Static and Dynamic Program Analysis Using WALA
(T.J. Watson Libraries for Analysis)

Julian Dolby and Manu Sridharan
IBM T.J. Watson Research Center
PLDI 2010 Tutorial

http://wala.sf.net
What is WALA?

- Java libraries for static and dynamic program analysis
- Initially developed at IBM T.J. Watson Research Center
- Open source release in 2006 under Eclipse Public License

- **Key design goals**
  - Robustness
  - Efficiency
  - Extensibility
(Some) Previous Uses of WALA

◆ Research
  • over 40 publications from 2003-present
  • Including one at PLDI’10 (MemSAT)
  • 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)
  • Tivoli Storage Manager: Javascript analysis
  • WebSphere: analysis of J2EE apps
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

◆ **Context-sensitive slicing framework**
  - With customizable dependency tracking
Other Key WALA Features

- **Multiple language front-ends**
  - **Bytecode**: Java, .NET (internal)
  - **Source (CAst)**: Java, Javascript, X10, PHP (partial, internal), ABAP (internal)
  - Add your own!

- **Generic analysis utilities / data structures**
  - Graphs, sets, maps, constraint solvers, ...

- **Limited code transformation**
  - Java bytecode instrumentation via Shrike
  - But, main WALA IR is immutable, and no code gen provided
    - Designed primarily for computing analysis info
What We’ll Cover

◆ Overviews of main modules
  • Important features
  • Key class names
  • How things fit together
  • How to customize

◆ “Deep dives” into real code
  • Interprocedural dataflow analysis example
  • CAst Javascript front-end
How to get WALA

◆ Walkthrough on “Getting Started” page at wala.sf.net

◆ Code available in SVN repository
  • Trunk or previous tagged releases
  • Split into several Eclipse projects, e.g.,
    com.ibm.wala.core, com.ibm.wala.core.tests

◆ Dependence on Eclipse
  • Easiest to build / run from Eclipse, but command line also supported
  • Runtime dependence on some Eclipse plugins (progress monitors, GUI functionality, etc.); must be in classpath
WALA Data Structures

- Fixpoint Dataflow Solvers
- Graphs and Algorithms
- Bit Sets
Basic Bit Set Representations

- **IntSet**
- **MutableIntSet**
- **SparseIntSet**
- **BitVectorBase**
- **MutableSparseIntSet**
- **OffsetBitVector**
- **BitVector**

- 2 5 8  n
- b b+1 b+n
- base (b)
- 1 2 3  n

(WALA logo)
Hybrid Bit Set Representation

Split bitset to save space using dense and sparse parts
- Dense words: \((\text{max} - \text{min}) / \text{bits per word}\)
- Sparse words: number of set bits
- Calculate best use of a single dense portion
- Rebalance on mutation, amortizing to save cost
Shared Bit Set Representation

Save space by sharing common portions of bit sets
- State split into common and private portions
- Repository manages set of common portions
- Common portions come and go on demand
Debugging Bit Sets

```java
public boolean add(int i) {
    boolean pr = left.add(i);
    boolean sr = right.add(i);

    if (pr != sr) {
        assert pr == sr;
    }

    return pr;
}
```

- Meant to help debug new bitset implementations
  - Parameterized by two other implementations
  - Assert two implementations give same results
  - Factory interface allows use as standard bitsets
  - For development only: major time and space costs
Basic Graph Representation

NodeManager<T>

Iterator<T> iterator()
int getNumberOfNodes()
void addNode(T n)
boolean containsNode(T n)

EdgeManager<T>

Iterator<T> getPredNodes(T n)
int getPredNodeCount(T n)
Iterator<T> getSuccNodes(T n)
int getSuccNodeCount(T N)
void addEdge(T src, T dst)
boolean hasEdge(T src, T dst)

Graph<T>
Numbered Graph Representation

NumberedNodeManager<T>

NumberedEdgeManager<T>

int getNumber(T N)

T getNode(int number)

int getMaxNumber()

Iterator<T> iterateNodes(IntSet s)

IntSet getSuccNodeNumbers(T node)

