Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries

Authored by **M Madsen, B Livshits, M Fanning**
Aarhus University & Microsoft Research

ESEC / FSE 2013

Presented by **Jing Pu**

Outline

- Motivation
- Challenges
- Approach & Key Techniques
- Evaluation
- Conclusion

Motivation

- Research target

  JavaScript applications execute in a rich execution environment

  - web programs
  - Server-side programs

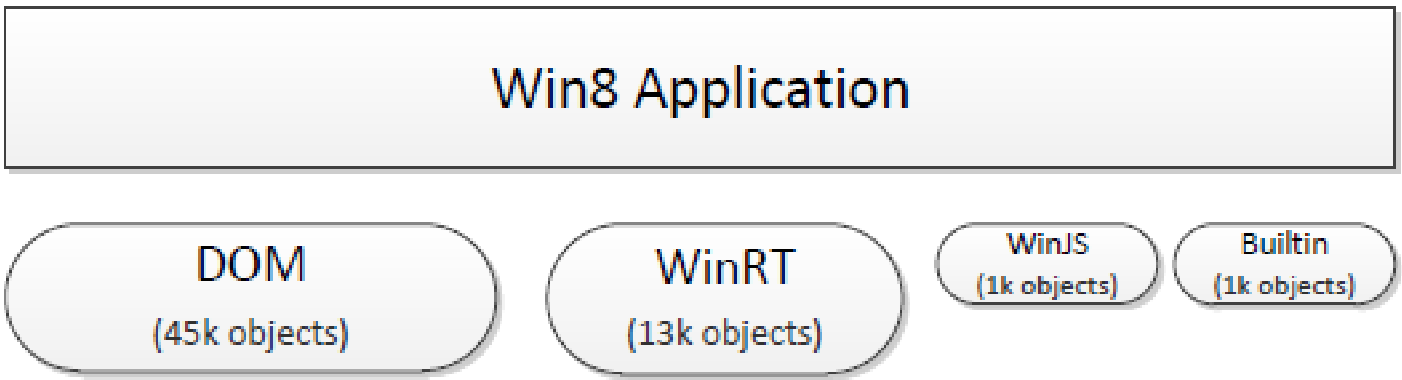
- Problem

  Library and OS invocation codes are ignored and not well analyzed.

- How to in-depth statically analyze:

  JavaScript applications in the windows 8 OS ?

Win 8 JavaScript applications



- This is the composition of a typical Windows 8 JavaScript application.
- Large size of library objects.
- Depends on libraries communicating with HTML DOM
- Uses Windows Runtime libraries
- Used built-in DOM API and other popular libraries and frameworks.

Challenges

- Rely on environment libraries
  - Browser API
  - HTML DOM
  - Invoke OS libraries at Windows runtime
- Popular libraries reflective JavaScript features
  - Reflective calls
  - Eval
  - Computed properties
  - Runtime modification of properties
- Reason about the objects information return from libraries & pass into callbacks

Approach & Key Techniques

- Approach

  Infer what the **objects** are based on **observing uses of library functionality** within application code.

- Key Techniques
  - Pointer analysis
  - Use analysis

Examples

- Example 1: DOM-manipulating code snippet

```
var canvas = document.querySelector("#leftcol .logo");
var context = canvas.getContext("2d");
context.fillRect(20, 20, c.width / 2, c.height / 2);
context.strokeRect(0, 0, c.width, c. height);
```

Q: What **object** does **querySelector** return ?

