abc: A workbench for Aspect-Oriented Programming Language and Compiler Research

Programming Tools Group, University of Oxford, UK
Sable Research Group, McGill University, Canada
Århus University, Denmark

abc: A workbench for Aspect-Oriented Programming Language & Compilers research

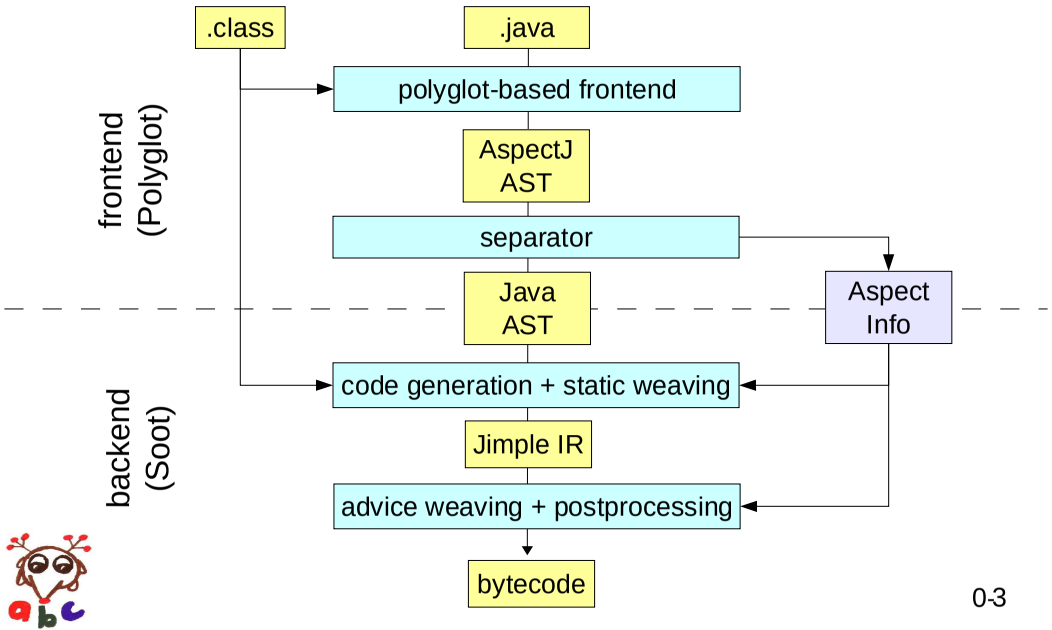
Programming Tools Group, University of Oxford, UK
Sable Research Group, McGill University, Canada
Aarhus Universitet, Denmark

The need for a research compiler

- explore AOP language design space
- experiment with better code generation
- experiment with static analyses for safety checks and optimisations

side-benefit: clarify language definition

Architecture of abc



Schedule

9:00	overview
9:10	adding syntactic extensions to AspectJ
9:55	hands-on exercises
10:05	adding new joinpoints
10:50	hands-on exercises
11:00	break + continue hands-on exercises
11:30	introduction to Soot
12:00	optimisations and analysis
12:30	close

additions to frontend only: parse, check, transform to pure AspectJ

additions to frontend, AspectInfo, Weaver, Runtime

Soot's analysis framework; cflow optimisations

Let us help you with your extensions – we're here the whole week. Possibly have a BoF / dinner to discuss common development plans?

abc: Part I
Syntactic extensions

Programming Tools Group, University of Oxford, UK
Sable Research Group, McGill University, Canada
Århus University, Denmark

Syntactic Extensions

- Introduction to Polyglot
- Example: global pointcuts

What is Polyglot?

An *extensible* Java compiler

Sample extensions:

- Jif : Java information flow and program partitioning
- PolyJ 2.0 : Java with parameterized types
- JMatch : Abstract iterable pattern matching for Java
- Jx: Nested inheritance in Java
- Jedd: BDD-based analyses
- JPred : Practical predicate dispatch

Produced by Andrew Myers, Nate Nystrom *et al.* at Cornell

How does Polyglot do it?

- Structured as a series of visitors
- Each visitor pass rewrites AST; about 15 such visitors
- Rigorous use of interfaces and factories makes it easy to change type system, environment, ...
- Delegates for overriding members of non-final AST classes (*cf.* intertype decls)

The AspectJ extension

Like any other Polyglot extension, five new packages:

- AST: new ast nodes (89 classes)
- Extension: overrides of existing Java AST nodes (13 classes)
- Parse: new lexer and grammar (2 files)
- Types: new types and type system (8 classes)
- Visit: new passes (35 classes)

- Includes Java/AspectInfo separator
- Many AST classes in pointcut language are light-weight
- The tricky bits are the type rules for ITDs, and the separator into Java & AspectInfo

plus new “driver” that controls pass order *etc*

Polyglot summary

- Extensible in all dimensions:
  - syntax, type system, visitors
- Potential merge problems with pure Java compiler only occur in extension dir and type system
- Extensions to *abc* have same structure as *abc* itself

Example extension: global pointcuts

global : <pattern> : <pc>;

add a new conjunct <pc>
to the pointcut
of each advice declaration
in any aspect that matches <pattern>

joinpoints in class C cannot be advised by aspects in subpackages of "special":

```
class C {
    global : special..* : !within(C);
    ...
}
```

aspect A only applies to classes in the package "foo.bar":

```
aspect A {
    global : A : within(foo.bar.*);
    ...
}
```

The translation we aim for

global pointcut declaration is turned into a named pointcut:

```
class C {
    pointcut  globalpc$7345() : !within(C);
    ...
}
```

every advice declaration in matching aspect gets extra conjunct in pointcut:

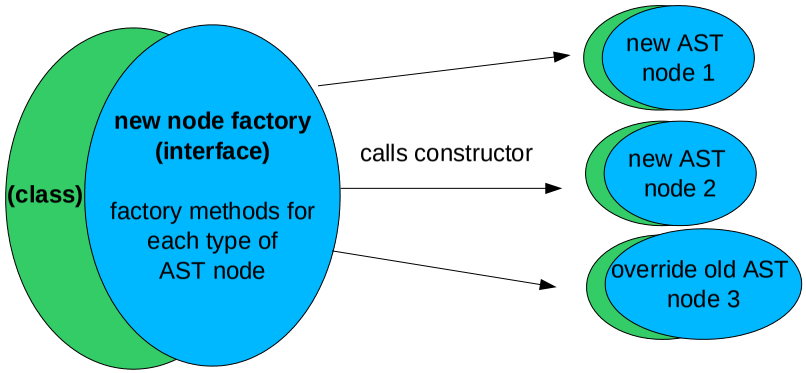
```
package special.bar.foos;
aspect Foo {
    before(int x) : call(* foo(int)) &&
                    args(x) && C.globalpc$7345()
    {...}

    after() :  call(* bar(..)) && C.globalpc$7345()
    {...}
}
```

Our task list

General	This example
• AST: new ast nodes	one for global pointcuts *GlobalPointcutDecl*
• Extension: overrides of existing AspectJ nodes	new method on AdviceDecl *EAJAdviceDecl*
• Parse: new lexer and grammar	new keyword: *global* new nonterminal new grammar rules
• Types: new types and type system	none
• Visit: new passes	to rewrite to pure AspectJ *GlobalPointcutVisitor*

extend AspectJ "driver" classes to use the above

abc node factory for an extension



each abc extension has a single node factory that extends abc.aspectj.ast.NodeFactory_c, and that implements abc.aspectj.ast.NodeFactory

global pointcut: new AST node

create in package abc.eaj.ast:	**extends in abc.aspectj.ast:**
interface GlobalPointcutDecl	PointcutDecl
class GlobalPointcutDecl_c	PointcutDecl_c
interface EAJNodeFactory	AspectJNodeFactory
class EAJNodeFactory_c	AspectJNodeFactory_c
with factory method for new node type	

contains declaration

```
public GlobalPointcutDecl GlobalPointcutDecl(
                        Position pos,
                        ClassnamePatternExpr aspect_pattern,
                        Pointcut pc)
  {
     return new GlobalPointcutDecl_c(pos, aspect_pattern, pc);
  }
```


New AST Class (1)

```
package abc.eaj.ast;

public class GlobalPointcutDecl_c extends PointcutDecl_c
                        implements GlobalPointcutDecl
{
   // pattern for aspects
   ClassnamePatternExpr aspect_pattern;
   // pointcut (declared in PointcutDecl_c)
   // Pointcut pc;
   // generated pointcut name  (declared in PointcutDecl_c)
   // String name;

   // constructor
   public GlobalPointcutDecl_c(Position pos,
                        ClassnamePatternExpr aspect_pattern,
                        Pointcut pc)
```