IntSet getPredNodeNumbers(T node)
Labeled Graph Representation

LabeledEdgeManager\langle T, U \rangle

- \text{getDefaultLabel()}
- \text{Iterator\langle T \rangle getPredNodes(T N, U label)}
- \text{Iterator\langle ? extends U \rangle getPredLabels(T N)}
- \text{int getPredNodeCount(T N, U label)}
- \text{Iterator\langle ? extends T \rangle getSuccNodes(T N, U label)}
- \text{Iterator\langle ? extends U \rangle getSuccLabels(T N)}
- \text{int getSuccNodeCount(T N, U label)}
- \text{void addEdge(T src, T dst, U label)}
- \text{boolean hasEdge(T src, T dst, U label)}
- \text{Set\langle ? extends U \rangle getEdgeLabels(T src, T dst)}
Generic Graph Operations: Breadth First Search

- BFSIterator
- BoundedBFSIterator
- BFSPathFinder
Generic Graph Operations: Depth First Search

DFSPathFinder
DFSFinishTimelIterator
NumberedDFSFinishTimelIterator
DFSDiscoverTimelIterator
NumberedDFSDiscoverTimelIterator
Generic Graph Operations: SCCs

SCCIterator
Graph Algorithms

- Dominators
- NumberedDominators
- DominanceFrontiers
Dataflow Systems

IKilldallFramework

getFlowGraph()

getTransferFunctionProvider()

ITransferFunctionProvider

getMeetOperator()

f(Vin,...) → Vout

getNodeTransferFunction()

f(Vin) → Vout

getEdgeTransferFunction()
Dataflow Example

At each node, transfer function adds its number

Example from GraphDataflowTest
Dataflow Example

```java
public UnaryOperator<BitVectorVariable> 
getNodeTransferFunction(String node) {
    return new BitVectorUnionConstant(
        values.getMappedIndex(node));
}

public UnaryOperator<BitVectorVariable> 
getEdgeTransferFunction(String from, String to) {
    if (from == nodes[1] && to == nodes[3])
        return new BitVectorFilter(zero());
    else if (from == nodes[1] && to == nodes[2])
        return new BitVectorFilter(one());
    else {
        return BitVectorIdentity.instance();
    }
}

public AbstractMeetOperator<BitVectorVariable> 
getMeetOperator() {
    return BitVectorUnion.instance();
}
```
Node A(0) = { 0 }
Node B(1) = { 0 1 }
Node C(2) = { 0 2 }
Node D(3) = { 1 3 }
Node E(4) = { 0 1 2 3 4 }
Node F(5) = { 0 1 2 3 4 5 }
Node G(6) = { 6 }
Node H(7) = { 7 }
INTERMEDIATE REPRESENTATION
IR Factories

makeIR(method, context, ssa_options)
IR Structure

IR

0
v3 = (v1 == v2)

1

2
If v3

3

4
V4 = invoke equals v1,v2

5

6
V5 = φ(v3,v4)

7

8
return v5

CFG

0-2

3-5

6-8
Instruction Types

- SSAInstruction
  - SSAGotoInstruction
  - SSACCompareInstruction
  - SSACConversionInstruction
  - SSAAAbstractInvokeInstruction
    - SSAINvokeInstruction
  - SSAFieldAccessInstruction
    - SSAGetInstruction
    - SSAPutInstruction
IR Structure: Pi Nodes

- Pi nodes provide distinct value numbers in context
  - A “copy” of a value in a distinct context
  - e.g. inside a conditional or loop
- Used to denote precise information
  - e.g. aliasing with other values
  - e.g. precise type
- Specified by policy during IR creation

SSAPiNodePolicy

getPi(instruction)

value instruction
IR Structure: Pi Nodes

v3 = (v1 == v2)

0

If v3

1

v6 = π(v2)

2

V4 = invoke equals v1, v6

3

v7 = π(v2)

V5 = φ(v3, v4)

4

5

6

7

8

true branch

false branch

return v5

CFG

0-2

3-5

6-8
IR Utilities: DefUse

v3 = (v1 == v2)

If v3

V4 = invoke equals v1, v2

V5 = \( \phi(v3, v4) \)

return v5
IR Utilities: CDG
IR Utilities: Type Inference

Compute most precise 'most general type'
- Uses declared types and other known types
  - e.g. concrete types from constants
  - e.g. concrete types from allocation