A: HTMLCanvasElement

M: Use pointer analysis & use analysis

Examples

- Example 2: Stubs from WinRT library

```
Windows.Storage.Stream.FileOutputStream = function() {};
Windows.Storage.Strean.FileOutputStream.prototype = {
    writeAsync = function() {},
    flushAsync = function() {},
    close = function() {}
}
```

```
var s = Windows.Storage.Stream;
var fs = new s.FileOutputStream(...)
fs.writeAsync(...).then(function() {
  ...
});
```

Stubs

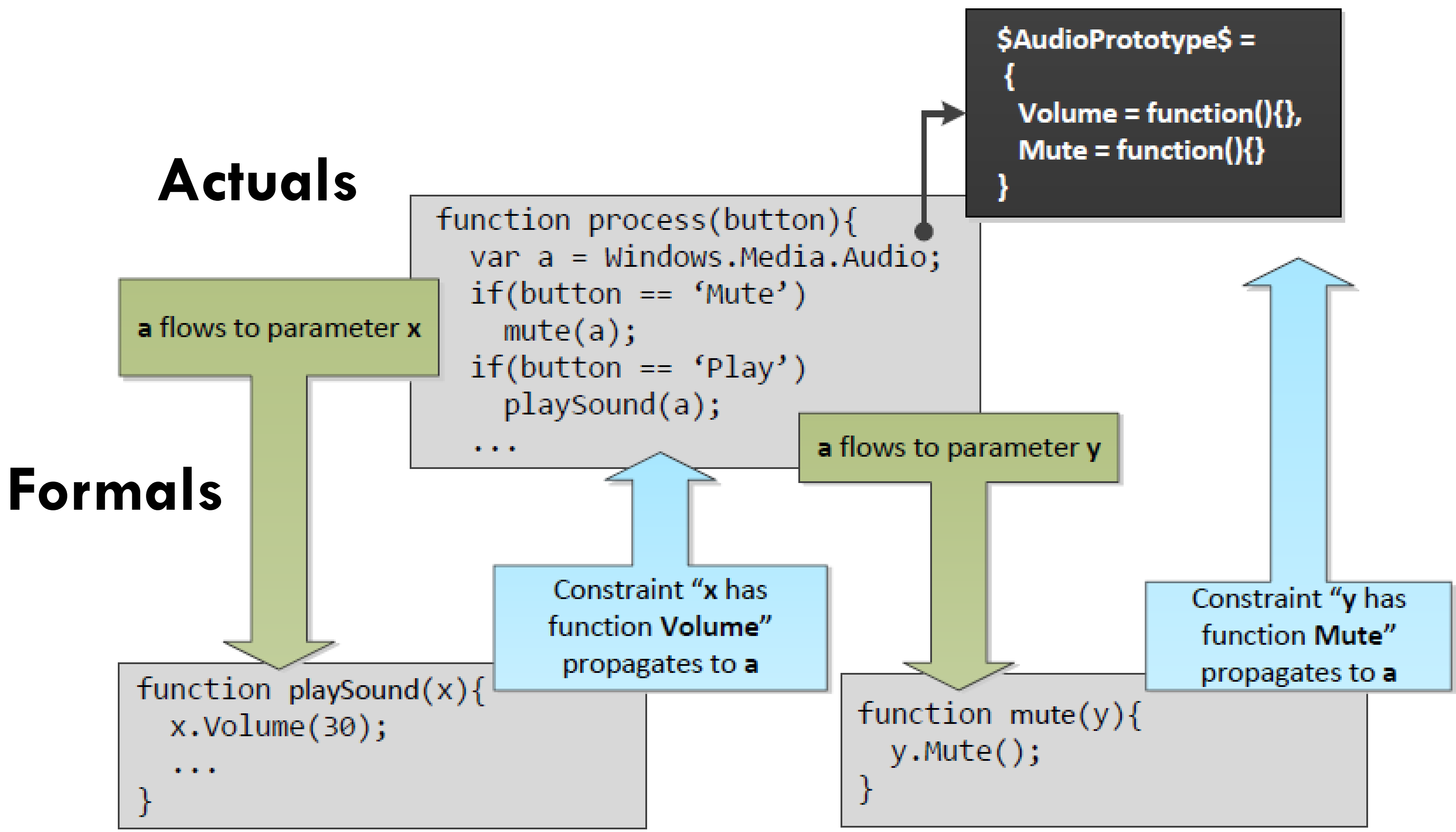
Application

Q: What **object** does **writeAsync** return ?

