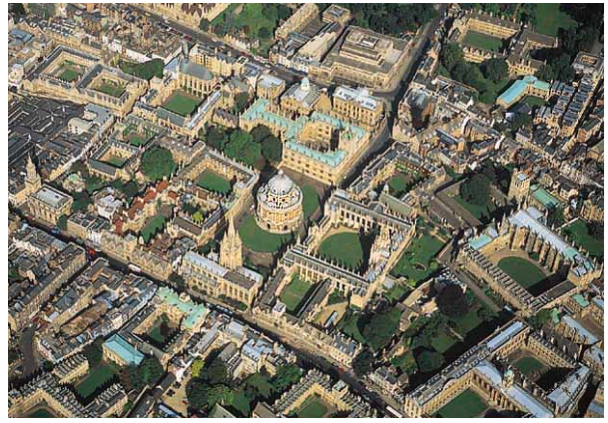
Optimizing AspectJ with abc

McGill

Laurie Hendren
Jennifer Lhoták
Ondřej Lhoták
Chris Goard
Clark Verbrugge



Oxford

Oege de Moor
Pavel Avgustinov
Sascha Kuzins
Damien Sereni
Ganesh Sittampalam
Julian Tibble

Aarhus

Aske Simon Christensen

Outline

- AspectJ introduction
- Challenges of building a compiler for AspectJ
- **abc** as an extensible and optimizing compiler
- Optimizations implemented with **abc**
- Future Work

AspectJ Programming Language

- a seamless *aspect-oriented* extension to Java
- originally developed at Xerox PARC
- tools for AspectJ now developed and supported by the Eclipse AspectJ project
  - **ajc** compiler for the AspectJ language

(**`http://eclipse.org/aspectj`**)

AspectJ Programming Language

- a seamless *aspect-oriented* extension to Java
- originally developed at Xerox PARC
- tools for AspectJ now developed and supported by the Eclipse AspectJ project
  - **ajc** compiler for the AspectJ language

  (**`http://eclipse.org/aspectj`**)

- **abc**, the **aspectbench compiler**, is a new, alternative compiler for the AspectJ language, designed for *extensibility* and *optimization* (**`http://aspectbench.org`**)

AspectJ Introduction

- introduce a small Java program, a little expression interpreter
- illustrate three main uses of AspectJ by applying it to this small example
  - aspects for additional static checking at compile time
  - adding fields/classes/constructors to classes via aspects
  - dynamic aspects for applying advice (code) at specified run-time events

Example Java Program - expression interpreter

Consider a small interpreter for an expression language, consisting of:

- SableCC-generated files for scanner, parser and tree utilities in four packages: **`parser`**, **`lexer`**, **`node`** and **`analysis`**.
- main driver class, **`tiny/Main.java`**, which reads the input, invokes parser, evaluates resulting expression tree, prints input expression and result.
- expression evaluator class, **`tiny/Evaluator.java`**

```
> java tiny.Main
Type in a tiny exp followed by Ctrl-d :
3 + 4 * 6 - 7
The result of evaluating: 3 + 4 * 6 - 7
is: 20
```

AspectJ for Static (compile-time) Checking

- Programmer specifies a pattern describing a static program property to look for and a string with the warning text.
- An AspectJ compiler must check where the pattern matches in the program, and issue a compile-time warning (string) for each match.

```
public aspect StyleChecker {
  declare warning :
    set(!final !private * *) &&
    !withincode(void set*(..) ) :
    "Recommend use of a set method.";
}
```

Using the `StyleChecker` aspect

The compilation:

```
abc StyleChecker.java */*.java
```

produces the compile-time output:

```
  parser/TokenIndex.java:34:
  Warning -- Recommend use of a set method.
    index = 4;
    ^-------^
...
```

AspectJ for Intertype Declarations

- Programmer specifies, in a separate aspect, new fields/methods/constructors to be added to existing classes/interfaces.
- An AspectJ compiler must weave in code to implement these additions.
- Other classes in the application can use the added fields/members/constructors.
- In our example, we can use an aspect to add fields and accessors to the code generated by SableCC, without touching the generated classes.