Interface allows use across languages

new TypeInference(ir, doPrimitives); getType(vn)
IR Source: CAst Source Map

- IR
- IMeth
- AstMethod
- Position

- GetMethod()
- instructionPosition(index)
- getURL()
- getFirstLine()
- getFirstCol()
- getFirstOffset()
- getLastLine()
- getLastOffset()
- getLastCol()

instructionPosition takes an offset in the instruction array
IR Source: Bytecode Map

GetMethod()

IBytecodeMethod

getBytecodeIndex(int)

getLineNumber(int)

GetBytecodeIndex takes an offset in the instruction array
IR Source: Local names

- IR
- Instruction index
- Local value number
- getLocalNames(int, int)
- Array of names
SCOPES AND CLASS HIERARCHIES
Building a Call Graph

```java
def buildCG(String jarFile) {
    // represents code to be analyzed
    AnalysisScope scope = AnalysisScopeReader
        .makeJavaBinaryAnalysisScope(jarFile, null);
    // a class hierarchy for name resolution, etc.
    IClassHierarchy cha = ClassHierarchy.make(scope);
    // what are the call graph entrypoints?
    Iterable<Entrypoint> e =
        Util.makeMainEntrypoints(scope, cha);
    // encapsulates various analysis options
    AnalysisOptions o = new AnalysisOptions(scope, e);
    // builds call graph via pointer analysis
    CallGraphBuilder builder =
        Util.makeZeroCFABuilder(o, new AnalysisCache(), cha, scope);
    CallGraph cg = builder.makeCallGraph(o, null);
}
```
AnalysisScope

- Represents a set of files to analyze

- To construct from classpath:
  AnalysisScopeReader.makeJavaBinaryAnalysisScope()

- To read info from scope text file:
  AnalysisScopeReader.readJavaScope()
  - Each line of scope file gives loader, lang, type, val
    - E.g., “Application,Java,jarFile,bcel-5.2.jar”
    - Common types: classFile, sourceFile, binaryDir, jarFile
    - Examples in com.ibm.wala.core.tests/dat

- **Exclusions**: exclude some classes from consideration
  - Used to improve scalability of pointer analysis, etc.
  - Also specified in text file; see, e.g.,
    com.ibm.wala.core.tests/dat/GUIExclusions.txt
Background: Class Loaders

- In Java, a class is identified by name and class loader
  - E.g., `< Primordial, java.lang.Object >`

- Class loaders form a tree, rooted at Primordial

- Name lookup first delegates to parent class loader
  - So, can’t write an app class `java.lang.Object`

- User-defined class loaders can provide isolation
  - Used by Eclipse for plugins, J2EE app servers

- WALA naming models class loaders
Multiple Names in Bytecode

```
// this is java.lang.Object
class Object {
    public String toString() { ... }
}
// no overriding of toString()
class B extends Object {}
class A extends B {}

Legal names in bytecode:
<Application, A, toString()>,
<Application, B, toString()>,
<Application, java.lang.Object, toString()>,
<Primordial, java.lang.Object, toString()>

Resolved entity:
<Primordial, java.lang.Object, toString()>
```
## WALA Name Resolution

Entity references resolved via `IClassHierarchy`

<table>
<thead>
<tr>
<th>Entity</th>
<th>Reference Type</th>
<th>Resolved Type</th>
<th>Resolver Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>TypeReference</td>
<td>IClass</td>
<td>lookupClass()</td>
</tr>
<tr>
<td>method</td>
<td>MethodReference</td>
<td>IMethod</td>
<td>resolveMethod()</td>
</tr>
<tr>
<td>Field</td>
<td>FieldReference</td>
<td>IFIELD</td>
<td>resolveField()</td>
</tr>
</tbody>
</table>
More on class hierarchies

- For Java class hierarchy: `ClassHierarchy.make(scope)`
- Supports Java-style subtyping (single inheritance, multiple interfaces)
- Necessary for constructing method IRs, since only resolved `IMethods` have bytecode info

- Watch out for memory leaks!
  - Resolved entities (`IClass`, `IMethod`, etc.) keep pointers back to class hierarchy
  - In general, use entity references in analysis results
INTERPROCEDURAL DATAFLOW ANALYSIS
Tabulation-Based Analysis
(Reps, Horwitz, Sagiv, POPL95)

◆ “Functional approach” to context-sensitive analysis (Sharir and Pnueli, 1981)