A: Promise[Proto]

Examples

- Example 3: Pointer analysis & use analysis



Pointer analysis

- Uses Datalog declaration and analysis rules
- Accepts an input program represented as a set of relations. Domains:
  - **H**: Heap-allocated objects and functions
  - **V**: Program variables
  - **C**: Call sites
  - **P**: Properties
  - **Z**: Integers
- Generate output relations representing the analysis result, e.g.
  - Points-to relation
  - Call graph construction
  - Prototype chain relation

Pointer analysis – inference rules

$\textsc{PointsTo}(v, h)$	:–	$\textsc{NewObj}(v, h, \_).$
$\textsc{PointsTo}(v_1, h)$	:–	$\textsc{Assign}(v_1, v_2), \textsc{PointsTo}(v_2, h).$
$\textsc{PointsTo}(v_2, h_2)$	:–	$\textsc{Load}(v_2, v_1, p), \textsc{PointsTo}(v_1, h_1), \textsc{HeapPtsTo}(h_1, p, h_2).$
$\textsc{HeapPtsTo}(h_1, p, h_2)$	:–	$\textsc{Store}(v_1, p, v_2), \textsc{PointsTo}(v_1, h_2), \textsc{PointsTo}(v_2, h_2).$
$\textsc{HeapPtsTo}(h_1, p, h_3)$	:–	$\textsc{Prototype}(h_1, h_2), \textsc{HeapPtsTo}(h_2, p, h_3).$
$\textsc{Prototype}(h_1, h_2)$	:–	$\textsc{NewObj}(\_, h_1, v), \textsc{PointsTo}(v, f), \textsc{HeapPtsTo}(f, \texttt{"prototype"}, h_3).$
$\textsc{CallGraph}(c, f)$	:–	$\textsc{ActualArg}(c, 0, v), \textsc{PointsTo}(v, f).$
$\textsc{Assign}(v_1, v_2)$	:–	$\textsc{CallGraph}(c, f), \textsc{FormalArg}(f, i, v_1), \textsc{ActualArg}(c, i, v_2), z > 0.$
$\textsc{Assign}(v_2, v_1)$	:–	$\textsc{CallGraph}(c, f), \textsc{FormalRet}(f, v_1), \textsc{ActualRet}(c, v_2).$

