Analyzing JavaScript and the Web with WALA (T.J. Watson Libraries for Analysis)

Max Schaefer, Manu Sridharan, Julian Dolby

PLDI 2013 Tutorial

`http://wala.sf.net`

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

What is WALA?

- Java libraries for static and dynamic program analysis
  - (With some JavaScript libraries!)
- Initially developed at IBM T.J. Watson Research Center
- Open source release in 2006 under Eclipse Public License
- <u>Key design goals</u>
  - Robustness
  - Efficiency
  - Extensibility

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

(Some) Previous Uses of WALA

- Research
  - over 100 publications (based on Google Scholar)
  - Including two at PLDI'13
  - `http://wala.sf.net/wiki/index.php/Publications.php`
- Products
  - Rational Software Analyzer: NPEs (Loginov et al. ISSTA'08), resource leak detection (Torlak and Chandra, ICSE'10)
  - Rational AppScan: taint analysis (Tripp et al., PLDI'09), string analysis (Geay et al., ICSE'09), JavaScript call graphs (Sridharan et al., ECOOP'12)
  - Tivoli Storage Manager: JavaScript analysis

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

WALA Features: Static Analysis

- Pointer analysis / call graph construction
  - Several algorithms provided (RTA, variants of Andersen's analysis)
  - Highly customizable (e.g., context sensitivity policy)
  - Tuned for performance (time and space)
- Interprocedural dataflow analysis framework
  - Tabulation solver (Reps-Horwitz-Sagiv POPL'95) with extensions
  - Also tuned for performance

Other Key WALA Features

- Multiple language front-ends
  - Will focus on JavaScript here
- Generic analysis utilities / data structures
  - Graphs, sets, maps, constraint solvers, …
- Limited code transformation
  - Main WALA IR is <u>immutable</u>, with no code generation
  - ASTs can be transformed...but advanced
  - <u>Recommendation</u>: use WALA for computing analysis results, do transformation separately
  - JS normalizer can simplify dynamic analysis (details later)

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

What We'll Cover

1. Call graph basics
2. Representation of scripts / methods
3. Representation of HTML / DOM
4. WALA IR for JavaScript
5. Customizing call graphs
6. Utilities implemented in JavaScript
7. Advanced topics

How to get WALA

- Walkthrough on "Getting Started" page at `wala.sf.net`
- Code available on Github: `github.com/wala/WALA`
  - Trunk or previous tagged releases
  - Several Eclipse projects (prefixed with `com.ibm.`):
    - `wala.util`: language-independent utilities
    - `wala.core`: analyses for core WALA SSA IR, Java bytecode
    - `wala.cast`: common framework for AST frontends
    - `wala.cast.js`: JavaScript-specific IR generation, analysis extensions
    - `wala.cast.js.rhino`: converts Mozilla Rhino AST to CAst
- Building the code (see "Getting started" page)
  - Easiest to build / run from Eclipse
  - Ant build files to download 3rd-party jars
  - Recently added Maven build support

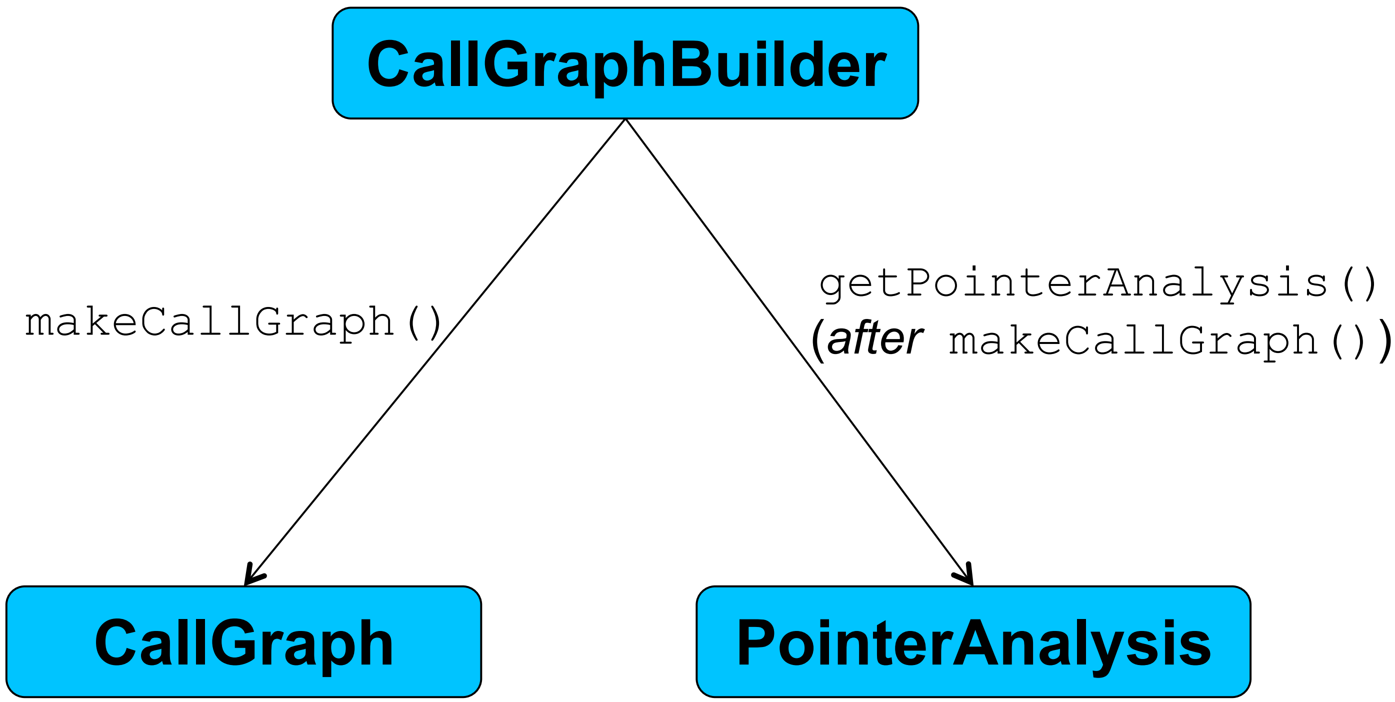
WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Call Graph Basics

WALA and Call Graphs

- Typically, WALA analysis starts with a call graph
  - Focus on inter-procedural analyses
- Nodes for (cloned) methods, edges for call targets
- Often, call graph constructed via pointer analysis
  - WALA usually computes them simultaneously

Call Graph Builder Overview



CallGraph API

- **Nodes of type `CGNode`: `IMethod` + `Context`**
- **`IR` typically obtained from `CGNode`, *not* `IMethod`**
- **To iterate over nodes in `CallGraph` cg:**

  ```
  for (CGNode n: cg) { … }
  ```

- **To find all callees of a `CGNode` n in cg:**

  ```
  cg.getSuccNodes(n)
  ```

- **Callees for `CallSiteReference` site in `CGNode` n:**

  ```
  cg.getPossibleTargets(n, site)
  ```

  - **`CallSiteReference` from relevant `SSAAbstractInvokeInstruction`**

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

PointerAnalysis API

- `PointerKey`: abstraction of a pointer
  - For local variable, `LocalPointerKey`
  - For instance field, `InstanceFieldKey`
- To obtain pointer keys for `PointerAnalysis` pa, use `HeapModel`, e.g.:
  `pa.getHeapModel().getPointerKeyForLocal(…)`
- `InstanceKey`: abstraction of an object
  - For allocation site in some `CGNode`, `AllocationSiteInNode`
- To get points-to set for `PointerKey` pk (`Set<InstanceKey>`): `pa.getPointsToSet(pk)`

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Building CallGraph for HTML

```
URL url = …;
// use Rhino to parse JavaScript
JSCallGraphUtil.setTranslatorFactory(
  new CAstRhinoTranslatorFactory());
// build the call graph
CallGraph cg = JSCallGraphBuilderUtil.makeHTMLCG(url);
```

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Representation of Code Structure

Scopes and hierarchies

- `AnalysisScope`: code to be analyzed
  - mostly invisible for JavaScript analysis
  - always includes `prologue.js` (std library models)
- `IClassHierarchy`
  - Represents type hierarchy (more useful for Java)
  - Resolves names ("references") to representations
    - E.g., `MethodReference` to `IMethod`
    - Save memory by only retaining references in analysis results
- To re-use WALA analyses, `IClassHierarchy` required

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

WALA Name Resolution

Entity references resolved via `IClassHierarchy`

Entity	Reference Type	Resolved Type	Resolver Method
class	TypeReference	IClass	`lookupClass()`
method	MethodReference	IMethod	`resolveMethod()`
field	FieldReference	IField	`resolveField()`

What do these mean for JavaScript???