New AST class (2)

Polyglot implements all tree rewriting by “reconstruct” and “visitChildren” in each AST class

```
// construct new node if any of the children has changed
protected Node reconstruct(Node n, ClassnamePatternExpr aspect_pattern,Pointcut pc) ...
   (more detail on next slide)

// visit each of the children
public Node visitChildren(NodeVisitor v) ...
   (more detail two slides on)

// store (pattern,pointcut) pair in datastructure (of visitor) for later lookup during rewriting pass
public void registerGlobalPointcut(GlobalPointcuts visitor,
                        Context context,
                        EAJNodeFactory nf) ...

} // end of new AST class
```

New AST class: *reconstruct*

```
protected Node reconstruct(Node n,
                           ClassnamePatternExpr aspect_pattern,
                           Pointcut pc)
{
    // has one of the children changed?
    if (   aspect_pattern != this.aspect_pattern || pc != this.pc)
    {
        // then make a fresh copy of this node
        GlobalPointcutDecl_c new_n = (GlobalPointcutDecl_c) n.copy();
        new_n.aspect_pattern = aspect_pattern;
        new_n.pc = pc;

        return new_n;
    }
    return n;
}
```

New AST class: *visitChildren*

```
public Node visitChildren(NodeVisitor v)
{
    // visits pointcut
    Node n = super.visitChildren(v);

    ClassnamePatternExpr aspect_pattern =
          (ClassnamePatternExpr) visitChild(this.aspect_pattern, v);

    return reconstruct(n, aspect_pattern, pc);
}
```

Our task list

General	This example
✓ AST: new ast nodes	one for global pointcuts *GlobalPointcutDecl*
• Extension: overrides of existing AspectJ nodes	new method on AdviceDecl *EAJAdviceDecl*
• Parse: new lexer and grammar	new keyword: *global* new nonterminal new grammar rules
• Types: new types and type system	none
• Visit: new passes	to rewrite to pure AspectJ *GlobalPointcutVisitor*

extend AspectJ “driver” classes to use the above

Extension (override AspectJ)

```
package abc.eaj.extension;
...
public class EAJAdviceDecl_c extends AdviceDecl_c implements EAJAdviceDecl
{
    // constructor
    public EAJAdviceDecl_c(Position pos, Flags flags,
                           AdviceSpec spec, List throwTypes,
                           Pointcut pc, Block body) ...

    // add a new conjunct to the pointcut of an advice declaration
    public EAJAdviceDecl conjoinPointcutWith(GlobalPointcuts visitor, Pointcut global)
    {
        EAJAdviceDecl_c n = (EAJAdviceDecl_c) this.copy();
        n.pc = visitor.conjoinPointcuts(pc, global);
        return n;
    }
}
```

important:
rewrite must make *new copy*, not modify existing advice declaration

and override AdviceDecl factory method in EAJNodeFactory

Our task list

General	This example
✓ AST: new ast nodes	one for global pointcuts *GlobalPointcutDecl*
✓ Extension: overrides of existing AspectJ nodes	new method on AdviceDecl *EAJAdviceDecl*
• Parse: new lexer and grammar	new keyword: *global* new nonterminal new grammar rules
• Types: new types and type system	none
• Visit: new passes	to rewrite to pure AspectJ *GlobalPointcutVisitor*

extend AspectJ “driver” classes to use the above

Parse: new grammar

a new production for an existing nonterminal

```
include "../../aspectj/parse/aspectj.ppg";
package abc.eaj.parse;

terminal Token GLOBAL;

non terminal GlobalPointcutDecl global_pointcut_decl;

extend class_member_declaration ::= global_pointcut_decl:a
    ... build AST...
  ;
extend interface_member_declaration ::= global_pointcut_decl:a
    ...
  ;

global_pointcut_decl ::=
    GLOBAL:x COLON classname_pattern_expr:a COLON pointcut_expr:b
    SEMICOLON:y
    {:
      Grm.parserTrace("GLOBAL type_pattern_expr, pointcut_expr");
      RESULT = parser.nf.GlobalPointcutDecl(parser.pos(x,y), a, b);
    :};
```

The abc Lexer

many new keywords:
aspect, privileged, pointcut, proceed, target...

minimise “stealing” of keywords from Java

stateful lexer:

Java:	aspect, privileged and pointcut
aspect:	after, around, before, declare, issingleton, percflow, percflowbelow, pertarget, perthis, pointcut, proceed
pointcut:	adviceexecution, args, call, cflow, cflowbelow, error, execution, get, handler, initialization, parents, precedence, preinitialization, returning, set, soft, staticinitialization, target, throwing, warning, within, withincode

New Lexer

“Driver” class to link in the new extension (new keywords, new joinpoints...)

```
package abc.eaj;

public class AbcExtension extends abc.main.AbcExtension
{
  ...other features of this class discussed later ...

  public void initLexerKeywords(AbcLexer lexer) {
    // Add the base keywords
    super.initLexerKeywords(lexer);

    // new keyword
    lexer.addGlobalKeyword("global",    // the lexeme
            new LexerAction_c(   new Integer(abc.eaj.parse.sym.GLOBAL),
                                 // token passed to parser
                              new Integer(lexer.pointcut_state())));
                                 // enter pointcut state
  }
}
```

add keywords to lexer states with:
addJavaKeyword
addAspectJKeyword
addPointcutKeyword
addGlobalKeyword
addAspectJContextKeyword

Our task list

General	This example
✓ AST: new ast nodes	one for global pointcuts *GlobalPointcutDecl*
✓ Extension: overrides of existing AspectJ nodes	new method on AdviceDecl *EAJAdviceDecl*
✓ Parse: new lexer and grammar	new keyword: *global* new nonterminal new grammar rules
✓ Types: new types and type system	none
• Visit: new passes	to rewrite to pure AspectJ *GlobalPointcutVisitor*

extend AspectJ “driver” classes to use the above

Two new passes for global pointcuts

Pass 1:
Collect all global pointcuts, and store them in a mapping of (class pattern expression (cpe), pointcut (pc)) pairs

Pass 2:
for each aspect A:
  for each (cpe,pc) pair:
    if A matches cpe
    then for each advice declarations ad in A:
      ad.pointcut = ad.pointcut && reference to pc

one class for both visitors, store mapping in static data

New Visitor Class

```
package abc.eaj.visit;
...
public class GlobalPointcuts extends ContextVisitor
{
   // the two passes
   public final static int COLLECT = 1;
   public final static int CONJOIN = 2;
   private int pass;

   // mapping from cpe to pc
   static HashMap globalpcs = new HashMap();
   static int unmatchedCollectPasses = 0;

   // factory for constructing new nodes
   EAJNodeFactory nodeFactory;
```

Polyglot base class for visitors that need access to *context*: variables in scope, current class, *etc*.

New Visitor Class: key methods

called when entering AST node typically adjust context

```
public NodeVisitor enter(Node parent, Node n){
   if (pass == COLLECT && n instanceof GlobalPointcutDecl)
      ((GlobalPointcutDecl) n).registerGlobalPointcut(this, context(),
                                                      nodeFactory);
   return super.enter(parent, n);
}
```

called when leaving AST node this does the rewriting

```
public Node leave(Node parent, Node old, Node n, NodeVisitor v)
{
   n = super.leave(parent, old, n, v);
   if (pass == CONJOIN && n instanceof EAJAdviceDecl) {
      EAJAdviceDecl adviceDecl = (EAJAdviceDecl) n;
      PCNode aspct = PCStructure.v().getClass(context().currentClass());
      return applyMatchingGlobals(aspct, adviceDecl);
   }
   return n;
}
```

New visitor class: apply global pcs

```
protected EAJAdviceDecl applyMatchingGlobals(PCNode aspct,
                                             EAJAdviceDecl ad)
{
    Iterator i = globalpcs.keySet().iterator();

    while (i.hasNext()) {
        ClassnamePatternExpr pattern = (ClassnamePatternExpr) i.next();

        if (pattern.matches(aspct)) {
            Pointcut global = (Pointcut) globalpcs.get(pattern);
            ad = ad.conjoinPointcutWith(this, global);
        }
    }

    return ad;
}
```

Our task list

General	This example
✔ AST: new ast nodes	one for global pointcuts *GlobalPointcutDecl*
✔ Extension: overrides of existing AspectJ nodes	new method on AdviceDecl *EAJAdviceDecl*
✔ Parse: new lexer and grammar	new keyword: *global* new nonterminal new grammar rules
✔ Types: new types and type system	none
✔ Visit: new passes	to rewrite to pure AspectJ *GlobalPointcutVisitor*

extend AspectJ “driver” classes to use the above

Driving *abc*

abc.eaj.AbcExtension. For registering

- new keywords
- new shadow and joinpoint types
- extended runtime

abc.eaj.ExtensionInfo:

- create lexer, parser, type system,...
- introduce new passes
- insert new passes between existing ones

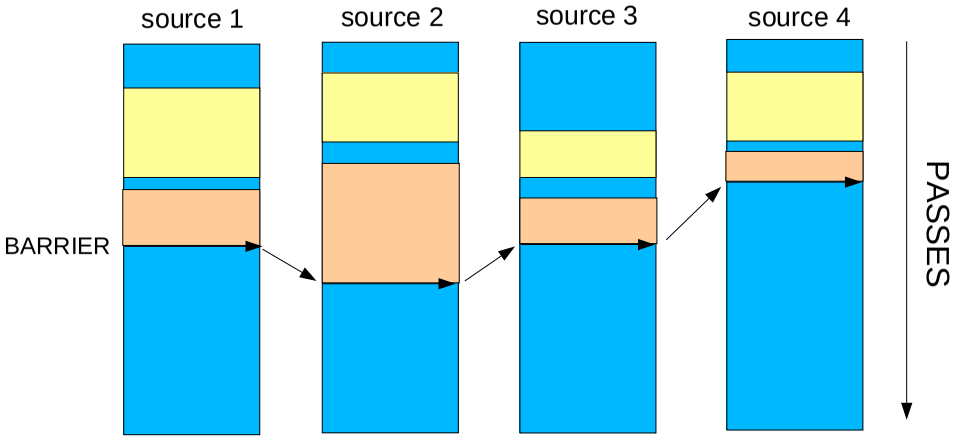
abc pass structure

new global pc passes go here

parse_and_clean	parse and build initial class hierarchy
patterns_and_parents	eval patterns; declare parents; eval patterns
precedence_relation	compute aspect precedences
disambiguate_signatures	inner/outer classes and method signatures
add_members	populate types with members
interface_ITDs	propagate interface ITDs to implementors
disambiguate_all	resolve all variable/method references
fold_and_checkcode	type checking; reachability etc
saveAST	make Soot connection
mangle_names	mangle names of private ITDs etc
aspectj_transforms	separate pure Java and AspectInfo
passes_jimple	mapping from classes to ASTs for Jimple

complex extension passes go after semantic checks

Barrier Passes



pass sequence is run independently for each source file;
barrier passes serve to ensure all files have reached the same stage before proceeding

Insert new passes

```
protected void passes_patterns_and_parents(List l, Job job)
  {
    super.passes_patterns_and_parents(l, job);

    l.add(new VisitorPass(COLLECT_GLOBAL_POINTCUTS,
                          job,
                          new GlobalPointcuts(GlobalPointcuts.COLLECT,
                                              job,
                                              (EAJTypeSystem) ts,
                                              (EAJNodeFactory) nf)));
    l.add(new GlobalBarrierPass(COLLECTED_GLOBAL_POINTCUTS, job));
    l.add(new VisitorPass(CONJOIN_GLOBAL_POINTCUTS,
                          job,
                          new GlobalPointcuts(GlobalPointcuts.CONJOIN,
                                              job,
                                              (EAJTypeSystem) ts,
                                              (EAJNodeFactory) nf)));
  }
```

barrier passes ensure that all source files reach this stage before proceeding

Our task list

General	This example
✔ AST: new ast nodes	one for global pointcuts *GlobalPointcutDecl*
✔ Extension: overrides of existing AspectJ nodes	new method on AdviceDecl *EAJAdviceDecl*
✔ Parse: new lexer and grammar	new keyword: *global* new nonterminal new grammar rules
✔ Types: new types and type system	none
✔ Visit: new passes	rewrite to pure AspectJ *GlobalPointcutVisitor*

it's as easy as abc!

Exercise: surround advice

runs before and after the joinpoint

surround(<formals>) : <pc> {beforeadvice}{afteradvice}

transforms to two pieces of normal advice

before(<formals>) : <pc> {beforeadvice}
after(<formals>) : <pc> {afteradvice}

consider advice precedence rules...

✔ extend AspectJ "driver" classes to use the above

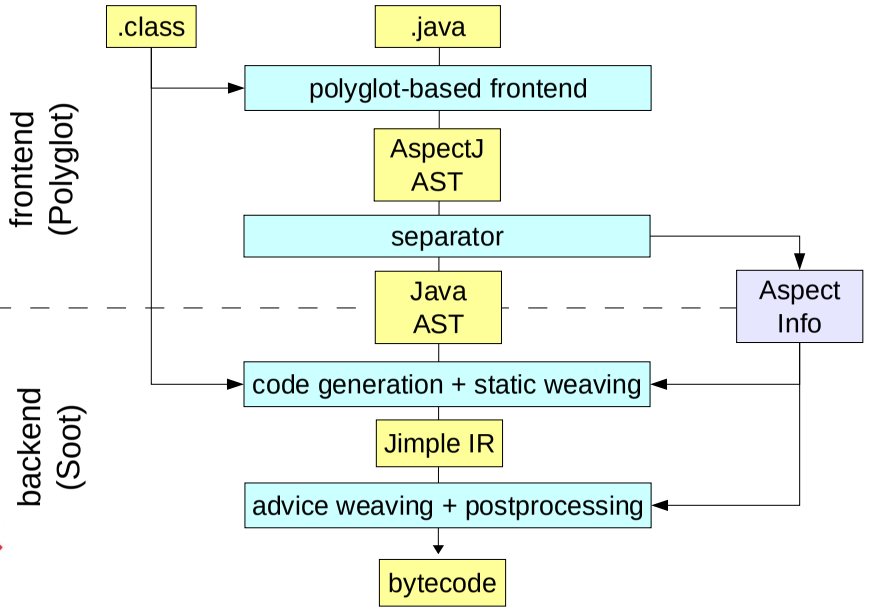
abc: Part II
Adding new joinpoints

Programming Tools Group, University of Oxford, UK
Sable Research Group, McGill University, Canada
Århus University, Denmark

Adding new joinpoints

- AspectInfo
- matcher
- weaver

Architecture of abc



Example Java/aspects separation

```
aspect Aspect {
 private int C.count = 0;
 private void C.inc() {
       count++;
 }
 before(C x) : call(* foo(..)) &&
                target(x) {
       x.inc();
 }
}
```

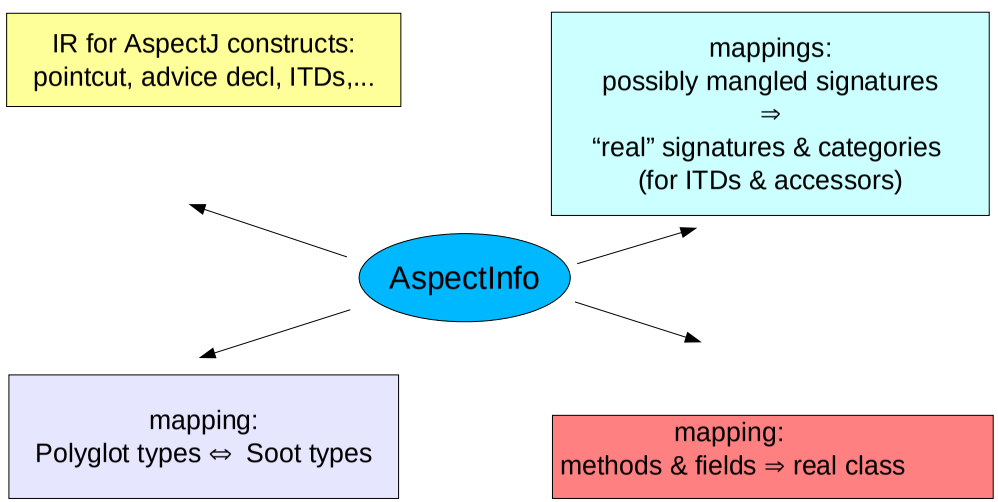
execution joinpoint:
- real name "inc"
- real parameters *none*
- real class "C"

```
public class Aspect {
  public static void Aspect$inc$5(final C this$6) {
    this$6.Aspect$count$3++;
  }

  public final void before$2(C x) { x.Aspect$inc$5(); }

  public Aspect() { super(); }

  public static int init$count$7(final C this$4) {
   return (int) 0; }

  public static Aspect aspectOf()
    throws
    org.aspectj.lang.NoAspectBoundException {
     return null;        // to be filled in later
  }

  public static boolean hasAspect() { return true; }
}
```