Output relations ← **Input relations**

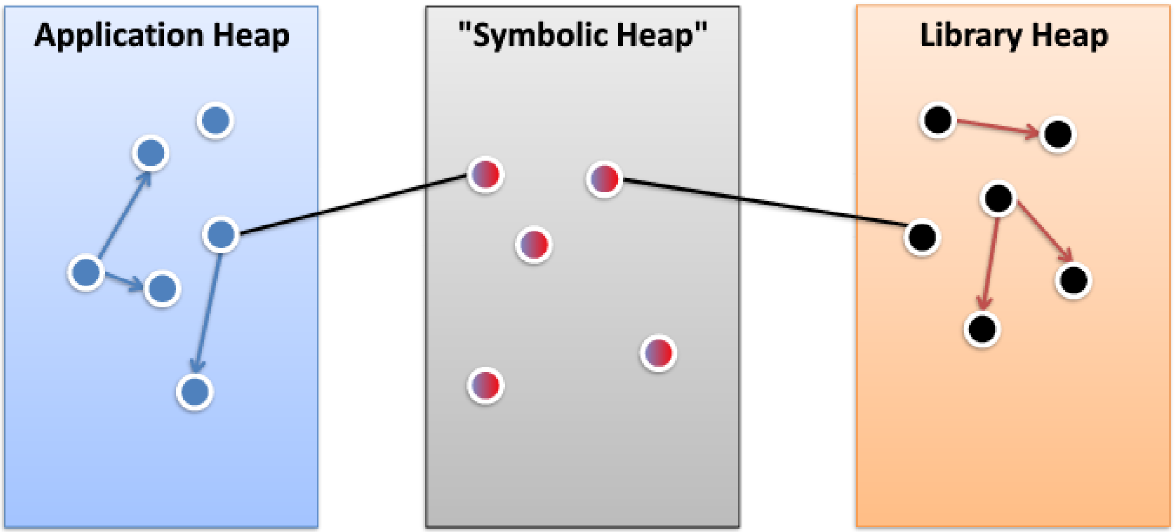
Use analysis – Inference rules

$$
\begin{array}{lll}
\textsc{ResolvedVariable}(v) & :- & \textsc{PointsTo}(v, \_). \\
\textsc{PrototypeObj}(h) & :- & \textsc{Prototype}(\_, h). \\
\\
\textsc{DeadArgument}(f, i) & :- & \textsc{FormalArg}(f, i, v), \neg\textsc{ResolvedVariable}(v), \textsc{AppAlloc}(f), i > 1. \\
\textsc{DeadReturn}(c, v_2) & :- & \textsc{ActualArg}(c, 0, v_1), \textsc{PointsTo}(v_1, f), \textsc{ActualRet}(c, v_2), \\
 & & \neg\textsc{ResolvedVariable}(v_2), \neg\textsc{AppAlloc}(f). \\
\\
\textsc{DeadLoad}(h, p) & :- & \textsc{Load}(v_1, v_2, p), \textsc{PointsTo}(v_2, h), \neg\textsc{HasProperty}(h, p), \textsc{AppVar}(v_1), \textsc{AppVar}(v_2). \\
\textsc{DeadLoad}(h_2, p) & :- & \textsc{Load}(v_1, v_2, p), \textsc{PointsTo}(v_2, h_1), \textsc{Prototype}(h_1, h_2), \\
 & & \neg\textsc{HasProperty}(h_2, p), \textsc{Symbolic}(h_2), \textsc{AppVar}(v_1), \textsc{AppVar}(v_2). \\
\textsc{DeadLoadDynamic}(v_1, h) & :- & \textsc{LoadDynamic}(v_1, v_2), \textsc{PointsTo}(v_2, h), \neg\textsc{ResolvedVariable}(v_1), \\
 & & \textsc{AppVar}(v_1), \textsc{AppVar}(v_2). \\
\\
\textsc{DeadPrototype}(h_1) & :- & \textsc{NewObj}(\_, h, v), \textsc{PointsTo}(v, f), \textsc{Symbolic}(f), \neg\textsc{HasSymbolicPrototype}(h). \\
\\
\textsc{CandidateObject}(h_1, h_2) & :- & \textsc{DeadLoad}(h_1, p), \textsc{HasProperty}(h_2, p), \textsc{Symbolic}(h_1), \neg\textsc{Symbolic}(h_2), \\
 & & \neg\textsc{HasDynamicProps}(h_1), \neg\textsc{HasDynamicProps}(h_2), \neg\textsc{SpecialProperty}(p). \\
\textsc{CandidateProto}(h_1, h_2) & :- & \textsc{DeadLoad}(h_1, p), \textsc{HasProperty}(h_2, p), \textsc{Symbolic}(h_1), \neg\textsc{Symbolic}(h_2), \\
 & & \neg\textsc{HasDynamicProps}(h_1), \neg\textsc{HasDynamicProps}(h_2), \textsc{PrototypeObj}(h_2). \\
\\
\textsc{NoLocalMatch}(h_1, h_2) & :- & \textsc{Prototype}(h_2, h_3), \\
 & & \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_2, p), \\
 & & \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_3, p), \\
 & & \textsc{CandidateProto}(h_1, h_2), \textsc{CandidateProto}(h_1, h_3), h_2 \neq h_3. \\
\\
\textsc{UnifyProto}(h_1, h_2) & :- & \neg\textsc{NoLocalMatch}(h_1, h_2), \textsc{CandidateProto}(h_1, h_2), \\
 & & \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_2, p). \\
\textsc{FoundPrototypeMatch}(h) & :- & \textsc{UnifyProto}(h, \_). \\
\textsc{UnifyObject}(h_1, h_2) & :- & \textsc{CandidateObject}(h_1, h_2), \neg\textsc{FoundPrototypeMatch}(h_1) \\
 & & \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_2, p).
\end{array}
$$

(b) Use analysis inference.