(fields mean nothing)

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

JS "classes"

- Usually, `IClass` represents a JavaScript *function* (`JavaScriptCodeBody`)
  - Including a function for top-level code in script
  - Including library functions from `prologue.js`
- For well-formedness, all classes “subclass” a synthetic root type
- A few other classes for built-in JS types (Boolean, String, etc.)

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

JS methods

- **Each `IClass` representing a JS function has an `IMethod` representing "normal" function invocations, named `do`**
- ***May* also have a synthetic `IMethod` representing invocations via 'new', named `ctor`**
  - **Added during call graph construction as needed**
  - **Has instructions to model 'new' semantics**
- **Source-level function names**
  - **Look at name of `IMethod`'s declaring class**

<u>src.js</u>

```
function f() {
  function g(){}
}
```

<u>Functions</u>:

```
src.js,
src.js/f,
src.js/f/g
```

Printing IRs

```
public static void printIRs(String filename) {
  // use Rhino to parse JavaScript
  JSCallGraphUtil.setTranslatorFactory(
    new CAstRhinoTranslatorFactory());
  // build a class hierarchy, for access to code info
  IClassHierarchy cha =
     JSCallGraphUtil.makeHierarchyForScripts(filename);
  // for constructing IRs
  IRFactory<IMethod> factory = AstIRFactory.makeDefaultFactory();
  for (IClass klass : cha) {
    // ignore models of built-in JavaScript methods
    if (!klass.getName().toString().startsWith("Lprologue.js")) {
      // get the IMethod representing the code (the 'do' method)
      IMethod m = klass.getMethod(AstMethodReference.fnSelector);
      if (m != null) {
        IR ir = factory.makeIR(m, Everywhere.EVERYWHERE,
                                      new SSAOptions());
        System.out.println(ir);
      }
    }
  }
}
```

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

HTML Support in WALA

- Extracting JS code from HTML (including node attributes)
- Generating JS code to model (static) DOM structure
- Models of DOM APIs in preamble.js (incomplete)
- Modeling semantics of browser-based JS (window object)
- Detailed source locations, including nesting within HTML

Example

Input

```
<html>
<body>
<script>
function fizz() {
  alert("hi");
}
</script>
<a onclick="fizz()"></a>
</body>
</html>
```

Model (roughly)

```
window.MAIN = function WINDOW_MAIN(){
  function fizz() {
    alert(“hi”);
  }
  var aNode1 = new DOMHTMLElem();
  aNode1.onclick = function() {
    fizz();
  }
  var scriptNode1 = new DOMHTMLElem();
  … // construct other DOM nodes
  while (true) {
    aNode1.onclick();
  }
};
window.MAIN();
```

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

Notes on HTML modeling

- All JS code nested in `__WINDOW_MAIN__` function
  - To help model window as global object
- Generated JS model stored in temp file by default
  - To control, see `JSSourceExtractor`
- For no modeling of DOM node structure, use `DomLessSourceExtractor`
- Uses Jericho HTML parser by default; can also use Validator.nu HTML5 parser (see `com.ibm.wala.cast.js.html.nu_validator`)
- See `PrintIRs.printIRsForHTML(String)` for example code

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

Source Positions

- For JS, usually have start line, start and end offset for each method / IR instruction (assuming Rhino)
- For location of method `IMethod` *m*: `((ASTMethod)m).getSourcePosition()`
- For location of IR instruction at offset *i* in `ASTMethod` *m*: `m.getSourcePosition(i)`
- For scripts in HTML, `IncludedPositions` provided
  - Call `getIncludePosition()` to get `Position` of corresponding `<script>` tag
  - Inline script positions relative to start of script; combine with include info to find position in HTML

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

Part Two:
WALA JavaScript IR

Basic Structure

- Traditional 3-address IR
- Structured by Control Flow Graph (CFG)
- Static Single Assignment (SSA) form
  - fully-pruned SSA
  - integrated copy propagation

WALA

Outline

- Example JavaScript code
- Overview of IR
- Handling JavaScript features
- Constructors
- Source position information

Example Code

```
function outer(s) {
  var x = arguments[0];
  if (s.indexOf('o') > 0) {
    function inner(y)
      var t = ".suffix";
      var arr = [ x + t, y ];
      this.data = arr;
    }
    return new inner(s);
  }
}
var outerProp = outer("outer").data;
```

Example Code

```
function outer(s) {
 var x = arguments[0];
  if (s.indexOf('o') > 0) {
    function inner(y)
      var t = ".suffix";
      var arr = [ x + t, y ];
      this.data = arr;
    }
    return new inner(s);
  }
}
var outerProp = outer("outer").data;
```

prototype chain
lexical scoping
arguments array
"method" calls
copy propagation
object creation

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Overview of IR

Top-level IR

```
0    v1 = new <JavaScriptLoader,LArray>@0
1    v6 = global:Function
2    v2 = construct v6@2 v4:#.../outer
3    global:outer = v2
4    v9 = global:$$undefined
6    v14 = global:outer
7    check v14
BB2
8    v16 = global:__WALA__int3rnal__global
9    check v16
BB3
10    v13 = invoke v14@10 v16,v17:#outer
BB4
12    v10 = prototype_values(v13)
13    v12 = getfield < JavaScriptLoader, LRoot,
            data, <JavaScriptLoader,LRoot>> v10
```

Allocate `arguments` array

```
0    v1 = new <JavaScriptLoader,LArray>@0
1    v6 = global:Function
2    v2 = construct v6@2 v4:#.../outer
3    global:outer = v2
4    v9 = global:$$undefined
6    v14 = global:outer
7    check v14
BB2
8    v16 = global:__WALA__int3rnal__global
9    check v16
BB3
10    v13 = invoke v14@10 v16,v17:#outer
BB4
12    v10 = prototype_values(v13)
13    v12 = getfield < JavaScriptLoader, LRoot,
          data, <JavaScriptLoader,LRoot>> v10
```

Read JavaScript standard library `Function` object

```
0    v1 = new <JavaScriptLoader,LArray>@0
1    v6 = global:Function
2    v2 = construct v6@2 v4:#.../outer
3    global:outer = v2
4    v9 = global:$$undefined
6    v14 = global:outer
7    check v14
BB2
8    v16 = global:__WALA__int3rnal__global
9    check v16
BB3
10    v13 = invoke v14@10 v16,v17:#outer
BB4
12    v10 = prototype_values(v13)
13    v12 = getfield < JavaScriptLoader, LRoot,
            data, <JavaScriptLoader,LRoot>> v10
```

Create first-class function object `outer`

```
0    v1 = new <JavaScriptLoader,LArray>@0
1    v6 = global:Function
2    v2 = construct v6@2 v4:#.../outer
3    global:outer = v2
4    v9 = global:$$undefined
6    v14 = global:outer
7    check v14
BB2
8    v16 = global:__WALA__int3rnal__global
9    check v16
BB3
10    v13 = invoke v14@10 v16,v17:#outer
BB4
12    v10 = prototype_values(v13)
13    v12 = getfield < JavaScriptLoader, LRoot,
            data, <JavaScriptLoader,LRoot>> v10
```

