (Jensen et al, SAS 2010; Madsen et al, 2014)
  - Not a solved problem
- **Other ideas?**

Frameworks

JavaScript frameworks are extremely useful and popular, e.g., JQuery.

- Some of the hairiest JavaScript code you'll ever see
- Very difficult to get precision and performance
- One of the biggest open problems in JavaScript analysis
- Some progress, but much remains (Shäfer et al, PLDI'13; Andreasen et al, OOPSLA'14)

Dynamic Code Injection

Handling `eval` and family.

Dynamic code injection is the bane of static analysis. What can we do?

- Some application domains don't use `eval`
  - Browser addons
  - Machine-generated JavaScript
- Sometimes we can eliminate `eval` from the program
  - Unnecessary uses of `eval` when other techniques will work
- **What about when we do have to deal with `eval`?**
  - Assume and enforce?
  - Dynamically patch analysis?
  - Other ideas?

Dialects of JavaScript

Different JavaScript engines have effectively their own dialects.

- JavaScript engine implementors sometimes consider the ECMA language specification more of a “suggestion”
  - Mozilla SpiderMonkey allows assignment to object prototype fields
- Different engines refine underspecified behavior in different ways
  - V8 vs SpiderMonkey: different iteration orders for `for..in` loops
- Production engines are used to proselytize potential future language extensions
  - Mozilla SpiderMonkey: object proxies, typed arrays

Different Environments

JavaScript is used in different settings which require static analyses to model different external environments.

- **Web pages:** DOM
- **Addons:** XPCOM
- **Server:** Node.js API

This is a major problem and concern for JavaScript analysis infrastructures.

The End

Questions?