- **Generate symbolic facts based on the facts and constraints after pointer analysis**
- **Recover missing flow(arguments, return values and loads) due to missing implementations of libraries.**

Use analysis – Heap Partitioning



- **Abstract locations** are used as approximation of runtime object allocation in the program.
- Distinguish abstract locations in between H_A, H_L and H_S
- **Symbolic locations** are introduced for reasoning about abstract locations returned by library calls **where flow is dead due to libraries**
- **Reference: "Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries" ppt**

Use analysis – Unification

- Dead flow scenarios
  - Dead Returns / Dead Arguments / Dead loads
- Why
  - Variables within **V** domain normally have points-to links to heap elements in **H**
  - Ignore library code and use of stubs
  - Missing interprocedural flow in the presence of libraries
- Solution
  - Unification strategies

Unification strategies

- Three unification strategies
  - Matching of at least one property

    Too many objects get linked
  - Matching of all properties

    Too few objects get linked

    – Unsoundness & Imprecision
  - Prototype-based unification
    1. Disallow commonly-used properties (e.g. prototype, length) for unification
    2. Consider most precise object in the prototype hierarchy to unify first

    Best – improve precision

Example – Prototype-based unification

```
var firstName     = "Lucky";
var lastName      = "Luke";
var favoriteHorse = "Jolly Jumper";
function compareIgnoreCase(s1, s2) {
  return s1.toLowerCase() < s2.toLowerCase();
}
```

- Function **compareIgnoreCase** is defined in application and is used as callback passed into library.
- Return arguments **s1** and **s2** have **toLowerCase** property
- However, all string constants have this property, should not unify all of them
- Consider prototype object: String[Proto]

Inference Algorithm

- Iterative Inference Algorithm
  - Collects and records occurrence of dead returns/arguments/loads
  - Introduces symbol location for each location
  - Perform unification: unifying symbolic objects with appropriate application or library abstract locations
  - Terminates when no more dead flows can be founded and no more unification can be performed

Use analysis – Other techniques

- Extend Partial Inference to Full Inference
  - Do not assume existence of stubs, fully depends on uses found in applications
  - Allow symbolic location to point to another symbolic location to resolve limited dead loads

  – While in partial inference, symbolic location is only allowed to link to a non-symbolic location

- Namespace Mechanisms
  - Solve the points-to problem of global variable
  - Solve missing prototype problems caused by JavaScript calls to library constructors created by namespace mechanisms.
  - Introduce a special symbolic prototype object to deal with this case

- Array Access and Dynamic Properties
  - Introduce a single symbolic object and inject it into array variables for unification analysis.

Evaluation

- Experimental Setup
  - Evaluate both partial and full inference algorithms
  - Evaluation Tool –

    Front end: C#, parses JavaScript application and generates input facts for analysis;

    Back end: Z3 Datalog engine
  - Machine: Windows 7 machine, Xeon 64-bit 4-core CPUT, 3.07 GHz with 6 GB of RAM
- Results

Benchmarks

- A set of 25 JavaScript applications

Lines	Functions	Alloc. sites	Call sites	Properties	Variables
245	11	128	113	231	470
345	74	606	345	298	1,749
402	27	236	137	298	769
434	51	282	194	336	1,007
488	53	369	216	303	1,102
627	59	341	239	353	1,230
647	36	634	175	477	1,333
711	315	1,806	827	670	5,038
735	66	457	242	363	1,567
807	70	467	287	354	1,600
827	33	357	149	315	1,370
843	63	532	268	390	1,704
1,010	138	945	614	451	3,223
1,079	84	989	722	396	2,873
1,088	64	716	266	446	2,394
1,106	119	793	424	413	2,482
1,856	137	991	563	490	3,347
2,141	209	2,238	1,354	428	6,839
2,351	192	1,537	801	525	4,412
2,524	228	1,712	1,203	552	5,321
3,159	161	2,335	799	641	7,326
3,189	244	2,333	939	534	6,297
3,243	108	1,654	740	515	4,517
3,638	305	2,529	1,153	537	7,139
6,169	506	3,682	2,994	725	12,667
1,587	**134**	**1,147**	**631**	**442**	**3,511**

Fig. 9: Benchmarks, sorted by lines of code.