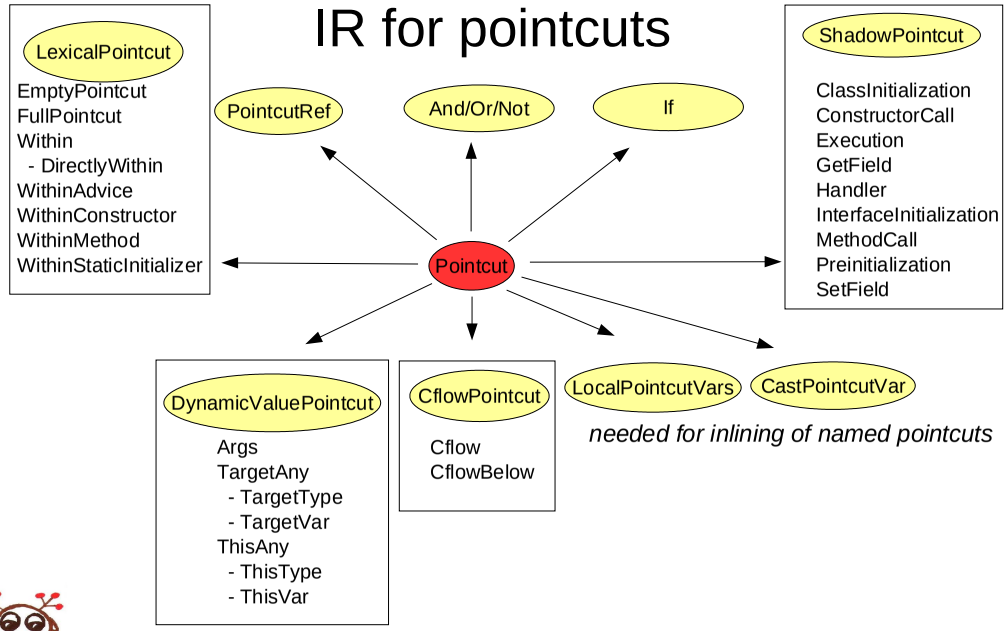
```
* intertype method inc
      host C
      implementation Aspect$inc$5
* before advice, pointcut
      call(* foo(..)) && target(x)
      method for body before$2(C x)
* intertype field count,
      host C
      initial value from init$count(C)
```

all the information that is not Java goes into AspectInfo

What's in AspectInfo?



IR for pointcuts

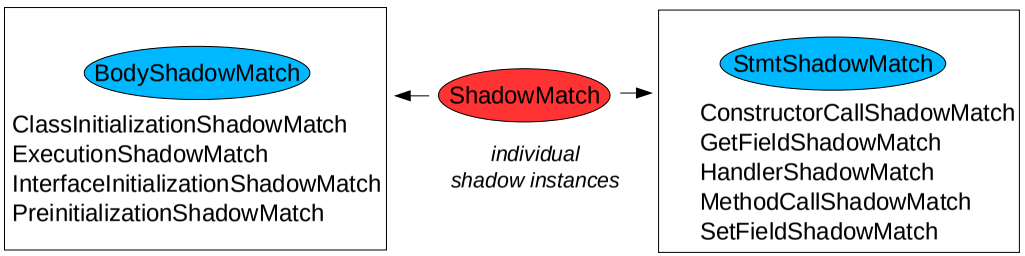


needed for inlining of named pointcuts

Examples: AspectJ pointcut to IR

AspectJ	**Intermediate Representation**
execution(int Foo.foo(char))	withinmethod(int Foo.foo(char)) && execution()
adviceexecution()	withinadvice() && execution()
call(Foo.new(int))	constructorcall(Foo.new(int))
initialization(Foo.new(..))	(withinconstructor(Foo.new(..)) && classinitialization()) || interfaceinitialization(Foo)

Shadow Kinds



individual shadow instances

for blocks of Jimple code

for single Jimple statements (or a pair, for constructor call)

and a singleton class for each kind named
<*kind*>ShadowType (a subtype of ShadowType)
with a method
ShadowMatch matchesAt(pos)
singleton instance of <*kind*>ShadowType created by
static <*kind*>ShadowMatch.newType()

Computing Advice Lists (simplified)

```
for each weavable class C
    for each method M in C
        for each "position" pos in M
            for each shadow type t
                ShadowMatch sm = t.matchesAt(pos);
                for each advice declaration ad
                    Pointcut pc = ad.getPointcut();
                    Residue r = pc.matchesAt(sm);
                    add (pos,ad,r) to "advice list" of M;
```

"positions" are defined in AbcExtension

Residue is an IR of the dynamic test to be inserted (e.g. for if, args, ...)

advice lists are applied in a separate weaving pass

[for non-statement shadows, pc.matchesAt(..) takes more parameters]

Example: Cast Pointcut

```
before(int i)
  cast(short) && args(i)
  && if (i < Short.MIN_VALUE   ||  i > Short.MAX_VALUE)
{
  System.err.println( “Warning: loss of “ +
                                “precision casting “ +
                                i   +  `` to a short.");
}
```

general syntax:

basic_pointcut_expr ::=
    PC_CAST LPAREN type_pattern_expr RPAREN

Our task list

General	This example
• extend parser, new AST nodes	as before...
• AspectInfo: IR for aspect-specific features	new class for cast pointcut
• Matcher: extend shadow matcher	how to match new pointcut
• New runtime	dynamic representation of cast joinpoint
• Extend driver classes to use new runtime	AbcExtension

New AST node

```
package abc.eaj.ast;
...
public class PCCast_c extends Pointcut_c
                      implements PCCast
{
   protected TypePatternExpr type_pattern;

   ... reconstruct, visitChildren ...

  // translate to AspectInfo
   public abc.weaving.aspectinfo.Pointcut makeAIPointcut()
   {
      return new abc.eaj.weaving.aspectinfo.Cast
          (type_pattern.makeAITypePattern(), position());
   }
}
```

Our task list

General	This example
✓ extend parser, new AST nodes	as before...
• AspectInfo: IR for aspect-specific features	new class for cast pointcut
• Matcher: extend shadow matcher	how to match new pointcut
• New runtime	dynamic representation of cast joinpoint
• Extend driver classes to use new runtime	AbcExtension

New AspectInfo Class

```
package abc.eaj.weaving.aspectinfo;
...
public class Cast extends ShadowPointcut
{
   private TypePattern pattern;
...
   protected Residue matchesAt(ShadowMatch sm)
   {
      if (!(sm instanceof CastShadowMatch)) return NeverMatch.v();
      Type cast_to = ((CastShadowMatch) sm).getCastType();

      if (!getPattern().matchesType(cast_to)) return NeverMatch.v();
      return AlwaysMatch.v();
   }
...
}
```

next step: define CastShadowMatch

Our task list

General	This example
✓ extend parser, new AST nodes	as before...
✓ AspectInfo: IR for aspect-specific features	new class for cast pointcut
• Matcher: extend shadow matcher	how to match new pointcut
• New runtime	dynamic representation of cast joinpoint
• Extend driver classes to use new runtime	AbcExtension

Shadow Matcher (1)

```
package abc.eaj.weaving;
...
public class CastShadowMatch extends StmtShadowMatch
{
   ..
   public static CastShadowMatch matchesAt(MethodPosition pos)
   {
      if (!(pos instanceof StmtMethodPosition)) return null;

      Stmt stmt = ((StmtMethodPosition) pos).getStmt();
      if (!(stmt instanceof AssignStmt)) return null;
      Value rhs = ((AssignStmt) stmt).getRightOp();
      if(!(rhs instanceof CastExpr)) return null;

      Type cast_to = ((CastExpr) rhs).getCastType();
      return new CastShadowMatch(pos.getContainer(), stmt, cast_to);
   }
```


Shadow Matcher (2)

```
public static ShadowType shadowType()
{
   return new ShadowType() {
      public ShadowMatch matchesAt(MethodPosition pos) {
         return CastShadowMatch.matchesAt(pos);
      }
   };
}

public List /*<ContextValue>*/ getArgsContextValues()
... return singleton of value being cast ...

public ContextValue getTargetContextValue()
... return null ...

public SJPInfo makeSJPInfo()
... make the relevant static joinpoint info ...
```

Our task list

General	This example
✓ extend parser, new AST nodes	as before...
✓ AspectInfo: IR for aspect-specific features	new class for cast pointcut
✓ Matcher: extend shadow matcher	how to match new pointcut
• New runtime	dynamic representation of cast joinpoint
• Extend driver classes to use new runtime	AbcExtension

New Runtime (1)

```
package org.aspectbench.eaj.lang.reflect;

import org.aspectj.lang.Signature;

public interface CastSignature extends Signature
{
   public Class getCastType();
}
```

obvious implementation in
package org.aspectbench.eaj.runtime.reflect.CastSignatureImpl