Intertype Declarations - example

- All AST nodes generated by SableCC are subclasses of **`node.Node`**.
- We must **not** directly modify the code generated by SableCC.

```
public aspect AddValue {
  int node.Node.value; // a new field

  public void node.Node.setValue(int v)
     { value = v; }

  public int node.Node.getValue()
     { return value; }
}
```

Using the AddValue aspect

```
abc AddValue.java */*.java
```

where, the evaluator visitor can be now written using the **value** field to store intermediate values.

```
public void outAMinusExp(AMinusExp n)
  { n.setValue(n.getExp().getValue() -
               n.getFactor().getValue());
  }
```

instead of the “old" way of storing intermediate values in a hash table. The aspect-oriented method is more efficient because fewer objects are created during the evaluation.

AspectJ for Dynamic Advice

- Programmer specifies a pattern describing run time events, and some extra code (advice) to execute before/after/around those events.
- An AspectJ Compiler must weave the advice into the base program for all potentially matching events.

AspectJ for Dynamic Advice

- Programmer specifies a pattern describing run time events, and some extra code (advice) to execute before/after/around those events.
- An AspectJ Compiler must weave the advice into the base program for all potentially matching events.
- Since events can depend on dynamic information:
  - some execution state may need to be tracked, and
  - some advice may be conditional on the result of a *dynamic residue test*.

The basic idea - observing a program execution

```
execution(main)
   ...
   call(genStmt)
      ...
      call(genWhileStmt)
         ...
         call(genExpr)
            ...
            call(genExpr)
            ...
            call(genExpr)
         ...
         call(genStmt)
            ...
            call(genAssignStmt)
               ...
               call(genExpr)
               ...
            ...
         ...
      ...
   ...
```

Execution Trace of base program

Observers

Initialize a counter to 0

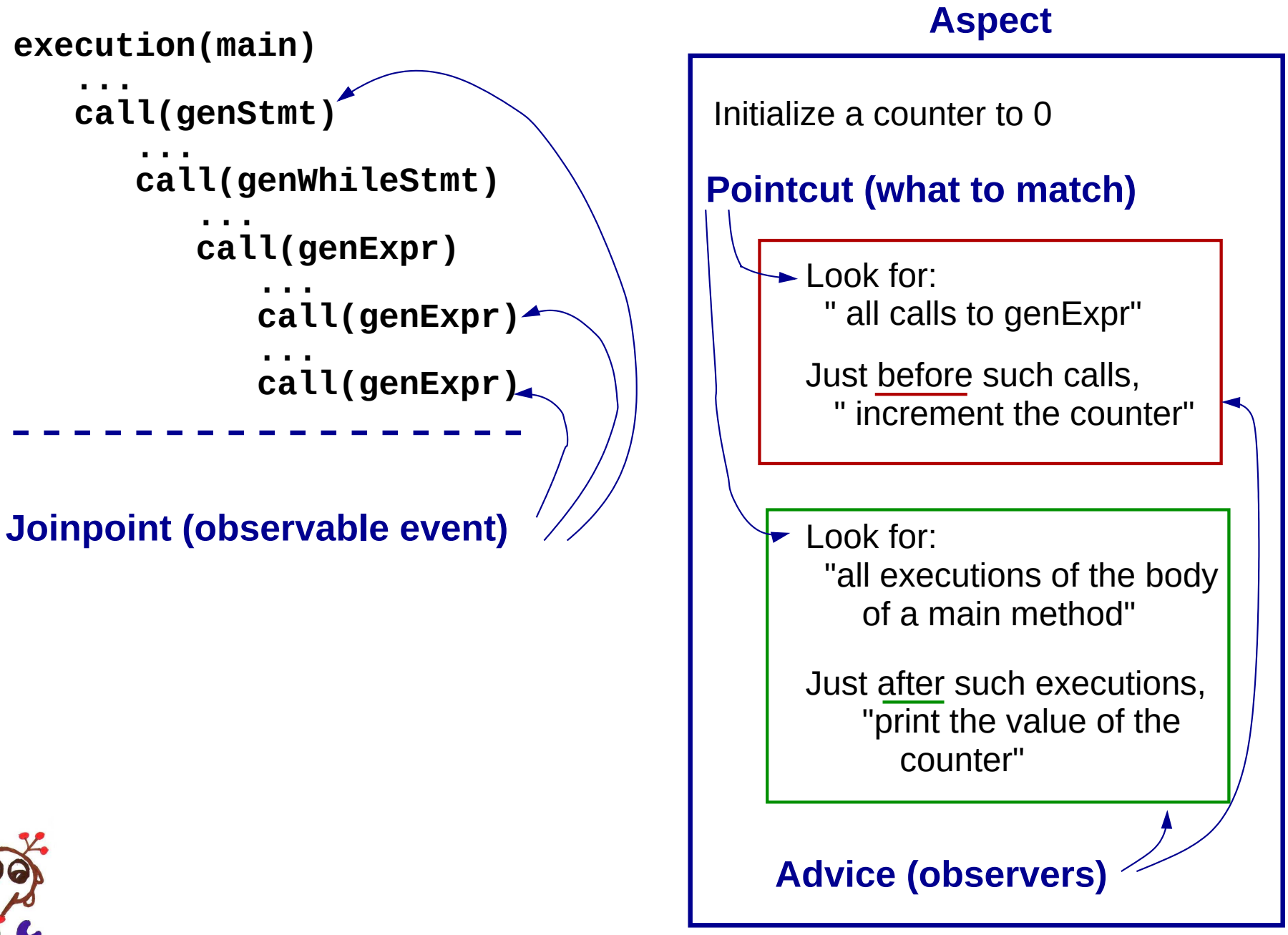
Look for:
" all calls to genExpr"