Name	Lines	Functions	Alloc. sites	Fields
Builtin	225	161	1,039	190
DOM	21,881	12,696	44,947	1,326
WinJS	404	346	1,114	445
Windows 8 API	7,213	2,970	13,989	3,834
Total	29,723	16,173	61,089	5,795

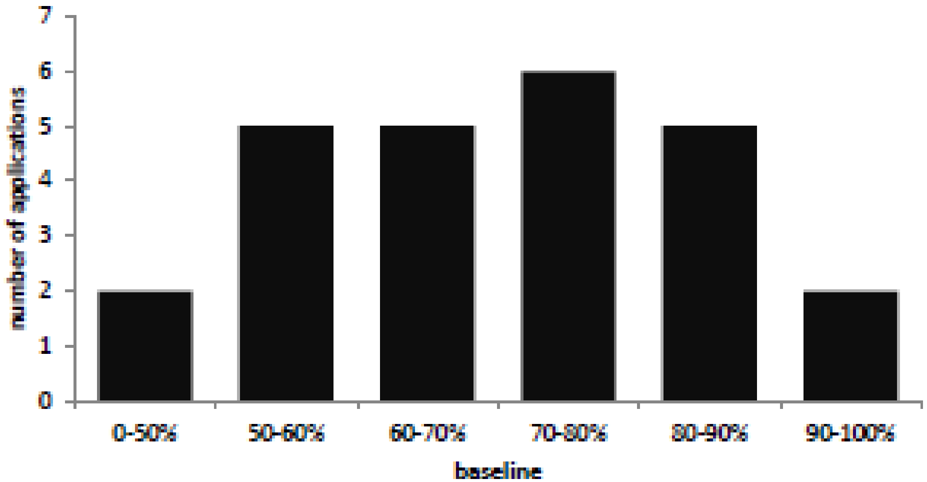
Fig. 2: Approximate stub sizes for widely-used libraries.

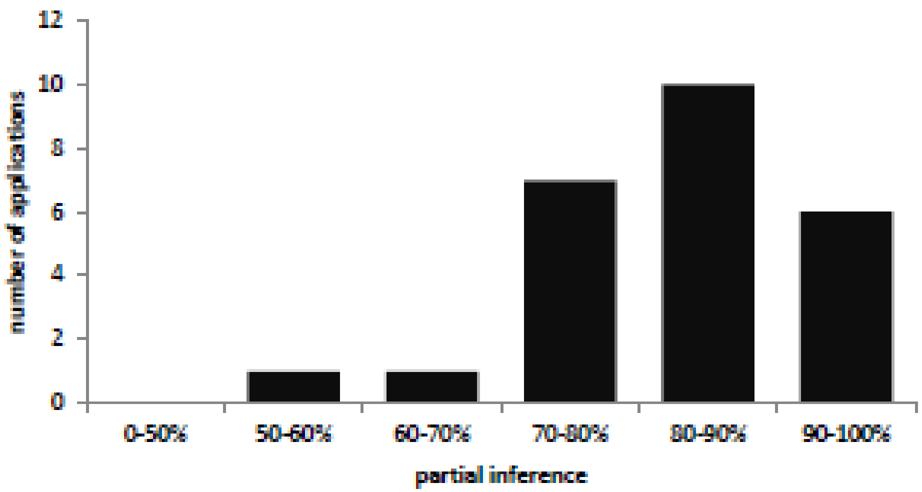
Stub size: 30,000 lines

Take stubs into account

Application size: 1,587 lines

Call Graph Resolution





- **Baseline**: points-to analysis without considering stubs.
- Partial Inference Algorithm

- **Comparison:**

  baseline resolved much few call sites

  partial inference algorithm is effective in recovering missing flow

Case studies – WinRT API Resolution

Technique	APIs used
naïve analysis	684
points-to	800
points-to + partial	1,304
points-to + full	1,320
Total	4,108