New Runtime (2)

```
package org.aspectbench.eaj.runtime.reflect;

...

public class EajFactory extends Factory
{
   ...

   public CastSignature makeCastSig(String stringRep) {
      CastSignatureImpl ret = new CastSignatureImpl(stringRep);
      ret.setLookupClassLoader(lookupClassLoader);
      return ret;
   }

   ...
}
```

Our task list

General	This example
✓ extend parser, new AST nodes	as before...
✓ AspectInfo: IR for aspect-specific features	new class for cast pointcut
✓ Matcher: extend shadow matcher	how to match new pointcut
✓ New runtime	dynamic representation of cast joinpoint
• Extend driver classes to use new runtime	AbcExtension

Use new runtime

```
package abc.eaj;
...
public class AbcExtension extends abc.main.AbcExtension
{
  ...
   // create list of shadow types that the weaver will iterate over
   protected List/*<ShadowType>*/ listShadowTypes()
   {
      List/*<ShadowType*/ shadowTypes = super.listShadowTypes();
      shadowTypes.add(CastShadowMatch.shadowType());
      return shadowTypes;
   }

   public String runtimeSJPFactoryClass() {
      return "org.aspectbench.eaj.runtime.reflect.EajFactory";
   }

  ...
}
```

Our task list

It's as easy as abc!

General	This example
✔ extend parser, new AST nodes	as before...
✔ AspectInfo: IR for aspect-specific features	new class for cast pointcut
✔ Matcher: extend shadow matcher	how to match new pointcut
✔ New runtime	dynamic representation of cast joinpoint
✔ Extend driver classes to use new runtime	AbcExtension

Exercise: array pointcuts

```
Bar around(Bar array[], int index) :
    arrayget() && target(array) && args(index) {
        Bar value=proceed(array,index);
        return value;
}
void around (Bar array[], Bar value, int index) :
    arrayset() && target(array) && args(value, index) {
        proceed(array,value,index);
}
```

general syntax:

```
basic_pointcut_expr ::=
    PC_ARRAYGET LPAREN RPAREN
  | PC_ARRAYSET LPAREN RPAREN
```

abc: Part III
Using Soot in the abc backend

Programming Tools Group, University of Oxford, UK
Sable Research Group, McGill University, Canada
Århus University, Denmark

Using Soot in the abc backend

(See also: http://www.sable.mcgill.ca/soot)

Outline

- Introduction to Soot
  - What is Soot? What can it do?
  - Why do we use Soot in **abc**?
  - Soot's intermediate representations.
  - Intra-procedural analysis framework (analysis within method bodies).
  - Call graphs, points-to analysis and inter-procedural analysis.
  - Soot's **dava** decompiler.
- Using Soot for cflow optimizations in **abc**.

What is Soot? What can it do?

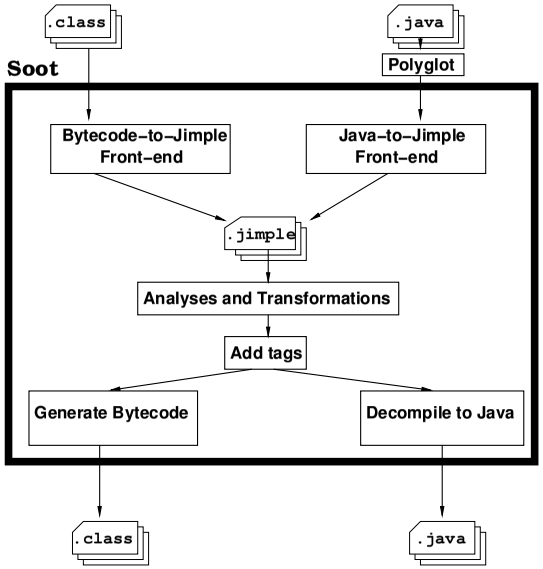
- Soot is a framework for analysing and transforming Java.
- Developed at McGill since 1998 - has been used in many research projects.
- Key features that Soot provides are:
  - Convenient IRs (mainly Jimple);
  - Existing analyses and transformations;
  - Framework for new analyses, transformations, code generation;
  - **Dava** decompiler;

www.sable.mcgill.ca/publications/

Why do we use Soot in abc?

- Provides a convenient basis for weaving, since its IR (Jimple) is simple, typed and stack-less.
- Use the existing bytecode optimizations to optimize the output of the weaver (**abc** has **-O** options).
- Develop AspectJ-specific analyses and transformations.
- Use existing whole program analyses and points-to frameworks to reason about call graphs and pointer/alias analyses.
- Enable efficient implementations of new, sophisticated pointcuts.

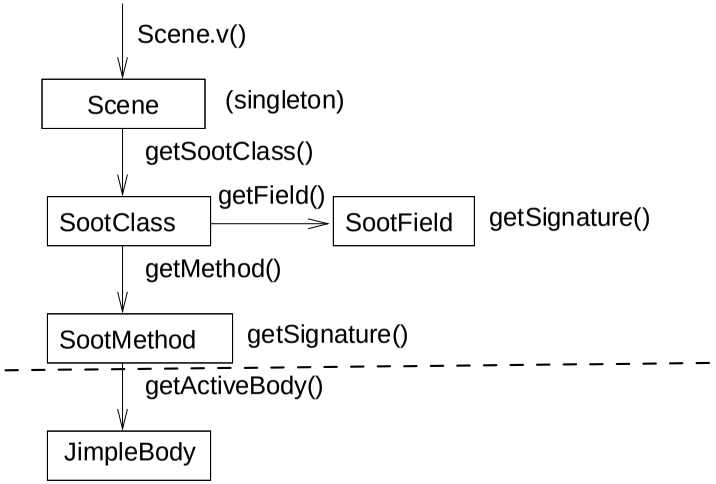
Soot



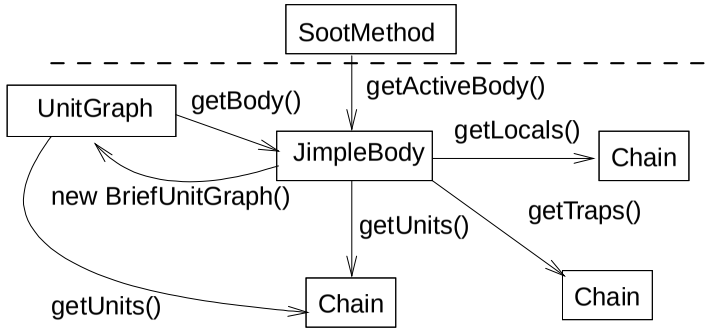
Soot Data Structure Basics

- Soot builds data structures to represent:
  - a complete environment (**`Scene`**)
  - classes (**`SootClass`**)
  - Fields and Methods (**`SootMethod`**, **`SootField`**)
  - bodies of Methods (come in different flavours, corresponding to different IR levels, ie. **`JimpleBody`**)
- These data structures are implemented using OO techniques, and designed to be easy to use and generic where possible.

Soot Classes



Body-centric View



Jimple

Jimple is:

- principal Soot Intermediate Representation
- 3-address code in a *control-flow graph*
- a *typed* intermediate representation
- *stackless*

Raja's Thesis, CASCON99, CC2000, SAS2000

Jimple example

Java:

```
public int bar(int a, int b) {
    return a+b;
}
```

Jimple:

```
public int bar(int, int) {
    Foo this;
    int a, b, $i0;

    this := @this;
    a := @parameter0;
    b := @parameter1;
    $i0 = a + b;
    return $i0;
}
```

Converting bytecode → Jimple → bytecode

- These transformations were relatively hard to design so that they produce correct, useful and efficient code.
- Worth the price, we **do** want a 3-addr typed IR.

raw bytecode	typed 3-address code (Jimple)
◦ each inst has implicit effect on stack	◦ each stmt acts explicitly on named variables
◦ no types for local variables	◦ types for each local variable
◦ $>$ 200 kinds of insts	◦ only 15 kinds of stmts

Bytecode → Jimple

- Naive translation from bytecode to untyped Jimple, using variables for stack locations.
- splits DU-UD webs (so many different uses of the stack do not interfere)
- types locals (SAS 2000)
- cleans up Jimple

Java → Jimple

- Input: Polyglot AST generated from .java sources
- Compile AST to Jimple
- Generate Jimple methods/classes for implicit Java features (initializers, inner class accessor methods, class literals, assertions)
- Output: Jimple to be analyzed/optimized, eventually converted to bytecode
- Combination of Polyglot, Java-to-Jimple, Jimple-to-Bytecode passes forms a complete Java compiler equivalent to javac.

This part is unchanged in abc.