Just before such calls,
" increment the counter"

Look for:
"all executions of the body of a main method"

Just after such executions,
"print the value of the counter"

The basic idea - with AspectJ terminology



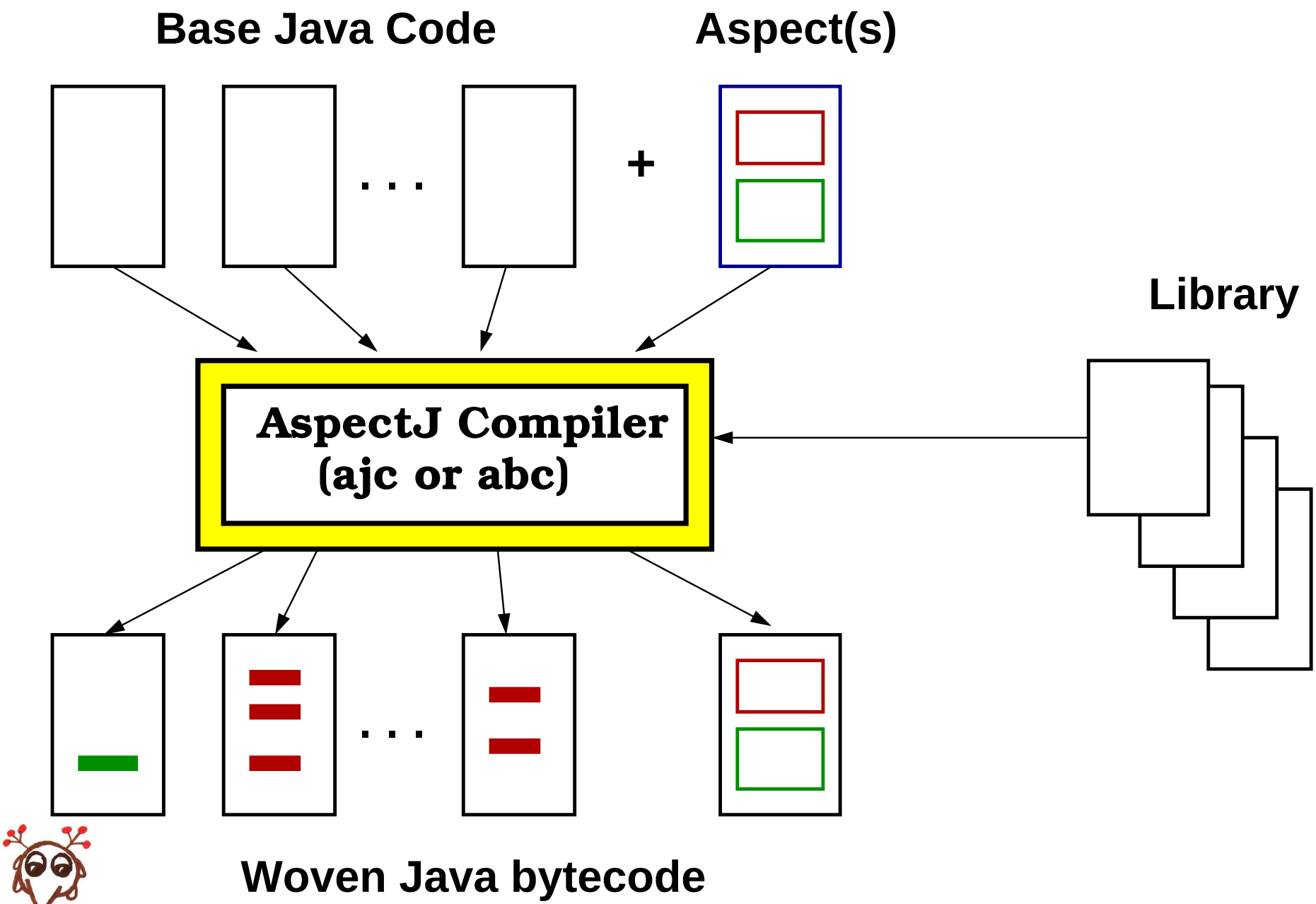
Example expressed using AspectJ

```
public aspect CountGenExpr {
    int count = 0;
    // advice to count calls to genExpr
    before () : call (String genExpr(..))
      { count++; }

    // pointcut to match execution of main
    pointcut main() :
      execution (public static void main (String[]));

    // advice to print out counter after exec of main
    after () : main()
      { System.err.println("# expr eval: " + count); }
    }
```