- Resolve calls to WinRT APT in Win 8 JavaScript applications
- Partial inference and full inference can find out much more WinRT uses

Case studies – Auto-complete

	Category	Code	Eclipse ✓	Eclipse #	IntelliJ ✓	IntelliJ #	VS 2010 ✓	VS 2010 #	VS 2012 ✓	VS 2012 #
		PARTIAL INFERENCE								

1	**DOM Loop**	`var c = document.getElementById("canvas");`
		`var ctx = c.getContext("2d");`
		`var h = c.height;`
		`var w = c.w␣`
			✗	0	✓	35	✗	26	✓	1

2	**Callback**	`var p = {firstName : "John", lastName : "Doe"};`
		`function compare(p1, p2) {`
		`  var c = p1.firstName < p2.firstName;`
		`  if(c != 0) return c;`
		`  return p1.last␣`
		`}`
			✗	0	✓	9	✗	7	✓*	k

3	**Local Storage**	`var p1 = {firstName : "John", lastName : "Doe"};`
		`localStorage.putItem("person", p1);`
		`var p2 = localStorage.getItem("person");`
		`document.writeln("Mr." + p2.lastName+`
		`", " + p2.f␣);`
			✗	0	✓	50+	✗	7	✗	7

		FULL INFERENCE								

4	**Namespace**	`WinJS.Namespace.define("Game.Audio",`
		`  play : function() {}, volume : function() {}`
		`);`
		`Game.Audio.volume(50);`
		`Game.Audio.p␣`
			✗	0	✓	50+	✗	1	✓*	k

5	**Paths**	`var d = new Windows.UI.Popups.MessageDialog();` `var m = new Windows.UI.␣`	✗	0	✗	250+	✗	7	✓*	k

Fig. 15: Auto-complete comparison. ⋆ means that inference uses all identifiers in the program. "␣" marks the auto-complete point, the point where the developer presses Ctrl+Space or a similar key stroke to trigger auto-completion.

Performance

- Running time of partial and full analysis are quite modest. (full analyses finish under 2-3 seconds)
- Full inference requires more iterations to reach fixpoint (approximately 2 to 3 times as many iterations as partial inference)
- Full inference is fast than partial inference (2 to 4 times faster): **cost of stubs**

Precision and Soundness

App	OK	Incomplete	Unsound	Unknown	Stubs	Total
app1	16	1	2	0	1	20
app2	11	5	1	0	3	20
app3	12	5	0	0	3	20
app4	13	4	1	0	2	20
app5	13	4	0	1	2	20
app6	15	2	0	0	3	20
app7	20	0	0	0	0	20
app8	12	5	0	1	2	20
app9	12	5	0	0	3	20
app10	11	4	0	3	2	20
Total	135	35	4	5	21	200

- Manually inspected 20 call sites in 10 benchmarks
- Check if approximated call targets match the actual call targets
- **OK**: the number of call sites which are both sound and complete
- **Incomplete**: the number of call sites are sound, but have spurious targets (Imprecision)
- **Unsound**: the number of call sites for which some call targets are missing
- **Unknown:** the number of call sites for which unable to determine due to code complexity
- **Stubs**: the number of call sites which are unsolved due to problematic stubs.

Unsoundness & Imprecision

- Unsoundness
  - Unable to deal with JSON data being pared
  - Unable to deal with JavaScript type coercion

    (Type coercion means that when the operands of an operator are different types, one of them will be converted to an "equivalent" value of the other operand's type. For instance: boolean == integer, the boolean operand will be converted to an integer first)
- Imprecision
  - Property names shares between different objects.
  - Stub errors

Conclusions

- Approach proposed combining classic points-to analysis with use analysis
- Able to analyze practical large JavaScript applications using complex windows runtime libraries and sophisticated JavaScript libraries
- Improve precision and scalability
- Useful for other applications: API use discovery and auto-completion

Questions ?

Thanks!