Declared functions are *global names* in JavaScript

```
0    v1 = new <JavaScriptLoader,LArray>@0
1    v6 = global:Function
2    v2 = construct v6@2 v4:#.../outer
3    global:outer = v2
4    v9 = global:$$undefined
6    v14 = global:outer
7    check v14
BB2
8    v16 = global:__WALA__int3rnal__global
9    check v16
BB3
10    v13 = invoke v14@10 v16,v17:#outer
BB4
12    v10 = prototype_values(v13)
13    v12 = getfield < JavaScriptLoader, LRoot,
             data, <JavaScriptLoader,LRoot>> v10
```

The `Undefined` object is special

```
0    v1 = new <JavaScriptLoader,LArray>@0
1    v6 = global:Function
2    v2 = construct v6@2 v4:#.../outer
3    global:outer = v2
4    v9 = global:$$undefined
6    v14 = global:outer
7    check v14
BB2
8    v16 = global:__WALA__int3rnal__global
9    check v16
BB3
10    v13 = invoke v14@10 v16,v17:#outer
BB4
12    v10 = prototype_values(v13)
13    v12 = getfield < JavaScriptLoader, LRoot,
          data, <JavaScriptLoader,LRoot>> v10
```

Check for cases requiring `ReferenceError`

```
0    v1 = new <JavaScriptLoader,LArray>@0
1    v6 = global:Function
2    v2 = construct v6@2 v4:#.../outer
3    global:outer = v2
4    v9 = global:$$undefined
6    v14 = global:outer
7    check v14
BB2
8    v16 = global:__WALA__int3rnal__global
9    check v16
BB3
10    v13 = invoke v14@10 v16,v17:#outer
BB4
12    v10 = prototype_values(v13)
13    v12 = getfield < JavaScriptLoader, LRoot,
             data, <JavaScriptLoader,LRoot>> v10
```

JavaScript standard library `Global` object

```
0    v1 = new <JavaScriptLoader,LArray>@0
1    v6 = global:Function
2    v2 = construct v6@2 v4:#.../outer
3    global:outer = v2
4    v9 = global:$$undefined
6    v14 = global:outer
7    check v14
BB2
8    v16 = global:__WALA__int3rnal__global
9    check v16
BB3
10    v13 = invoke v14@10 v16,v17:#outer
BB4
12    v10 = prototype_values(v13)
13    v12 = getfield < JavaScriptLoader, LRoot,
          data, <JavaScriptLoader,LRoot>> v10
```

Call function `outer` (`Global` is defined to be `this`)

```
0    v1 = new <JavaScriptLoader,LArray>@0
1    v6 = global:Function
2    v2 = construct v6@2 v4:#.../outer
3    global:outer = v2
4    v9 = global:$$undefined
6    v14 = global:outer
7    check v14
BB2
8    v16 = global:__WALA__int3rnal__global
9    check v16
BB3
10    v13 = invoke v14@10 v16,v17:#outer
BB4
12    v10 = prototype_values(v13)
13    v12 = getfield < JavaScriptLoader, LRoot,
            data, <JavaScriptLoader,LRoot>> v10
```

Get all *transitive* prototype objects

```
0    v1 = new <JavaScriptLoader,LArray>@0
1    v6 = global:Function
2    v2 = construct v6@2 v4:#.../outer
3    global:outer = v2
4    v9 = global:$$undefined
6    v14 = global:outer
7    check v14
BB2
8    v16 = global:__WALA__int3rnal__global
9    check v16
BB3
10    v13 = invoke v14@10 v16,v17:#outer
BB4
12    v10 = prototype_values(v13)
13    v12 = getfield < JavaScriptLoader, LRoot,
            data, <JavaScriptLoader,LRoot>> v10
```

Read properties of object and prototypes

```
0    v1 = new <JavaScriptLoader,LArray>@0
1    v6 = global:Function
2    v2 = construct v6@2 v4:#.../outer
3    global:outer = v2
4    v9 = global:$$undefined
6    v14 = global:outer
7    check v14
BB2
8    v16 = global:__WALA__int3rnal__global
9    check v16
BB3
10    v13 = invoke v14@10 v16,v17:#outer
BB4
12    v10 = prototype_values(v13)
13    v12 = getfield < JavaScriptLoader, LRoot,
          data, <JavaScriptLoader,LRoot>> v10
```

Handling JavaScript

IR for `outer`

```
0   v4 = new <JavaScriptLoader,LArray>@0
1   v6 = global:$$undefined
2   lexical:x@...outer = v6
3   v12 = global:Function
4   v8 = construct v12@4 v10:#.../inner
6   v15 = prototype_values(v4)
7   v13 = fieldref v15.v14:#0.0
BB2
8   lexical:x@...outer = v13
13  v21 = dispatch v20:#indexOf@13 v3,v22:#o
BB3
14  v16 = binaryop(gt) v21 , v14:#0.0
15  conditional branch(eq) v16,v24:#0
BB4
16  v25 = construct v8@16 v3
```

IR for `inner`

```
0    v4 = new <JavaScriptLoader,LArray>@0
1    v6 = global:$$undefined
3    v8 = global:$$undefined
6    v13 = global:Array
7    check v13
BB2
8    v11 = construct v13@8
BB3
9    v18 = lexical:x@...outer
10   check v18
BB4
11   v16 = binaryop(add) v18 , v10:#.suffix
12   fieldref v11.v15:#0 = v16 = v16
13   fieldref v11.v19:#1 = v3 = v3
15   fieldref v2.v20:#data = v11 = v11
```

JavaScript IR Issues

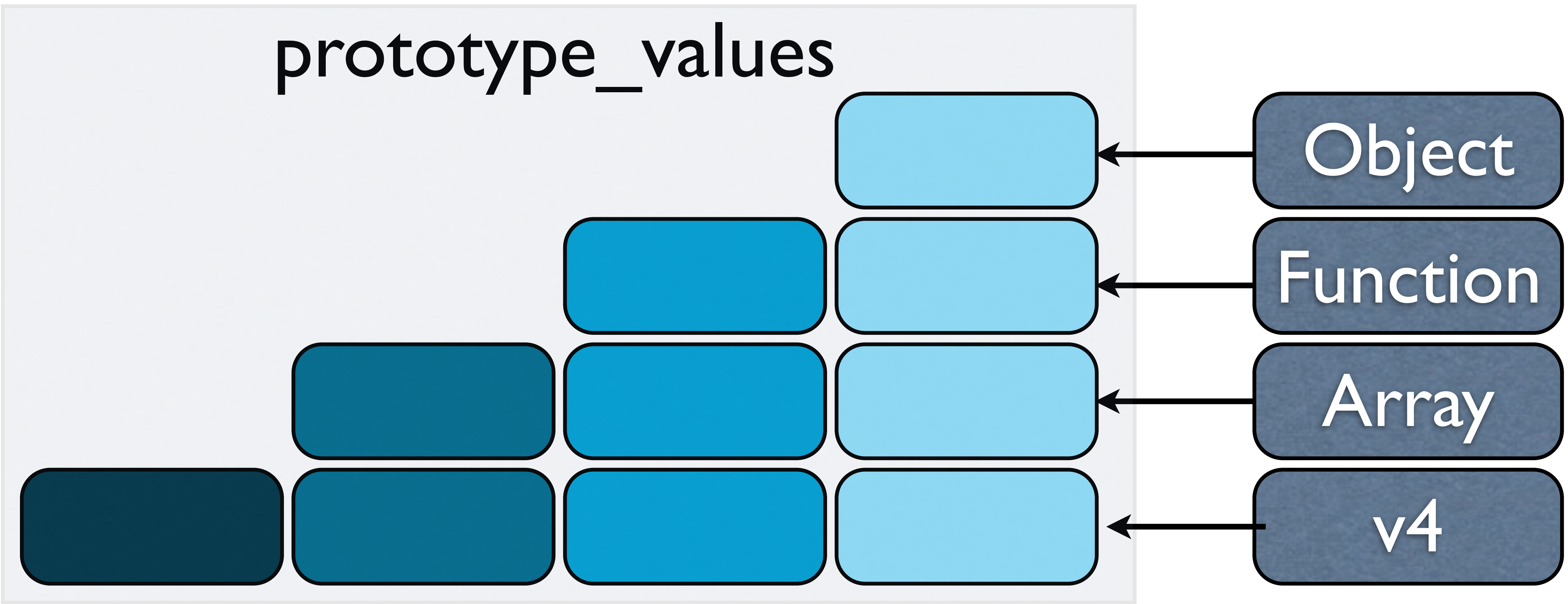
- Prototype chain
- Handling calls
- Object creation
- Lexical scoping
- Arguments array
- Copy propagation