◆ Tabulates partial function summaries on demand

◆ Some enhancements in WALA’s implementation
  • Multiple return sites for calls (for exceptions)
  • Optional merge operators
  • Handles partially balanced problems
  • Customizable worklist orderings
Tabulation Overview

TabulationDomain
- Provides numbering of domain facts

IFlowFunctionMap
- Edge flow functions for supergraph

ISupergraph
- Supergraph over which analysis is computed

Seeds
- Initial path edges for analysis

TabulationProblem

TabulationSolver

TabulationResult
Supergraph

- Collection of “procedure graphs,” connected by calls
  - ICFGSupergraph: procedure graphs are CFGs
  - SDGSupergraph: procedure graphs are PDGs

- Example call representation for ICFGSupergraph
  - (For general supergraph, possibly many calls, returns, entries, exits)
Domain / Flow Functions / Seeds

- **TabulationDomain**
  - Maintains mapping from facts to integers
  - Controls worklist priorities (optional)

- **IFlowFunctionMap**
  - Flow functions on supergraph edges
  - All functions map int (or two ints) to IntSet (via TabulationDomain)
  - Function for each type of edge (normal, call->return, call->entry, exit->return)
  - Also, call->return function for “missing” calls
    - For handling missing code, CG expansion (Snugglebug)

- **Seeds (TabulationProblem.initialSeeds())**
  - Depends on problem / domain representation
Partially Balanced Problems

- For when flows can start / end in a non-entrypoint
  - E.g., slice from non-entrypoint statement

- `PartiallyBalancedTabulationProblem`
  - Additional “unbalanced” return flow function for return without a call
  - `getFakeEntry()`: source node for path edges of partially balanced flows

- Compute with `PartiallyBalancedTabulationSolver`

- Examples: `ContextSensitiveReachingDefs`, `Slicer`
Debugging Your Analysis

◆ **IFDSExplorer**
  • Gives GUI view of analysis result
  • Needs paths to GraphViz `dot` executable and PDF viewer

◆ **Set VM property** `com.ibm.wala.fixedpoint.impl.verbose` to true for occasional lightweight info

◆ **Increase** `TabulationSolver.DEBUG_LEVEL` for detailed info
Deep Dive: Reaching Defs

- The classic dataflow analysis, for Java static fields
- Three implementations available in com.ibm.wala.core.tests
  - IntraprocReachingDefs
    - Uses BitVectorSolver, a specialized DataflowSolver
  - ContextInsensitiveReachingDefs
    - Uses BitVectorSolver over interprocedural CFG
  - ContextSensitiveReachingDefs
    - Uses TabulationSolver
- We’ll focus on ContextSensitiveReachingDefs
Example

class StaticDataflow {
    static int f, g;
    static void m() { f = 2; }
    static void testInterproc() {
        f = 3;
        m();  // (1)
        g = 4;
        m();  // (2)
    }
}

Context-sensitive analysis should give different result after (1) and (2)
The Domain and Supergraph

- **Supergraph**: `ICFGSupergraph`
  - **Procedures**: call graph nodes (`CGNode`)
    - In context-sensitive call graph, possibly many `CGNodes` for one `Imethod`
  - **Nodes**: `BasicBlockInContext<IEExplodedBasicBlock>`
    - “exploded” basic block has at most one instruction (eases writing transfer functions)
    - `BasicBlockInContext` pairs BB with enclosing `CGNode`

- **Domain (static field writes)**: `Pair<CGNode,Integer>`
  - Integer is index in IR instruction array; only valid way to uniquely identify IR instruction
  - `ReachingDefsDomain` extends `MutableMapping` to maintain fact numbering
Flow Functions (1)

- Normal flow for non-putstatics is `IdentityFlowFunction.identity()`
- Most call-related flow functions are also identity
  - Since static fields are in global scope
- Call-to-return function is `KillEverything.singleton()`
  - Defs must survive callee to reach return
Flow Functions (2)