Compile-time matching and weaving



After matching and weaving

```
public class GenCode {
 ...
 public String genStmt(...)
   { ...
      CountGenExpr.aspectOf().
                    before$1();
      genExpr(...);
       ...
      CountGenExpr.aspectOf().
                    before$1();
      genExpr(...);
       ...
   }
}
```

woven advice

joinpoint shadows

```
public class CountGenExpr {
  // aspect fields
  int count = 0;
  // singleton aspect instance
  private static final CountGenExpr
    singleton = new CountGenExpr();
  public static CountGenExpr aspectOf()
    { if (singleton != null)
        return(singleton);
      else
        throw new NoAspectBoundEx(...);
    }
  // advice bodies
  public final void before$1()
    { count++; }
  public final void after$1()
    { System.err.println("# eval:" +
        count); }
}
```

Dynamic Advice - example 2

```
public aspect ExtraParens {
String around() :
 execution(String node.AMultFactor.toString()) ||
 execution(String node.ADivFactor.toString())
 { String normal = proceed();
   return "(" + normal + ")";
 }
}
```

Compile: **abc ExtraParens.java */*.java**
Run: **java tiny.Main**

```
The result of evaluating:
3 + (4 * 6) + (9 / 3)
is: 30
```

Recap: uses of AspectJ for example

- **Static (compile-time) check:** Check that accessor methods are always used to set non-private non-final fields.
- **Intertype declaration:** Add a new field and associated accessor methods to the SableCC-generated **`node.Node`** class.
- **Dynamic advice:**
  - Count the number of expressions evaluated.
  - Intercept calls to **`toString()`** for factors and add surrounding parentheses, if they are not already there.

Challenges: front-end

- AspectJ-specific language features, including relatively complex pointcut (patterns) language.
- Intertype declarations, need to be able to extend the type system in non-trivial ways.

Challenges: front-end

- AspectJ-specific language features, including relatively complex pointcut (patterns) language.
- Intertype declarations, need to be able to extend the type system in non-trivial ways.
- **abc**'s solution:
  - use Polyglot, an extensible framework for Java compilers (Cornell)
  - express AspectJ language via LALR(1) grammar: base Java grammar + additional grammar rules for AspectJ
  - use Polyglot's extension mechanisms to override key points in type system to handle intertype declarations.

Challenges: back-end

- Need to handle input from .java and .class files.
- AspectJ compilers need additional modules: matcher, weaver
- need to produce **efficient** woven code (.class files)

Challenges: back-end

- Need to handle input from .java and .class files.
- AspectJ compilers need additional modules: matcher, weaver
- need to produce **efficient** woven code (.class files)
- **abc**'s solution:
  - clean design of matcher and weaver using a simplified and factored pointcut language
  - use Soot, which provides Jimple IR (typed 3-addr), standard optimizations, and an optimization framework

The abc approach

abc has been designed to be an:

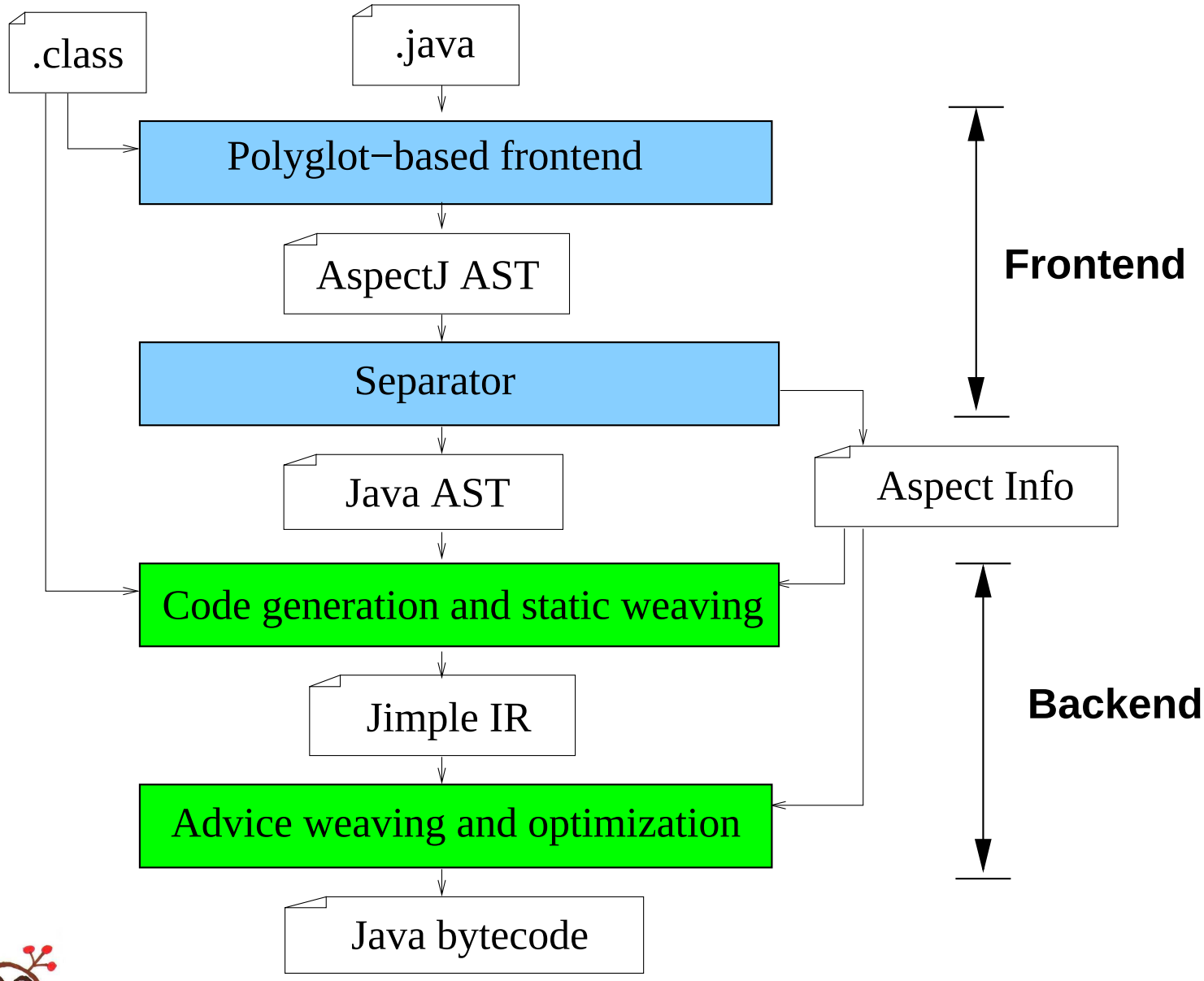
- **extensible compiler:**
  - easy to implement language extensions
  - build on two extensible frameworks, Polyglot and Soot

The abc approach

abc has been designed to be an:

- **extensible compiler:**
  - easy to implement language extensions
  - build on two extensible frameworks, Polyglot and Soot
- **optimizing compiler:**
  - convenient IR
  - good weaving strategies
  - standard compiler optimizations
  - AspectJ-specific optimizations

The abc architecture



Are there performance overheads?

From the AspectJ (ajc) FAQ:

> "Though we cannot show it without a benchmark suite, we believe that code generated by AspectJ has negligible performance overhead. Inter-type member and parent introductions should have very little overhead, and advice should only have some indirection which could be optimized away by modern VMs."