Prototype Chain

- JavaScript uses prototype-based inheritence
  - objects point a ‘prototype’
  - properties can be found in prototype
- Flow-insensitive model of all prototypes
  - conservative model of inheritence
  - no model for must-override

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Prototype Chains




```
0   v4 = new <JavaScriptLoader,LArray>@0
...
6   v15 = prototype_values(v4)
7   v13 = fieldref v15.v14:#0.0
```

Handling Calls

- Both function and method calls
  - objects have functions as propertiws
  - sometimes objects used as 'this' pointers

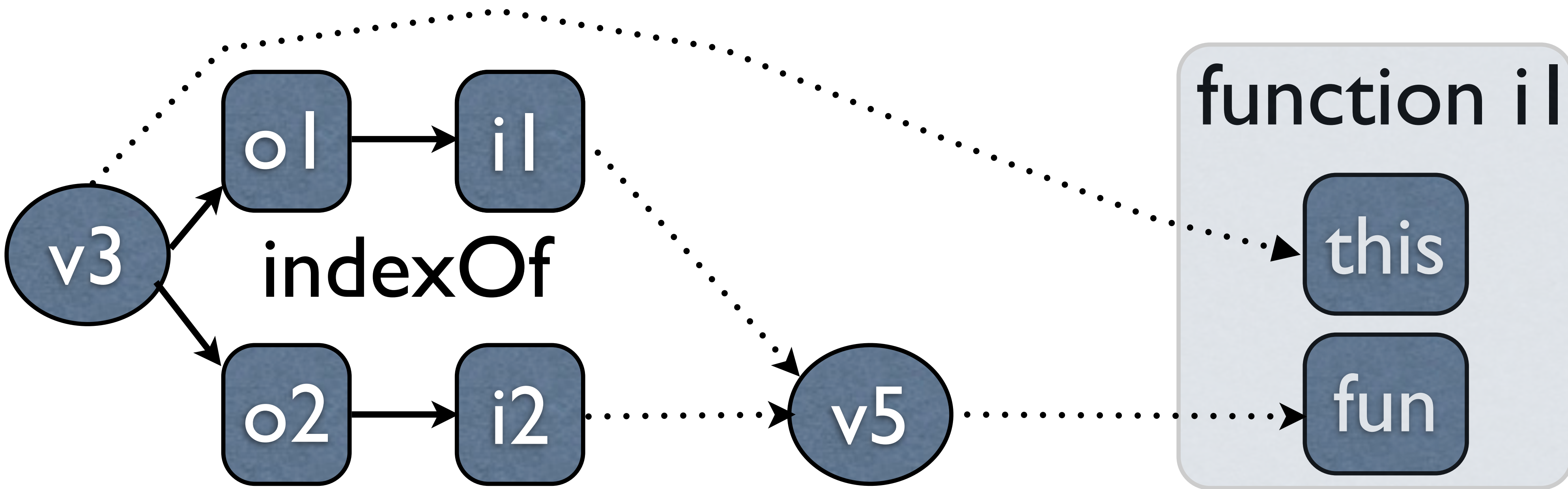
method	not method
`a.m(3);`	`var f = a.m;` `f(3);`

- WALA models `new` as another call type

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Handling Calls

```
s.indexOf('o')

13 v21 = dispatch v20:#indexOf@13 v3,v22:#o
```




```
var f = s.indexOf;
f('o')

6  v15 = prototype_values(v3)
7  v5 = fieldref v15. v20:#indexOf
13 v21 = invoke v5@13 v3,v22:#o
```

Object Creation

- `new` takes an expression, not type

```
var x = (...)? Object: Array;
var y = new x(5)
```

- `new` has diverse semantics

expression	meaning
Object()	fresh object
Object(5)	return 5
Array(3)	array size 3
Array(3,2)	return [3, 2]

Object Creation

- Model `new` as a dynamic dispatch
  - `new` translated to special method call
  - constructor methods generated

```
new inner(s)
```

```
3   v12 = global:Function
4   v8 = construct v12@4 v10:#.../inner
...
BB4
16 v25 = construct v8@16 v3
```

Lexical Scoping

- Access to enclosing function state
  - read and write support, unlike Java
  - allows “upward funargs”
- WALA models as heap locations
  - reads and writes are flow insensitive
  - does not do SSA renaming for them

Lexical Scoping

```
                         outer
...
2  lexical:x@...outer = v6
...
7   v13 = fieldref v15.v14:#0.0
BB2
8  lexical:x@...outer = v13
...
```

```
                         inner
...
9    v18 = lexical:x@...outer
10    check v18
BB4
11    v16 = binaryop(add) v18 , v10:#.suffix
...
```

Arguments Array

- JavaScript reifies arguments in an array
  - array-indexed access to all parameters
  - allows access to unnamed parameters
- WALA models with explicit array creation
  - “arguments” assigned to new array
  - accesses look like normal array accesses
- Pointer analysis adds dataflow from callers

Arguments array

```
0  v4 = new <JavaScriptLoader,LArray>@0
...
6  v15 = prototype_values(v4)
7  v13 = fieldref v15.v14:#0.0
BB2
8  lexical:x@...outer = v13
...
```

Copy Propagation

- SSA construction removes assignments
  - statements like `t = ".suffix"`
  - simplifies later analyses
- For flow-sensitive analyses, add Pi nodes
  - create new values in control contexts
  - `PiNodeCallGraphTest` has example

Copy Propagation

```
0    v4 = new <JavaScriptLoader,LArray>@0
1    v6 = global:$$undefined
2    v7 = v10:#.suffix
3    v8 = global:$$undefined
...
BB4
11    v16 = binaryop(add) v18 , v7
...
```

Constructors

Constructors

- Model semantics of `new`
- Generate synthetic IR
- Generate constructor for each case
  - different types
  - different parameter counts at calls
- Illustrate with two examples

Array Constructor for 0 parameters

Create new `Array` object

```
1    v4 = fieldref v1.v3:#prototype
2    v5 = new <JavaScriptLoader,LArray>@2
BB2
3    set_prototype(v5, v4)
4    putfield v5 = v7:#0 < JavaScriptLoader,
LRoot, length, <JavaScriptLoader,LRoot> >
BB3
5    return v5
```

Semantics is to create a 0-length array

Array Constructor for 0 parameters

Copy prototype to new object

```
1    v4 = fieldref v1.v3:#prototype
2    v5 = new <JavaScriptLoader,LArray>@2
BB2
3    set_prototype(v5, v4)
4    putfield v5 = v7:#0 < JavaScriptLoader,
LRoot, length, <JavaScriptLoader,LRoot> >
BB3
5    return v5
```

Semantics is to create a 0-length array

Array Constructor for 0 parameters

set `length` **to 0**

```
1    v4 = fieldref v1.v3:#prototype
2    v5 = new <JavaScriptLoader,LArray>@2
BB2
3    set_prototype(v5, v4)
4    putfield v5 = v7:#0 < JavaScriptLoader,
LRoot, length, <JavaScriptLoader,LRoot> >
BB3
5    return v5
```