Normal flow function for `putstatic`
(modified for formatting / clarity)

```java
public IntSet getTargets(int d1) {
    IntSet result = MutableSparseIntSet.makeEmpty();
    // first, gen this statement
    int factNum = domain.getMappedIndex(Pair.make(node, index));
    result.add(factNum);
    // if incoming statement defs the same static field, kill it;
    // otherwise, keep it
    if (d1 != factNum) { // must be different statement
        IField sf = cha.resolveField(putInstr.getField());
        Pair<CGNode, Integer> other = domain.getMappedObject(d1);
        SSAPutInstruction otherPut = getPutInstr(other);
        IField otherSF = cha.resolveField(otherPut.getField());
        if (!sf.equals(otherSF)) { result.add(d1); }
    }
    return result;
}
```
Seeds

◆ **Standard tabulation approach: special ‘0’ fact**
  - Add ‘0 -> 0’ edge to all flow functions
  - Seed with (main_entry, 0) -> (main_entry, 0)

◆ **Our approach: partially balanced tabulation**
  - For field write numbered \( n \) in basic block \( b \) of method \( m \), add seed \((m_entry, n) \rightarrow (b, n)\)  
    - (source fact doesn’t matter)
  - Unbalanced flow function is just identity
  - Advantage: keeps other flow functions cleaner
  - See `ReachingDefsProblem.collectInitialSeeds()`
Putting it all Together

- **ReachingDefsProblem** collects domain, supergraph, flow functions, seeds

- **Running analysis (simplified):**
  ```java
  PartiallyBalancedTabulationSolver solver = new PartiallyBalancedTabulationSolver(
      new ReachingDefsProblem());
  TabulationResult result = solver.solve();
  ```

- **In the real code:**
  - Lots of long generic type instantiations (sigh)
  - Handling `CancelException` (enables cancelling running analysis from GUI)
CALL GRAPHS / POINTER ANALYSIS
Call Graph Builder Overview

- **AnalysisOptions**: Specifies entrypoints, how to handle reflection, etc.
- **Heap Model**: How should objects and pointers be abstracted?
- **Context Selector**: What context to use when analyzing call to some method?

---

**CallGraphBuilder**

- **CallGraph**
- **PointerAnalysis**
Entrypoints

◆ What are entrypoint methods?
  • `main()` method
    ▪ `Util.makeMainEntrypoints()`
  • All application methods
    ▪ `AllApplicationEntrypoints`
  • JavaEE Servlet methods, Eclipse plugin entries, ...

◆ What types are passed to entrypoint arguments?
  • Just declared parameter types (`DefaultEntrypoint`)
  • Some concrete subtype (`ArgumentTypeEntrypoint`)
  • All subtypes (`SubtypesEntrypoint`)
Heap Model

- **Controls abstraction of pointers and object instances**

- **InstanceKey**: abstraction of an object
  - All objects of some type (ConcreteTypeKey)
  - Objects allocated by some statement in some calling context (AllocationSiteInNode)
  - ZeroXInstanceKeys: customizable factory

- **PointerKey**: abstraction of a pointer
  - Local variable in some calling context (LocalPointerKey)
  - Several merged vars (offline substitution), etc.
Context Selector

- Gives context to use for callee method at some call site

- **Context examples**
  - The default context (Everywhere)
  - A call string (CallStringContext)
  - Receiver object (ReceiverInstanceContext)

- **ContextSelector examples**
  - nCFAContextSelector: n-level call strings
  - ContainerContextSelector: object sensitivity for containers
Built-In Algorithms
(Grove and Chambers, TOPLAS 2001)

- **Rapid Type Analysis (RTA)**
- **0-CFA**
  - context-insensitive, class-based heap
- **0-1-CFA**
  - context-insensitive, allocation-site-based heap
- **0-1-Container-CFA**
  - 0-1-CFA with object-sensitive containers

Increasing precision

For builders, see com.ibm.wala.ipa.callgraph.impl.Util
Performance Tips

◆ Use AnalysisScope exclusions
  • Often, much of standard library (e.g., GUI libraries) is irrelevant

◆ Analyze older libraries
  • Java 1.4 libraries much smaller than Java 6

◆ Tune context-sensitivity policy
  • E.g., more sensitivity just for containers
Code Modelling (Advanced)

- **SSAContextInterpreter**: builds SSA for a method
  - Normally, based on bytecode
  - Customized for reflection, `Object.clone()`, etc.

- **MethodTargetSelector**: determines method dispatch
  - Normally, based on types / class hierarchy
  - Customized for native methods, JavaEE, etc.