Really no significant overheads?

- Studied overheads due to weaving in:
  [10] Dufour, Goard, Hendren, de Moor, Sittampalam and Verbrugge, *Measuring the dynamic behaviour of AspectJ programs*, OOPSLA 2004.
- In general, low overheads for large or cold simple advice.
- However, **very** high overheads in some cases:
  - **`around`** advice more complicated, leads to poor space and/or time performance; and
  - **`cflow`** pointcuts need potentially expensive dynamic checks.

Example of around advice

```
public static String genExpr(Expr e, boolean verbose)
  { if (verbose)
        log("Generating Expr " + e);
    if (e instanceof Literal)
        ...
  }
public static String genStmt(Stmt s, boolean verbose)
  { ... }
```

Want to intercept all calls to **gen*** methods and change second argument to false.

```
public aspect NeutralizeFlag {
  String around (boolean flag) :
    call(String gen*(*,boolean)) && args(*,flag)
    {  if (flag) print("changing flag to false");
       String s = proceed(false);
       return(s);
    }
}
```

What's tricky about around advice?

```
public aspect NeutralizeFlag {
  String around (boolean flag) :
    call(String gen*(*,boolean)) && args(*,flag)
    {  if (flag) print("changing flag to false");
       String s = proceed(false);
       return(s);
    }
}


    ...
    str1 = genExpr(e,true);
    ...
    str2 = genStmt(s,false);
    ...
```

- **proceed** means different things, depending on the joinpoint shadow being matched.
- **ajc** has two approaches, (1) inlining/specialization and (2) closures.

ajc inlining/specialization approach

```
...
str1 = aroundBodyAdvice1(e,true); // genExpr(e,true);
...
str2 = aroundBodyAdvice2(s,false); // genStmt(s,false);
...

public static final String
                  aroundBodyAdvice1(Expr e, boolean v)
{ ...
  aroundBody1(e,false);  // proceed(false);
  ...
}

public static final String
                  aroundBody1(Expr e, boolean v)
  { return genExpr(e,v); }
```

- possible code blowup due to many copies of advice
- doesn't work for proceed in local/anonymous classes or circular advice

ajc closure approach

- can be used for all situations
- **ajc** uses this approach when **`proceed`** is in a local/anonymous class, circular advice, or with **`-XnoInline`**
- creates a closure class for each matching joinpoint shadow, thus (potentially many classes).
- each closure class contains a **`run`** method specialized to the shadow.
- one general version of the advice, takes a closure as input
- advice calls **`run`** method of closure for **`proceed`**
- requires packaging arguments into arrays of Objects which leads to a lot of allocation, boxing/unboxing and casts

abc generic switch-based approach (Kuzins M.Sc.)

- Goals:
  - general-purpose strategy
  - avoid making multiple copies of advice
  - avoid many extra classes and creation of extra objects
- Strategy:
  - give each class containing a matching shadow a static **`ClassId`**
  - within each class, give each matching shadow a static **`ShadowId`**
  - use switches to dispatch to the correct implementation of **`proceed`**

abc generic switch-based approach - Example

```
public final String around$2 (boolean flag,
  int shadowID, int classID, Object arg1, boolean arg2)
{ if (flag) ...
  switch (classID) //implementation of proceed
    { case 1: s = CodeGen.proceed$2(shadowID,arg1,arg2);
                  break;
      case 2: s = Main.proceed$2(shadowID,arg1,arg2);
                  break;
    }
    ...
}

public class CodeGen {
  ...
  public static String proceed$2(
                   int shadowID, Object arg1, boolean arg2)
    { String s;
      switch (shadowID)
         { case 1: s = genExpr((Expr) arg1, false); break;
           case 2: s = genStmt((Stmt) arg1, false); break;
           ...
         }
      return(s);
    }
}
```

Postpass optimizations

- Can start with the switch-based strategy and then specialize some, or all shadows.

```
NeutralizeVerbose.aspectOf().
   around$2(flag,1,2,e,verbose);
```

- Can specialize/inline directly, or
- specialize/inline to methods as done by **ajc**.
  - New Idea! ... recognize when specialized methods are clones and only create one new method.