Jimple → Bytecode

- A naive translation introduces many spurious stores and loads.
- Two approaches (CC 2000),
  - aggregate expressions and then generate stack code; or
  - perform store-load and store-load-load elimination on the naive stack code.

Weaving example – source

```
public class Foo {
    public int foo(int x, int y, int z) {
       return bar(x, y, z);
    }
    public int bar(int a, int b, int c) {
        return a+b+c;
    }
}
aspect A {
    before(Foo x) :
    call(int bar(int,int,int)) && target(x) {
        System.out.println(x);
    }
}
```

Weaving example – original bytecode

```
public int foo(int x, int y, int z)
0:    aload_0
1:    iload_1
2:    iload_2
3:    iload_3
4:    invokevirtual     Foo.bar (III)I (7)
7:    ireturn
```

Weaving example – weaving by hand

```
public int foo(int x, int y, int z)
0:    invokestatic      A.aspectOf ()LA; (14)
3:    aload_0
4:    invokevirtual     A.before$0 (LFoo;)V (20)
7:    aload_0
8:    iload_1
9:    iload_2
10:   iload_3
11:   invokevirtual     Foo.bar (III)I (9)
14:   ireturn
```

Weaving example – ajc weaving

```
public int foo(int x, int y, int z)
0:    aload_0
1:    iload_1
2:    iload_2
3:    iload_3
4:    istore            %4
6:    istore            %5
8:    istore            %6
10:   astore            %7
12:   invokestatic      A.aspectOf ()LA; (52)
15:   aload             %7
17:   invokevirtual     A.ajc$before$A$124 (LFoo;)V (56)
20:   aload             %7
22:   iload             %6
24:   iload             %5
26:   iload             %4
28:   invokevirtual     Foo.bar (III)I (37)
31:   ireturn
```

Weaving example – original Jimple

```
public int foo(int, int, int)
{
    Foo this;
    int x, y, z, $i0;

    this := @this;
    x := @parameter0;
    y := @parameter1;
    z := @parameter2;
    $i0 = this.bar(x, y, z);
    return $i0;
}
```

Weaving example – woven Jimple

```
public int foo(int, int, int)
{
    Foo this;
    int x, y, z, $i0;
    A theAspect;

    this := @this;
    x := @parameter0;
    y := @parameter1;
    z := @parameter2;
    theAspect = A.aspectOf();
    theAspect.before$0(this);
    $i0 = this.bar(x, y, z);
    return $i0;
}
```

Weaving example – bytecode from Jimple

```
public int foo(int x, int y, int z)
0:    invokestatic      A.aspectOf ()LA; (14)
3:    aload_0
4:    invokevirtual     A.before$0 (LFoo;)V (20)
7:    aload_0
8:    iload_1
9:    iload_2
10:   iload_3
11:   invokevirtual     Foo.bar (III)I (9)
14:   ireturn
```

Weaving details do matter!

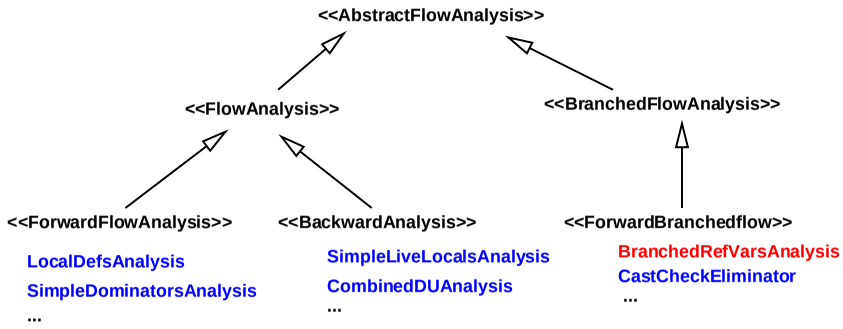
`lawOfDemeter.objectform.Pertarget.fieldIdentity`:

- ajc 1.2.0 used too many locals, too many cflow stacks.
  - ajc1.2.0: 616 locals 41 m, 38 s
  - ajc1.2.0+Soot: 3 locals 2 m, 56 s
- ajc 1.2.1 integrated abc's idea of cflow counters, and removed bug that generated too many cflow stacks.
  - ajc1.2.1: 38 locals 12.2 s
  - ajc1.2.1+Soot: 3 locals 12.2 s
- abc uses Soot's packing of locals, optimises cflow counters (still to come)
  - abc: 3 locals 8.7 seconds

Adding an intra-proc analysis/transform

- Soot provides:
  - An analysis framework.
  - Various control flow graph builders.
  - Various implementations of `FlowSet`.
  - Packs associated with different phases of Soot.
  - A large number of analyses and transforms already implemented.
- To add new analysis/transforms you can:
  - Extend an existing analysis and/or transform and add to a Pack (will see an example later); or
  - Implement a new analysis and/or transform and add to a Pack.

The `FlowAnalysis` Hierarchy



soot.toolkits.scalar

What you must provide...

A checklist of your obligations:

1. Subclass `*FlowAnalysis`
2. Implement abstraction: `merge()`, `copy()`
3. Implement flow function `flowThrough()`
4. Implement initial values: `newInitialFlow()` and `entryInitialFlow()`
5. Implement constructor
   (it must call `doAnalysis()`)

What Soot provides ...

- impls of abstraction domains (flow sets)
  - standard abstractions trivial to implement;
- an implemented solver, namely,
  - `doAnalysis()` method: executes intraprocedural analyses on a CFG using a worklist algorithm.

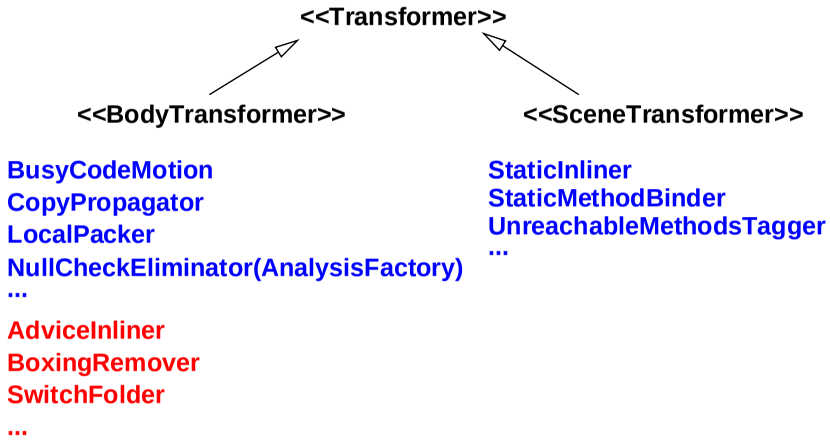
Enjoy: Flow Analysis Results

You can instantiate an analysis and collect results:

```
LiveVariablesAnalysis lv =
    new LiveVariablesAnalysis(g);

// retrieve analysis results for a stmt s:
lv.getFlowBefore(s);
lv.getFlowAfter(s);
```

Now you can use the results of your analysis in a transformation.

The `Transform` Hierarchy

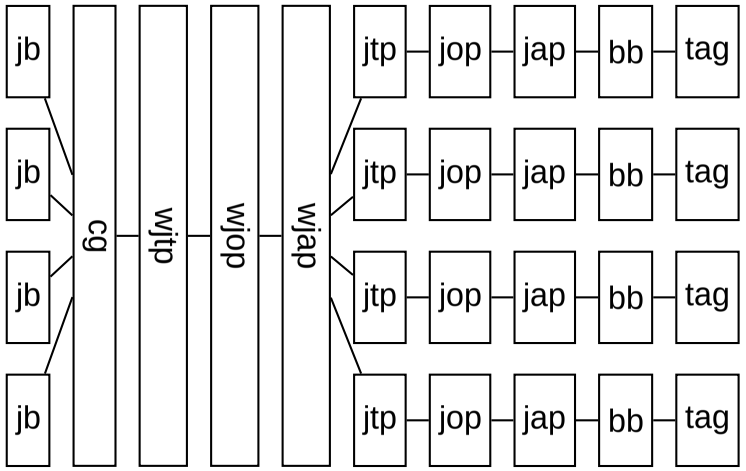


Adding transformations to Soot (easy way)

1. Implement a `BodyTransformer` or a `SceneTransformer`
   - `internalTransform` method does the transformation
2. Choose a pack for your transformation (usually `jtp`)
3. Add your transform to the pack.

```
Pack jtp = G.v().PackManager().getPack("jtp");
jtp.add(new Transform("jtp.nt",
                      new NullTransformer()));
```