Semantics is to create a 0-length array

`inner` Constructor

```
1    v5 = getfield < JavaScriptLoader, LRoot,
prototype, <JavaScriptLoader,LRoot> > v1
BB2
2    v6 = new <JavaScriptLoader,LObject>@2
BB3
3    set_prototype(v6, v5)
4    v8 = invoke v1@4 v6,v2 exception:v9
BB4
5    return v8
BB5
6    return v6
```

create new `Object`

`inner` Constructor

```
1    v5 = getfield < JavaScriptLoader, LRoot,
prototype, <JavaScriptLoader,LRoot> > v1
BB2
2    v6 = new <JavaScriptLoader,LObject>@2
BB3
3    set_prototype(v6, v5)
4    v8 = invoke v1@4 v6,v2 exception:v9
BB4
5    return v8
BB5
6    return v6
```

copy `inner` **prototype to new object**

`inner` Constructor

```
1    v5 = getfield < JavaScriptLoader, LRoot,
prototype, <JavaScriptLoader,LRoot> > v1
BB2
2    v6 = new <JavaScriptLoader,LObject>@2
BB3
3    set_prototype(v6, v5)
4    v8 = invoke v1@4 v6,v2 exception:v9
BB4
5    return v8
BB5
6    return v6
```

call `inner` **with new object as** `this` **parameter**

`inner` Constructor

```
1    v5 = getfield < JavaScriptLoader, LRoot,
prototype, <JavaScriptLoader,LRoot> > v1
BB2
2    v6 = new <JavaScriptLoader,LObject>@2
BB3
3    set_prototype(v6, v5)
4    v8 = invoke v1@4 v6,v2 exception:v9
BB4
5    return v8
BB5
6    return v6
```

return result of `inner`, if not null, or new object

Source Mapping

Source Locations

- WALA preserves source information
  - variable names available from SSA
  - IR instruction source locations from IR
- Information depends on front end parser
  - Rhino 1.7R3 gives lines, character offsets

`inner` Example

```
...
13    fieldref v11.v19:#1 = v3 = v3
tutorial-example.js [142->154] (line 7)
{11=[arr], 3=[y]}
...
```

```
AstIR ir;

String[] names =
  ir.getLocalMap().getLocalNames(pc, vn);
Position functionPos =
  ir.getMethod().getSourcePosition();
Position irPos =
  ir.getMethod().getSourcePosition(instIdx);
```

WALA

Position interface

```
public interface Position extends Comparable {
  int getFirstLine();
  int getLastLine();
  int getFirstCol();
  int getLastCol();
  int getFirstOffset();
  int getLastOffset();
  URL getURL();
  InputStream getInputStream() throws IOException;
}
```

Analyzing JavaScript and the Web with WALA

Max Schaefer, Manu Sridharan, Julian Dolby

PLDI 2013 Tutorial

`http://wala.sf.net`

Overview

- Context Sensitivity
- Advanced Topics
- Field-based Call Graph Construction
- WALADelta
- JS_WALA

Overview

- Context Sensitivity
  - Overview
  - Contexts, Context Keys, Context Items
  - Filtered Pointer Keys
  - Context Selectors
- Field-based Call Graph Construction
- Advanced Topics
- WALADelta
- JS_WALA

Example: No Context Sensitivity

```
function id(x) { return x; }

var x = 23, s = "Hello", y, t;
y = id(x);
t = id(s);
```

function id is only analysed once; analysis concludes number 23 and string "Hello" can flow into both y and t

Example: With Context Sensitivity

function id_1(x) { **return** x; }

function id_2(x) { **return** x; }

var x = 23, s = "Hello", y, t;

y = id_1(x);

t = id_2(s);

function id is analysed in two different contexts; analysis concludes that number 23 can flow into y and string "Hello" can flow into t, but not vice versa

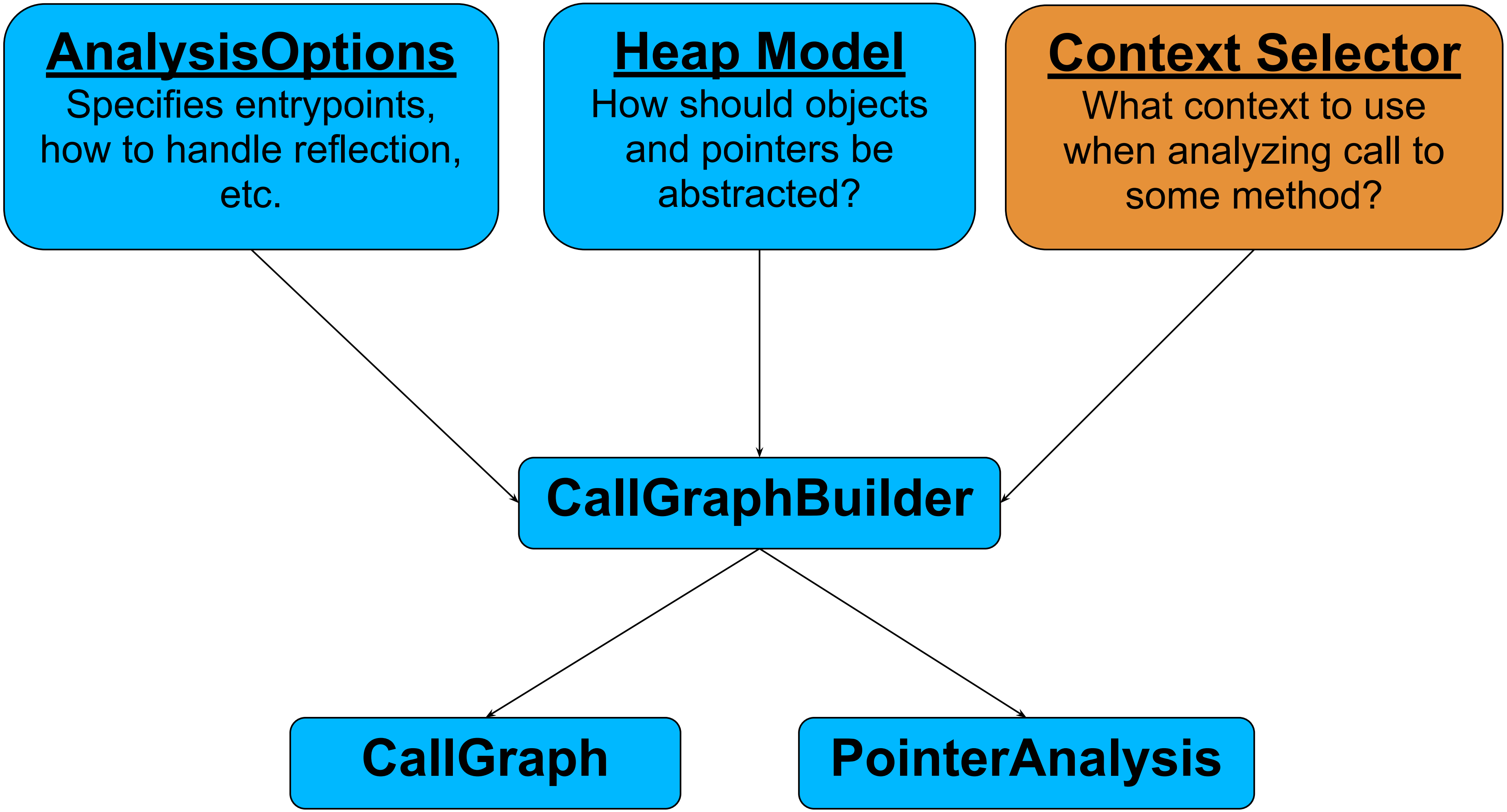
WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

Context Sensitivity

- basic idea: analyse same function separately for different contexts (based on call site, receiver object, etc.) to improve precision
- conceptually, the function is "cloned" for every context: each clone has separate abstract variables for parameters, local variables, and return values
- data flow is kept apart between clones, thus increasing precision, which can in turn improve scalability
- on the other hand, cloning increases the number of abstract variables and of constraints, making the analysis more expensive
- in general, it is hard to predict whether additional context sensitivity will speed up or slow down the analysis

Call Graph Builder Overview



ContextSelector Interface

```
package com.ibm.wala.ipa.callgraph;
public interface ContextSelector {
  Context getCalleeTarget(CGNode caller,
                          CallSiteReference site,
                          IMethod callee,
                          InstanceKey[] args);

  IntSet getRelevantParameters(CGNode caller,
                               CallSiteReference site);
}
```

context to use

information about call site

which arguments to include in args

Overview

- Context Sensitivity
  - Overview
  - Contexts, Context Keys, Context Items
  - Filtered Pointer Keys
  - Context Selectors
- Field-based Call Graph Construction
- Advanced Topics
- WALADelta
- JS_WALA

Context Interface

```
package com.ibm.wala.ipa.callgraph;
public interface Context {
  ContextItem get(ContextKey name);
}
```

simply a map from keys to items!

```
public interface ContextItem {}
```

marker interfaces; invent your own!

```
public interface ContextKey {}
```

Note: context keys in array ContextKey.PARAMETERS have special meaning to pointer analysis (discussed later)

Example Context: Everywhere

```
package com.ibm.wala.ipa.callgraph.impl;
public class Everywhere implements Context {
  public static final Everywhere EVERYWHERE = new Everywhere();
  private Everywhere() {}

  public ContextItem get(ContextKey name) { return null; }

  @Override public int hashCode() { ... }
  @Override public boolean equals(Object obj) { ... }
}
```

singleton

context carries no information

Every context should override hashCode and equals to implement value semantics!