- **ClassTargetSelector**: determines types of allocated objects
  - Normally, type referenced in `new` expression
  - Customized for adding synthetic fields, JavaEE, etc.
Refinement-Based Points-To Analysis

- Refines analysis precision as requested by client
  - Computes results on demand
  - See Sridharan and Bodik, PLDI 2006

- Implemented in DemandRefinementPointsTo
  - Baseline analysis is context insensitive
    - Field sensitivity, on-the-fly call graph via refinement
    - Context sensitivity (modulo recursion), other regular properties via additional state machines
  - Refinement policy can be easily customized
  - Can also compute “flows to” on demand

- Usage fairly well documented on wiki
- See DemandCastChecker for an example client
SLICING
Slicer Overview

- **Statement**: Seed for the slicer
- **DataDependenceOptions**: Which data deps to consider
- **ControlDependenceOptions**: Which control deps to consider
- **CallGraph**
- **PointerAnalysis**

```
Slicer.compute{Forward|Backward}Slice()
```

```
Collection<Statement>
```
Statement

- Identifies a node in the System Dependence Graph (SDG) [Horwitz, Reps, Binkley, TOPLAS’90]

- **Key statement types**
  - **NormalStatement**
    - Normal SSA IR instruction
    - Represented by CGNode and instruction index
  - **ParamCaller, ParamCallee**
    - Extra nodes for modeling parameter passing
    - SDG edges from def → ParamCaller → ParamCallee → use
  - **NormalReturnCaller, NormalReturnCallee**
    - Analogous to ParamCaller, ParamCallee
    - Also ExceptionalReturnCaller/Callee
  - **HeapParamCaller, HeapParamCallee, etc.**
    - For modeling interprocedural heap-based data deps
    - Edges via interprocedural mod-ref analysis
Dependence Options

◆ DataDependenceOptions
  • FULL (all deps) and NONE (no deps)
  • NO_BASE_PTRS: ignore dependencies for memory access base pointers
    ▪ E.g., exclude forward deps from defs of \( x \) to \( y=x.f \)
  • NO_HEAP: ignore dependencies to/from heap locs
  • NO_EXCEPTIONS: ignore deps from throw to catch
  • Various combinations (e.g., NO_BASE_NO_HEAP)

◆ ControlDependenceOptions
  • FULL (all deps) and NONE (no deps)
  • NO_EXCEPTIONAL_EDGES: ignore exceptional control flow
Thin Slicing

- Just “top-level” data dependencies (see [Sridharan-Fink-Bodik PLDI’07])

- For context-sensitive thin slicing, use Slicer with DataDependenceOptions.NO_BASE_PTRS and ControlDependenceOptions.NONE

- For efficient context-insensitive thin slicing, use the CISlicer class
Performance Tips

◆ Some configs do not scale to large programs
  • E.g., context-sensitive slicing with heap deps
  • Discussion in [SFB07]

◆ Run with minimum dependencies needed

◆ Apply pointer analysis scalability tips
  • Exclusions, earlier Java libraries
INSTRUMENTING BYTECODES WITH SHRIKE
Key Shrike Features

- **Patch-based instrumentation API**
  - Each instrumentation pass implemented as a patch
  - Several patches can be applied simultaneously to original bytecode
    - No worries about instrumenting the instrumentation
  - Branch targets / exc. handlers automatically updated

- **Efficient**
  - Unmodified class methods copied without parsing
  - Efficient bytecode representation / parsing
    - Array of immutable instruction objects
    - Constant instrs represented with single shared object

- **Some ugliness hidden**
  - JSRs, exception handler ranges, 64k method limit
Key Shrike Classes