Soot phases



Whole-program analyses

- CHA call graph based on class hierarchy
- VTA – more precise call graph (OOPSLA2000)
- Spark (stable): context-ins. points-to, call graph and side-effect analysis (CC2003)
- Paddle (current research): BDD-based framework for context-sensitive:
  - points-to analysis
  - call graph analysis
  - cflow analysis
  - type analysis (`instanceof` checks)
  - side-effect analysis (aspect purity)
  - escape analysis (`thisJoinPoint[StaticPart]`)

Soot has the SPARK points-to toolkit

If you want to know
... the types of the receiver o in the call: `o.m(...)`

1. Enable `cg.spark` so that points-to sets are computed.
2. Then use points-to information.

```
Local o;
PointsToAnalysis pa =
     Scene.v().getPointsToAnalysis();
PointsToSet ptset = pa.reachingObjects(o);
java.util.Set types = ptset.possibleTypes();
```

Call Graph

- Collection of edges representing all method invocations known to Soot
  - explicit method invocations
  - implicit invocations of static initializers
  - implicit calls of `Thread.run()`
  - implicit calls of finalizers
  - implicit calls by `AccessController`
  - ...
- `Filter` can be used to select specific kinds of edges

Call Graph Edges

- Each `Edge` contains
  - Source method
  - Source statement (if applicable)
  - Target method
  - Kind of edge

source m.	source stmt.	target m.	kind
●	●	●	VIRTUAL

```
foo() {                    bar() {
   o.bar();-------------->    /* */
}                          }
```

Querying Call Graph

`edgesOutOf(SootMethod)` iterator over edges with given source method

`edgesOutOf(Unit)` iterator over edges with given source statement

`edgesInto(SootMethod)` iterator over edges with given target method

main()	o.foo();	C1.foo()	VIRTUAL
main()	o.goo();	C1.goo()	VIRTUAL
main()	o.goo();	C2.goo()	VIRTUAL
bar()	o.foo();	C2.foo()	VIRTUAL

soot.jimple.toolkits.callgraph.CallGraph

Code Example

```
void mayCall( SootMethod src ) {
   CallGraph cg =
            Scene.v().getCallGraph();
   Iterator targets =
     new Targets(cg.edgesOutOf(src));

   while( targets.hasNext() ) {
      SootMethod tgt =
         (SootMethod) targets.next();
      System.out.println( ""+
        src+" may call "+tgt );
   }
}
```

Dava decompiler

```
public int foo(int x, int y, int z)
{
    A.aspectOf().before$0(this);
    return this.bar(x, y, z);
}
```

- Dava decompiles bytecode with strange aspect-generated control flow that breaks other decompilers.
- Dava is integrated with Soot and **abc**, use **abc `-dava <args>`** and decompiled output will be placed in a subdirectory called `dava/`.
- Very useful for debugging weavers.

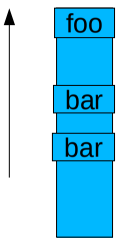
Optimisations in abc

- Intraprocedural Analyses using Soot
- Interprocedural Analysis
  - Analysing AspectJ Programs
  - Optimising *cflow*

Running Example: Optimising *cflow*

```
pointcut fooFromBar(int x) :
    call(* foo()) &&
    cflow( call(* bar(*)) && args(x) )
```

call stack:



bind x to the argument of the last call to bar

Obvious implementation:
maintain thread-local stack of bindings
- push before each call to bar
- pop after each call to bar
- check top upon each call to foo

Simple optimisations

no variable binders?
use an integer counter instead of stack

share stacks for multiple pointcuts:
e.g. unify cflows in
call(* bar(..)) && **cflow** (**call**(* foo(..)) && **args**(t,*,*))
call(* bar(..)) && **cflow** (**call**(* foo(..)) && **args**(*,s,*))
to
cflow(**call**(* foo(..)) && **args**(x,y,*))

Especially useful because of *inlining* of named pointcuts

Simple IR transformations: no need for dataflow analysis

Intraprocedural Optimisations

each cflow stack is local to a thread
want to retrieve it just once per method body
but never get the stack if it is not used

```
CflowStack threadLocalStack = null;
...
if (threadLocalStack == null)
    threadLocalStack = cflow.getThreadStack();
```

rely on Soot's null-check eliminator to remove this

getThreadStack() never returns null

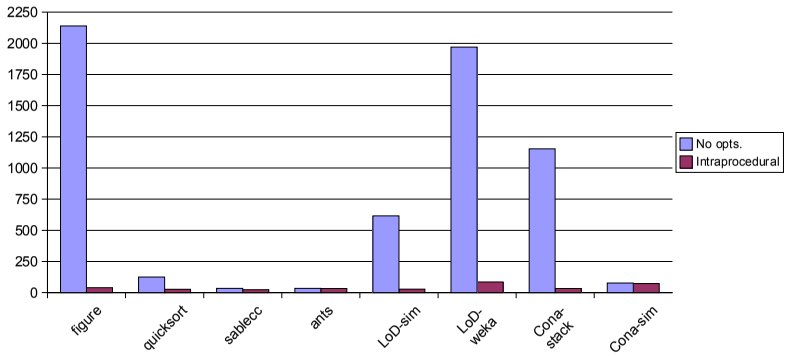
Intraprocedural Optimisations (2)

```
NullCheckEliminator.AnalysisFactory f =
   new NullCheckEliminator.AnalysisFactory() {
      (...)

         public boolean isAlwaysNonNull(Value v) {
            // Standard null-check eliminator
            if (super.isAlwaysNonNull(v)) return true;

            if (v instanceof InvokeExpr) {
               InvokeExpr ie = (InvokeExpr) v;
               SootMethodRef m = ie.getMethodRef();
               return CflowCodeGenUtils.isCflowGetMethod(m); }
            return false; }

   };
   // this should run before Dead Assignment Elimination
   PackManager.v().getPack("jop").insertBefore(
   new Transform("jop.nullcheckelim", new NullCheckEliminator(f)), "jop.dae");
```

Factory for constructing customised Null-Check eliminators

cflow-specific

Intraprocedural Optimisations: Summing Up



We have reduced, but not eliminated, overheads for cflow
Similar speedup for ajc 1.2 vs. 1.2.1 (some of the same opts)

see PLDI 2005 for detailed results

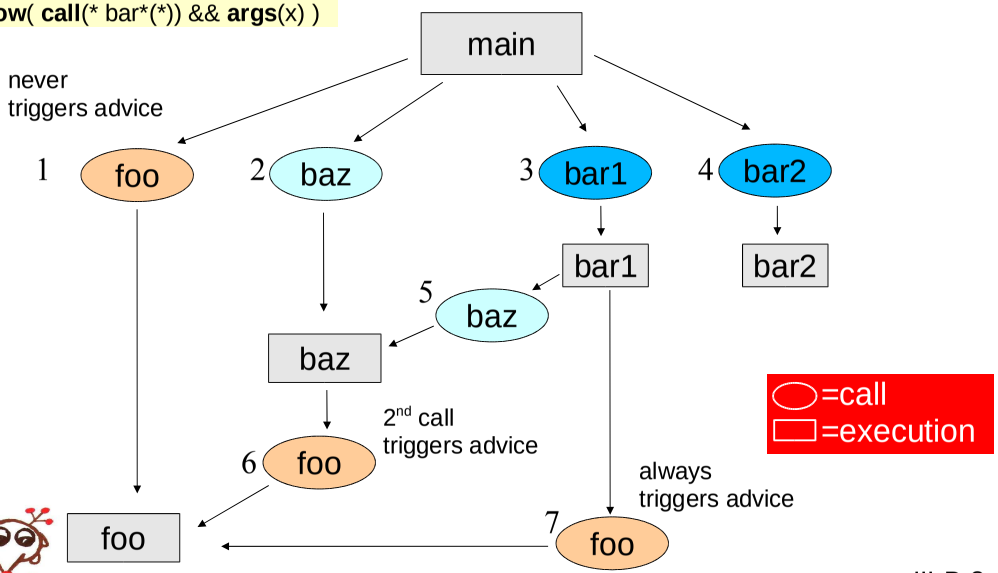
Optimising *cflow*

```
aspect Aspect {
   pointcut fooFromBar(int x) :
      call(* foo()) &&
      cflow( call(* bar*(*)) && args(x) );

   before(int x) : fooFromBar(x) {
      System.out.println("foo from bar, x="+x);
   }
}
```

```
public class Cflow {

   void foo() {}
   void bar1(int x) { foo(); baz(); }
   void bar2(int x) {}
   void baz() { foo(); }

   public static void main(String[] args) {
      Cflow c = new Cflow();
      c.foo();
      c.baz();
      c.bar1(3);
      c.bar2(4);
   }

}
```