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Example Context: Call Strings

```
package com.ibm.wala.ipa.callgraph.propagation.cfa;

public class CallStringContext implements Context {
  private final CallString cs;

  public CallStringContext(CallString cs) { this.cs = cs; }

  public ContextItem get(ContextKey name) {
    if(CALL_STRING == name)
      return cs;
    else
      return null;
  }

  @Override public int hashCode() { ... }

  @Override public boolean equals(Object obj) { ... }
}
```

essentially just a bounded-length IMethod array

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

Example Context: Object Sensitivity

```java
package com.ibm.wala.ipa.callgraph.propagation;

public class ReceiverInstanceContext implements Context {
  private final InstanceKey ik;

  public ReceiverInstanceContext(InstanceKey ik) { this.ik = ik; }

  public ContextItem get(ContextKey name) {
    if(name == ContextKey.RECEIVER)
      return ik;
    else if(name == ContextKey.PARAMETERS[0])
      return new SingleInstanceFilter(ik);
    return null;
  }

  @Override public int hashCode() { ... }

  @Override public boolean equals(Object obj) { ... }
}
```

see next slide for explanation

Overview

- Context Sensitivity
  - Overview
  - Contexts, Context Keys, Context Items
  - Filtered Pointer Keys
  - Context Selectors
- Field-based Call Graph Construction
- Advanced Topics
- WALADelta
- JS_WALA

Filtered Pointer Keys: Example

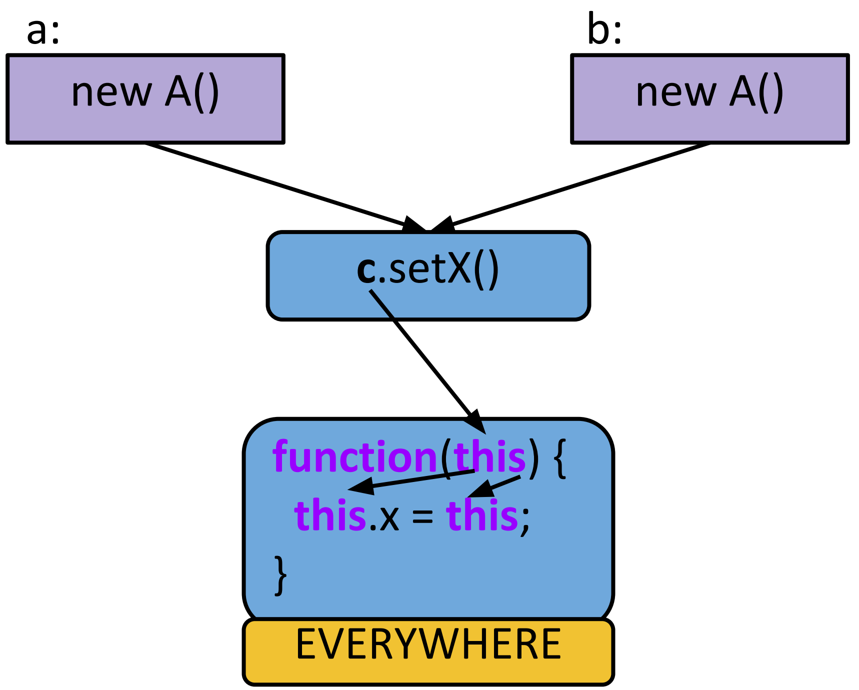
Consider this example:

```
function A() { }
A.prototype.setX = function() { this.x = this; };

var a = new A(), b = new A(),
      c = Math.random() > .5 ? a : b;
// c -> { a, b }
c.setX();
// possible: a.x -> a, b.x -> b; impossible: a.x -> b, b.x -> a
```

Filtered Pointer Keys: Example (ctd.)

Analysing the example context-insensitively yields imprecise results:

```
function A() { }
A.prototype.setX = function() { this.x = this; };

var a = new A(), b = new A(),
    c = Math.random() > .5 ? a : b;
c.setX();
```

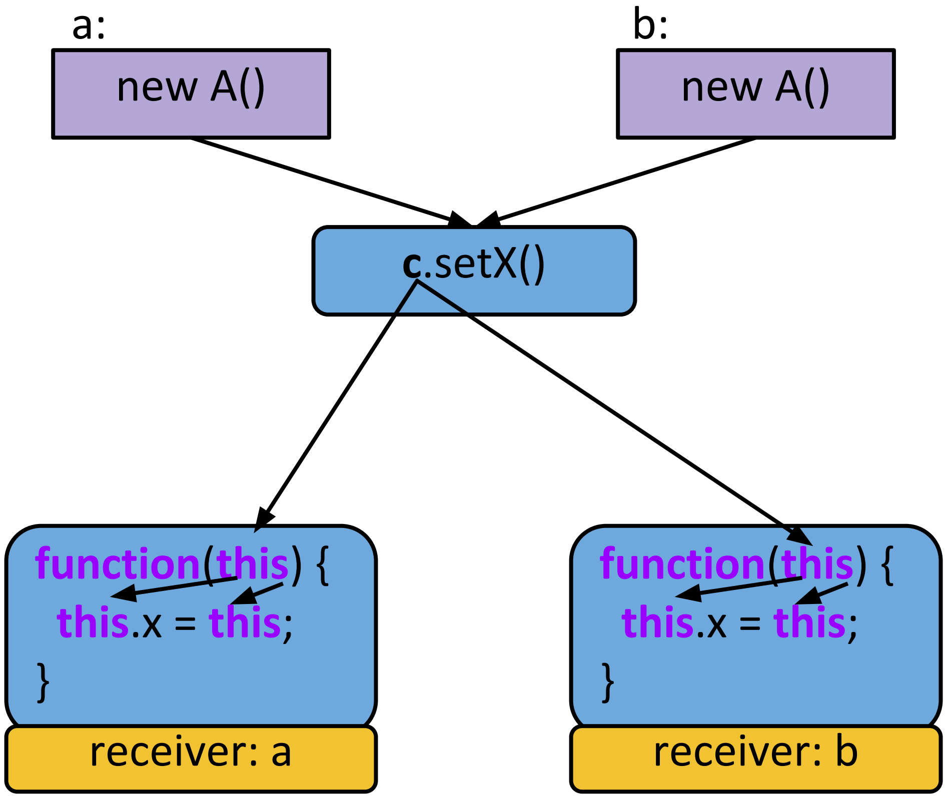


Analysis concludes that a.x -> b is possible!

Filtered Pointer Keys: Example (ctd.)