**ClassReader**
Immutable view of .class file info; reads data lazily

**ClassWriter**
Generates JVM representation of a class

**MethodEditor**
Core class for transforming bytecodes via patches

**ClassInstrumenter**
Utility for instrumenting an existing class (mutable)

**MethodData**
Mutable view of method info

**CTCompiler**
Compiles ShrikeBT method into JVM bytecodes

**ShrikeCT:**
reading / writing .class files

**ShrikeBT:**
instrumenting bytecodes
Instrumenting A Method (1)

```java
instrument(byte[] orig, int i) {
    // mutable helper class
    ClassInstrumenter ci = (1) new ClassInstrumenter(orig);
    // mutable representation of method data
    MethodData md = ci.visitMethod(i); (2)
    // see next slide; mutates md, ci
doInstrumentation(md); (3)
    // output instrumented class in JVM format
    ClassWriter w = ci.emitClass(); (4)
    byte[] modified = w.makeBytes(); (5)
}
```
doInstrumentation(MethodData md) {
    // manages the patching process
    MethodEditor me = new MethodEditor(md);  \(1\)
    me.beginPass(); \(2\)
    // add patches
    me.insertAtStart(new Patch() { ... }); \(3\)
    me.insertBefore(j, new Patch() { ... }); \(4\)
    ...
    // apply patches (simultaneously)
    me.applyPatches(); me.endPass(); \(5\)
}
Shrike Clients

- Small example: see `com.ibm.wala.shrike.bench.Bench`
- Dila (`com.ibm.wala.dila` in incubator)
  - Dynamic call graph construction (CallGraphInstrumentation)
  - Utilities for runtime instrumentation
    - Instrumenting class loader
    - Mechanisms for controlling what gets instrumented
  - Work continues on better WALA integration / docs
Java Annotation Support

- **Supported features**
  - Reading .class file attributes
  - Parsing some annotation info from attributes
    - E.g., generics (com.ibm.wala.types.generics)
  - Manipulating JVM class attributes directly
    - See ClassWriter.addClassAttribute()

- **Missing features**
  - Higher-level APIs for modifying known annotations
  - Automatic fixing of StackMapTable attribute after instrumentation
    - Speeds bytecode verification in Java 6
Eclipse Support

- WALA projects are Eclipse plug-ins
  - Easy to invoke from other plug-in
- Various utilities in `com.ibm.wala.ide` project
  - `EclipseProjectPath`: creates `AnalysisScope` for Eclipse project
  - `JdtUtil`: find all Java projects, code within projects, etc.
- JDT CAst frontend for Java source analysis
- Prototype utils in `com.ibm.wala.eclipse` project
  - E.g., display call graph for selected project
FRONT ENDS / CAST
WALA Bytecode Front End

WALA IR

Shrike Bytecode To IR Translator

JVML

CIL

.NET runtime

dot.gnu

ABC

Tamarin
Shrike IR Construction

Workflow:
- Shrike Readers
- ShrikeBT
- WALA Converter
- Bytecodes
- Shrike IR
- WALA IR
Shrike Readers

ShrikeCT
- JVML (java bytecode)

GNU dot.gnu
- CIL
- (internal IBM only)

WIN32
- CIL
- (IBM; Windows only)

Mozilla Tamarin
- ABC (ActionScript)
- (under development)
CASt IR Generation

source files

ASTs from source

Instruction Generation

Control Flow Graph Creation

Source Mapping Recording

Pre IR

SSA Conversion

IR
Translate Rhino structures to WALA Common AST (CAst)
- Combination of generic and JavaScript AST nodes
- Only piece of code that understands Rhino

Translate CAst AST to WALA Pre IR form
- Shared by another internal JavaScript translator
- Extends generic translation machinery
foo(x, y) {
    return x==y || x.equals(y);
}

v3 = (v1 == v2)

If v3

V3 = invoke equals v1,v2

return v3

AstTranslator implements recursive AST tree walk
foo(x, y) {
    return x==y || x.equals(y);
}

def getBlock(x : ||) {
    b1 = getBlock(x.left);
    b2 = getBlock(x.right);
    b3 = Block()
    b1.successor = b3;
    b3.add([b1.v == true])
    b3.false = b2;
}

AstTranslator builds CFG recursively over the AST
foo(x, y) {
    return x==y || x.equals(y);
}

v3 = (v1 == v2)
[1,10] - [1,14]

If v3
[1,10] - [1,29]

V3 = invoke equals v1,v2
[1,19] - [1,29]

return v3
[1,2] - [1,29]

AstTranslator copies source positions from AST
SSA Conversion

v3 = (v1 == v2)

If v3

v3 = invoke equals v1,v2

return v3

v3 = (v1 == v2)

If v3

v4 = invoke equals

v5 = φ(v3,v4)

return v5

WALA does copy propagation in SSA conversion
WALA implements fully-pruned SSA Conversion
  - i.e. phis only inserted for live values