Call Graph

```
call(* foo()) &&
cflow( call(* bar*(*)) && args(x) )
```



Interprocedural Optimisations

to implement cflow(p)

update shadow:
push/pop stack at each shadow matching p

query shadow:
test whether stack nonempty
bind formals from stack (if applicable)

at query shadow:
predict emptiness:
if yes or no, remove test

at update shadow:
predict whether observed by any query:
if not, remove push/pop

Optimising *cflow*

For each update shadow sh:

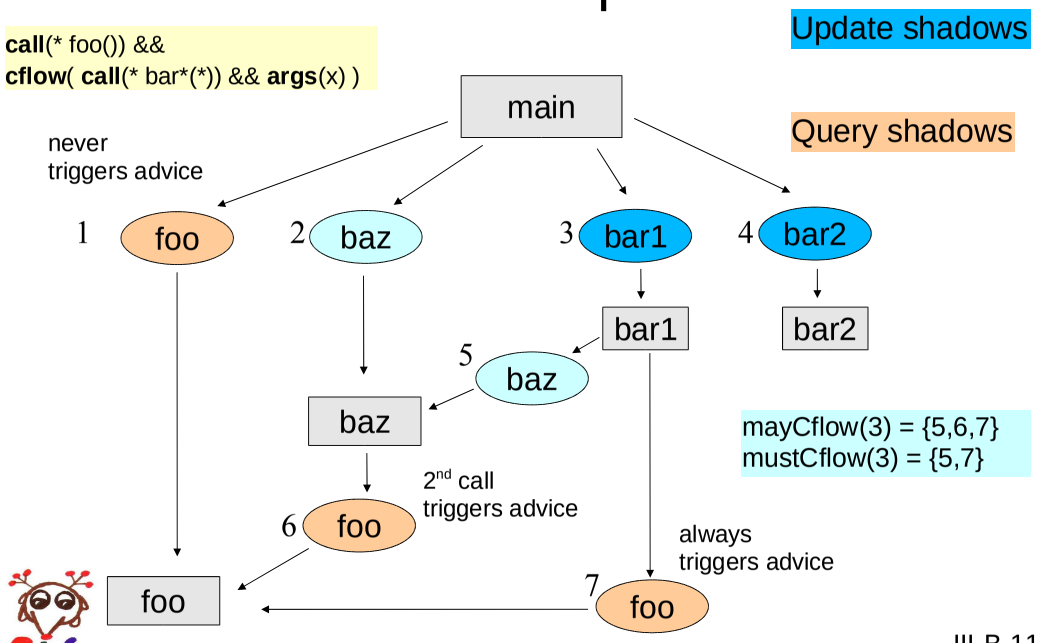
st ∈ *mayCflow*(sh) :
at statement st, we *may* be in the dynamic scope of sh

st ∈ *mustCflow*(sh) :
at statement st, we *must* be in the dynamic scope of sh

The set of statements that can be reached interprocedurally from sh

Call Graph

call(* foo()) &&
cflow(**call**(* bar*(*)) && **args**(x))



Optimising *cflow* (2)

At query shadow qsh:

If ¬(∃ upsh : qsh ∈ *mayCflow*(upsh))
qsh is *always false* and can be removed
If (∃ upsh : qsh ∈ *mustCflow*(upsh))
qsh is *always true* and can be removed

Only valid if no variables are bound

At update shadow upsh:

If ¬(∃ qsh ∈ mayCflow(upsh))
upsh is *never queried* and can be removed
If (∃ sh : upsh ∈ mustCflow(sh))
upsh is *guaranteed nonempty* and can be removed

Computing *cflow* information

computation of mayCflow(sh):

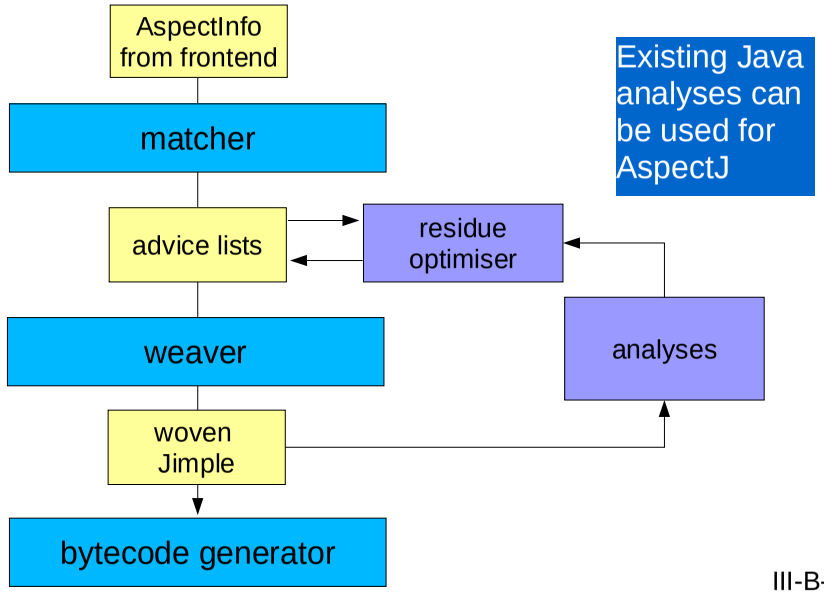
mayCflow ← { st | st is in intraprocedural shadow of sh}
repeat
  for all methods m | ∃ st ∈ mayCflow(sh) : st may call m *do*
    mayCflow ← mayCflow ∪ set of statements in m
until mayCflow does not change

computation of mustCflow(sh): dual

How is the "may call" relation computed?

- Call graph construction in Java is tricky
- AspectJ constructs (advice...) can affect the call graph of the base program

Reweaving: an analysis framework for AspectJ



cflow analysis: implementation

"may call" : use Paddle framework for callgraph construction
  still under active development

set representation: BDDs via Jedd
  (extension of Java for programming BDD-based analyses)

Implementing the analysis in Jedd:

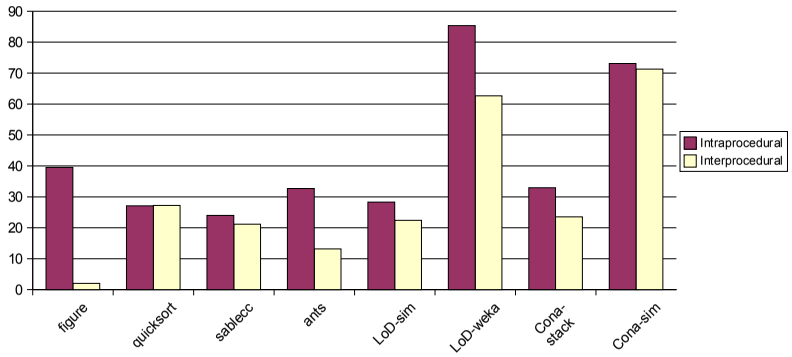
```
<stmt> mayCflow(Shadow sh) {
  <stmt> mayCflow = stmtsWithin(sh);
  <stmt> old;
  do {
    old = mayCflow;
    <method> targets = mayCflow(stmt) <> callTargets(stmt);
    mayCflow |= targets(method) <> stmtsIn(method);
  } while (mayCflow != old);
  return mayCflow;
}
```

cflow analysis: implementation (2)

Interface with abc: setting the *never valid* residues:

```
StackInfo si = stackInfo(stack);
BDDCflowStack bddcfs =
  new BDDCflowStack(cflowAnalysis, si.shadows, si.stmtMap.keySet(), si.bindsArgs );

for (Iterator stmtIt = bddcfs.neverValid(); stmtIt.hasNext();) {
  final Stmt stmt = (Stmt) stmtIt.next();

  for (Iterator rbIt = si.aa(stmt).getResidueBoxes().iterator(); rbIt.hasNext(); ) {
    final ResidueBox rb = (ResidueBox) rbIt.next();

    if (!(rb.getResidue() instanceof CflowResidue)) continue;
    CflowResidue cfr = (CflowResidue) rb.getResidue();
    if (cfr.setup() != stack) continue;

    rb.setResidue(NeverMatch.v()); }
}
```

Interprocedural Optimisations: Summing Up



All but one cflow *were static properties*
The analysis identified all of these

Send us *your* benchmarks!

Optimisations, Analyses and Beyond

- Statically determining properties:

```
declare error : uiCalls() && cflow(call(* EnterpriseBean+.*(..)))
      : "UI call from EJB";
```

  At each program point:

  - *Never matches*: no error
  - *Always matches*: error
  - *May match*: warning and runtime assert

Adapted from Laddad, *AspectJ in Action*

- *pcflow*(): analysis for precision
- Tracecuts: analysis crucial for efficiency