Analysing the example with object sensitivity does not seem to yield more precise results:

```
function A() { }
A.prototype.setX = function() { this.x = this; };

var a = new A(), b = new A(),
      c = Math.random() > .5 ? a : b;
c.setX();
```



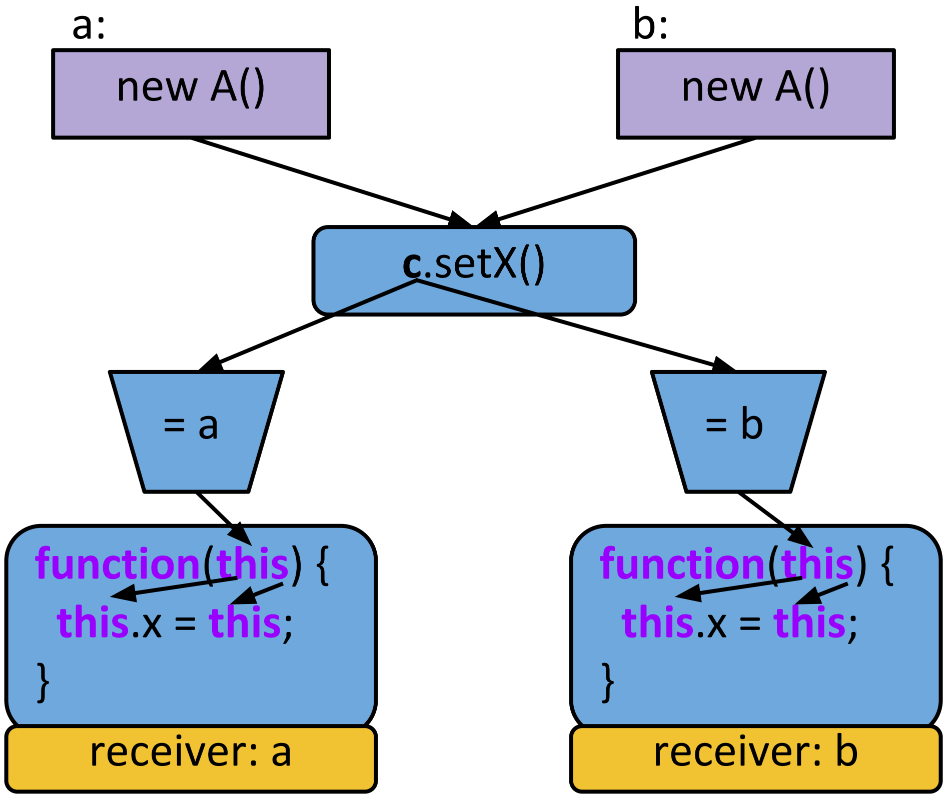
Analysis still concludes that a.x -> b is possible!

Filtered Pointer Keys: Example (ctd.)

Need to introduce filter to enforce that only the desired receiver object flows into **this**:

```
function A() { }
A.prototype.setX = function() { this.x = this; };

var a = new A(), b = new A(),
      c = Math.random() > .5 ? a : b;
c.setX();
```

Analysis concludes that a.x -> b is impossible!



Filtered Pointer Keys

- filtered pointer keys restrict propagation of abstract objects during flow analysis
- a SingleInstanceFilterKey is an abstract variable that only accepts a given instance key ik: its points-to set is either ∅ or { ik }
- filtered pointer keys can be used to split up data flow into a function's arguments among different clones: when setting up interprocedural flow from parameter i to argument i of function f, the analysis checks whether the context of f has an item for ContextKey.PARAMETERS[i]; if so, that item is used to filter data flow into the argument

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Overview

- Context Sensitivity
  - Overview
  - Contexts, Context Keys, Context Items
  - Filtered Pointer Keys
  - Context Selectors
- Field-based Call Graph Construction
- Advanced Topics
- WALADelta
- JS_WALA

Example Selector: 0-CFA

```
public class ContextInsensitiveSelector implements ContextSelector {
  public Context getCalleeTarget(CGNode caller,
                                 CallSiteReference site,
                                 IMethod callee,
                                 InstanceKey[] receiver) {
    return Everywhere.EVERYWHERE;
  }

  public IntSet getRelevantParameters(CGNode caller,
                                      CallSiteReference site) {
    return EmptyIntSet.instance;
  }
}
```

Example Selector: 1-CFA

```
public class OneCFASelector implements ContextSelector {
  public Context getCalleeTarget(CGNode caller,
                                 CallSiteReference site,
                                 IMethod callee,
                                 InstanceKey[] receiver) {
    CallString cs = new CallString(caller.getMethod());
    return new CallStringContext(cs);
  }

  public IntSet getRelevantParameters(CGNode caller,
                                      CallSiteReference site) {
    return EmptyIntSet.instance;
  }
}
```

Note: This is not actual WALA code. WALA implements a more general class `nCFASelector`.

k-CFA for k > 1

- the `getCalleeTarget` method only knows about the immediate caller
- if we want to implement k-CFA for k > 1, we need to somehow find out about the caller's caller (etc.)
- this information can be retrieved from the caller's context:

```
Context context = caller.getContext();
CallString caller_cs = (CallString)context.get(CALL_STRING);
CallString my_cs = new CallString(site, callee, k, caller_cs);
```

- class `CallString` ensures call string is truncated at `k` elements

Object Sensitivity

```
public class ObjectSensitivitySelector implements ContextSelector {
  public Context getCalleeTarget(CGNode caller,
                                 CallSiteReference site,
                                 IMethod callee,
                                 InstanceKey[] arguments) {
      return new ReceiverInstanceContext(arguments[1]);
  }

  public IntSet getRelevantParameters(CGNode caller,
                                      CallSiteReference site) {
   return IntSetUtil.make(new int[]{1});
  }
}
```

Using an Existing Context Selector

- main API entry points for analyzing JavaScript code with common context sensitivity policies are in JSCallGraphBuilderUtil
- build call graph for web page:

  CallGraph makeHTMLCG(URL url, CGBuilderType type)

- CGBuilderType is an enum:
  - ZERO_ONE_CFA: 0-CFA for functions, allocation site abstraction for heap objects
  - ZERO_ONE_CFA_NO_CALL_APPLY: like ZERO_ONE_CFA, but Function.prototype.call and apply are (unsoundly) ignored
  - ONE_CFA: 1-CFA for functions, allocation site abstraction for heap objects

Using Your Own Context Selector

- use `PropagationCallGraphBuilder.setContextSelector` to control which context selector is used
- the call graph builder allows *one* context selector at a time, so most context selectors allow daisy chaining:

```
class MyContextSelector extends ContextSelector {
  ContextSelector base;

  public Context getCalleeTarget(...) {
    if(IAmInterestedInThisCall())
      ...
    else
      return base.getCallTarget(...);
  }

  public IntSet getRelevantParameters(...) {
   IntSet myRelevantParms = ...;
   return base.getRelevantParameters(...).union(myRelevantParms);
  }
}
```

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

Summary: Context Selectors

- context selectors are strategy objects that determine which context a function should be analysed in
- a context selector must implement `getRelevantParameters` to indicate which parameters are relevant to the context selection strategy, and `getCallTarget` to compute the context based on information about the caller (including its context), call site, call target, and relevant arguments
- contexts are represented as arbitrary maps from keys to context items, must obey value semantics
- the set of all contexts should be *finite*, otherwise the analysis is not guaranteed to terminate

Overview

- Context Selectors
- Advanced Topics:
  - Target Selectors
  - Context Interpreters
  - Correlation Tracking
- Field-based Call Graph Construction
- WALADelta
- JS_WALA

Target Selectors

Target selectors (com.ibm.wala.ipa.callgraph.MethodTargetSelector) allow even greater control over call graph construction:

```
public interface MethodTargetSelector {
  IMethod getCalleeTarget(CGNode caller,
                          CallSiteReference site,
                          IClass receiver);
}
```

function to invoke

information about call site

The default JS target selector StandardFunctionTargetSelector simply selects method do of the receiver "class", but more sophisticated target selectors can select any other method, or even create a new synthetic method to serve as call target.

Example: Function.prototype.call

- target selector JavaScriptFunctionDotCallTargetSelector handles reflective calls using Function.prototype.call
- selector creates synthetic IMethod that simply invokes the target method using the right arguments:

```
// recall: v2 is this, v3 first argument etc.
res = call v2, v3, ...
return res
```

- obviously, the code for this method only depends on number of (actual) arguments
- thus, target selector creates only one method per arity and reuses them later

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Example: Constructors

- JavaScript constructor semantics is complex; for instance, `new Object(42)` does not actually create a new object (cf. ECMA 15.2.2.1), `new Function(...)` creates function, etc.
- in WALA, this is modelled by a special target selector and a context selector:
  - target selector `JavaScriptConstructTargetSelector` returns synthetic `IMethod` implementing appropriate constructor semantics
  - context selector adds one level of call string sensitivity to ensure different allocation sites are kept apart across the synthetic `IMethods`

Overview

- Context Sensitivity
- Advanced Topics:
  - Target Selectors
  - Context Interpreters
  - Correlation Tracking
- Field-based Call Graph Construction
- WALADelta
- JS_WALA

Context Interpreters

The analysis is parameterized by a context interpreter that generates IR for CGNodes:

```
public interface SSAContextInterpreter extends RTAContextInterpreter {
  public IR getIR(CGNode node);

  public DefUse getDU(CGNode node);

  public int getNumberOfStatements(CGNode node);

  public ControlFlowGraph<SSAInstruction, ISSABasicBlock>
      getCFG(CGNode n);
}
```

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Context Interpreters (ctd.)

- default context interpreter is ContextInsensitiveSSAInterpreter
- it generates the same IR regardless of the context
- more sophisticated context interpreters can generate custom IR based on the context; this is particularly useful for handling reflection:
  - JavaScriptFunctionApplyContextInterpreter handles Function.prototype.apply
  - ArgumentSpecializationContextInterpreter specializes uses of arguments array where the number of arguments is known

Overview

- Context Sensitivity
- Advanced Topics:
  - Target Selectors
  - Context Interpreters
  - Correlation Tracking
- Field-based Call Graph Construction
- WALADelta
- JS_WALA

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Correlation Tracking

- correlation tracking is a technique for precise handling of correlated dynamic property read/write pairs that access the same property:

```
dest[p] = src[p];
```

- such correlated pairs are extracted into an anonymous function taking p as argument, which is analyzed once per abstract value of p

```
(function(p) { dest[p] = src[p]; })(p);
```

- implementation consists of three parts:
  a. a correlation finder that identifies correlated pairs;
  b. a closure extractor that introduces the functions;
  c. a bespoke context selector.

The Correlation Finder

- implemented by class CorrelationFinder
- creates (context-insensitive) IR for every function in the program
- walks over all IR instructions, looking for dynamic property reads x[p]
- then uses DefUse information to find out whether the result of this dynamic property read flows into a dynamic property write of the form y[q]
- finally checks whether p and q are the same SSA variable
- if all checks succeed, record (x[p], y[q]) as correlated pair

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

The Closure Extractor

- implemented by class ClosureExtractor and factory class CorrelatedPairExtractorFactory
- it is an example of a CAst rewriter that rewrites WALA's AST before code generation
- when constructing a closure extractor, it needs to be passed information about which pieces of code to extract
- CorrelatedPairExtractorFactory uses the CorrelationFinder to provide this information, but in principle the closure extractor can extract (almost) arbitrary pieces of code into closures

Context Selection

- context selector PropertyNameContextSelector is designed to work with correlation extraction mechanism
- parameterized by a parameter index i; if invoked function uses its i'th parameter as a property name, it is analyzed using object sensitivity on that parameter
- closures extracted by the correlation extractor always have property name as first parameter, so normally we set i=2 (NB: 0th parameter is function object, 1st is receiver)
- this turns out to be a generally useful even for functions that do not arise from extraction
- all predefined CGBuilderTypes include correlation tracking by default

Overview

- Context Sensitivity
- Advanced Topics
- **Field-based Call Graph Construction**
- WALADelta
- JS_WALA

Field-based Call Graphs

- WALA's standard pointer analysis-based call graph construction usually do not scale for programs that make heavy use of frameworks
- we additionally provide cheap, approximate field-based call graph construction for clients that do not require soundness
- main highlights:
  - only tracks functions, no other objects
  - treats properties like global variables: like-named properties on different objects are conflated imprecise
  - ignores dynamic features (e[p], eval, arguments, ...)
- for full details see unsound

  Feldthaus, Schaefer, Sridharan, Dolby, Tip. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. ICSE, 2013.

Field-based Call Graph API

- field-based call graph builders subclass com.ibm.wala.cast.js.callgraph.fieldbased.FieldBasedCallGraphBuilder
- three variants:
  - **PessimisticCallGraphBuilder**: does (almost) no interprocedural propagation; very fast, but cannot resolve (most) call backs
  - **OptimisticCallGraphBuilder**: fixpoint iteration to account for interprocedural flows; slower, but more sound
  - **WorklistBasedOptimisticCallGraphBuilder**: faster variant of OptimisticCallGraphBuilder
- use com.ibm.wala.cast.js.rhino.callgraph.fieldbased.test.CGUtil:
  - constructor takes TranslatorFactory (e.g., CAstRhinoTranslatorFactory)
  - method JSCallGraph buildCG(URL url, BuilderType builderType), where builderType is PESSIMISTIC, OPTIMISTIC or OPTIMISTIC_WORKLIST

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Overview

- Context Sensitivity
- Advanced Topics
- Field-based Call Graph Construction
- **WALADelta**
- JS_WALA

WALADelta

WALADelta is a delta debugger for programs that process JavaScript code.

Delta Debugging Problem

Given a JavaScript processor P and a JavaScript program C such that P fails on C, find the smallest subprogram C' of C such that P still fails on C', but not on any smaller subprogram.

Rationale: It is usually easier to find out why P fails on C' than on C.

How WALADelta Works

Standard delta debugging algorithm:

- ensure that P really fails on C
- discard subtree of C's AST, ensuring that resulting program is still syntactically valid, e.g.:
    - remove statements within a block;
    - remove properties in an object literal;
    - discard **else** branch of an **if** statement
- if P still fails on the resulting program, keep reducing
- otherwise go back to previous program and try different reduction
- output smallest C for which P was still found to fail

In practice, not all possible reductions are tried to avoid exponential blowup.

Usage

Usually, WALADelta is invoked like this:

```
node delta.js --cmd my_cmd --errmsg FAILURE input.js
```

- my_cmd can be an arbitrary shell command string that is invoked with the reduced program as its only argument
- if my_cmd prints a message containing FAILURE to stderr, it is considered to have failed on the given input
- WALADelta will output diagnostics on the reduction process and the final reduction result
- much more sophisticated uses are possible, in particular there is special support for debugging WALA analyses; see https://github.com/wala/WALADelta for documentation

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Overview

- Context Sensitivity
- Advanced Topics
- Field-based Call Graph Construction
- WALADelta
- **JS_WALA**

JS_WALA

- JS_WALA is a collection of utilities for processing source-level JavaScript programs
- these tools are themselves implemented in JavaScript
- not directly related to WALA, but can be usefully combined
- currently one main tool: the JavaScript normalizer
- available from https://github.com/wala/JS_WALA

JS_WALA Normalizer

- source-to-source transformation that brings JavaScript programs into simple normal form (see website for details):
  - all **var** declarations hoisted to beginning of scope
  - exactly one **return** statement per function
  - global variable references rewritten into property accesses on global object
  - **for** and **do-while** loops desugared into **while** loops
  - **continue** desugared into **break**; every **break** has explicit label
  - single-statement loop bodies or conditional branches are wrapped into blocks
  - nested expressions flattened out by introducing temporary variables

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

Real-world Example: Debugging JS_WALA with WALADelta

- **problem**: JS_WALA sometimes give statements within function the same position as function itself
- first discovered on 4KLOC JavaScript file fullcalendar.js
- wrote script suspicious_positions.js to detect this problem for given file, and print "found suspicious positions" if so; ran

```
node delta.js --cmd 'node suspicious_positions.js' \
    --errmsg 'found suspicious positions' fullcalendar.js
```

- reduced example:

```
(function() { function setDefaults() { $.extend(); } });
```

- turned out to be a bug with handling nested functions