Benchmarks using `around` advice

- Base Programs:
  - **sim:** a discrete event simulator for certificate revocations
  - **weka:** machine learning library
  - **ants:** ant colony simulator (ICFP 2004 contest)
- Aspects:
  - **nullptr-rec:** small advice to check for coding standards (methods returning null), from Asberry.
  - **nullptr:** same as above, but fixed to remove recursive application within advice body
  - **delayed:** captures output calls and delays to end, example of the AspectJ *worker* pattern suggested by Laddad, uses **`proceed`** in an inner class
  - **profiler:** large advice, applies to every method call

Time and space measurements

	Time (s)			Size (instr.)		
Benchmark	**ajc**	**abc** switch	**abc** inline +clone	**ajc**	**abc** switch	**abc** inline+ clone
sim-nullptr-rec	124.0	23.6	21.8	10724	8216	15089
sim-nullptr	21.4	21.9	19.9	10186	7893	8869
weka-nullptr-rec	45.5	18.9	16.3	130483	103401	148183
weka-nullptr	16.0	19.0	15.8	134290	103018	89029
ants-delayed	18.2	17.5	17.1	3785	3688	3965
ants-profiler	21.2	22.5	20.1	13401	7202	14003

.... on to cflow optimizations .....

What is a `cflow` pointcut?

```
execution(main)
   ...
   call(genStmt)
      ...
      call(genWhileStmt)
         ...
         call(genExpr)
            ...
            call(genExpr)
            ...
            call(genExpr)
         ...
         call(genStmt)
            ...
            call(genAssignStmt)
               ...
               call(genExpr)
               ...
            ...
         ...
      ...
   ...
```

```
before(): call(* genStmt(..)) &&
  cflow(call(* genWhileStmt(..)))
{ counter++; }
```

Execution Trace of base program

Original ajc implementation of `cflow`

For each cflow clause **cflow**(*pointcut_expr*)

- Create a thread-local stack for this clause.
- For each joinpoint shadow matching *pointcut_expr*, insert *update shadows*:
  - before matching joinpoint shadow, weave a push
  - after matching joinpont shadow, weave a pop
  - each push and pop must retrieve correct thread-local stack
- To determine if clause **cflow**(*pointcut_expr*) is true, weave a dynamic test (*query shadow*).
  - the clause **cflow**(*pointcut_expr*) is true when the associated cflow stack is non-empty.

Update and query shadows for our example

```
before(): call(* genStmt(..)) &&
  cflow( call(* genWhileStmt(..)) )
{ counter++; }
```

- Update Shadows

```
context = ...;
cflowstack1.getThreadStack().push(context);
genWhileStmt(...);
cflowstack1.getThreadStack().pop();
```

- Query Shadows

```
if (!cflowstack1.getThreadStack().isEmpty())
  CountGenExpr.aspectOf().before$2(...);
genStmt(...);
```

Optimized `cflow` in abc

Intraprocedural:

- share identical cflow stacks
- replace stacks with counters when no context required
- reuse thread-local stacks/counters within a method

Interprocedural:

- detect when query and/or update shadows are not needed
- needs a call graph ... but weaving changes call graph

Sharing identical `cflow` stacks

- Often several cflow clauses are identical, and can share the same cflow stack.
- Typical example:

```
pointcut scope(): cflow(call(* MyPackage.*(..)));
pointcut one(): ... && scope();
pointcut two(): ... && scope();
...
pointcut foo(): ... && cflow(call(* MyPackage.*(..)));
```

- Identify identical (up to α-renaming) cflow clauses and assign them the same cflow stack.
- Simplified version of this optimization has been adopted in **ajc**1.2.1.

Replacing stacks with counters

- Original **ajc** (version 1.2 and earlier) always used cflow stacks.
- The stacks store an array of Objects, one element of the array for each context value to be stored.
- A common case is that there are no context values to store.
  - In this case, use a thread-local counter, instead of a thread-local stack.
- This optimization has been adopted in **ajc**1.2.1.

Reusing thread-local stack/counter within a method

- According to the AspectJ semantics, each stack/counter must be local to a thread.
- Retrieving the thread-local stack/counter has a significant overhead.
- Only load a thread-local stack/counter once and store reference in local variable.
- Implemented by generating:

```
if (localStack1 == null)
    localStack1 = cflowstack1.getThreadStack();
```

... and then optimizing away the unnecessary code using a customized intra-procedural null pointer analysis.

Whole program optimizations to eliminate shadows

```
before(): call(* genStmt(..)) &&
  cflow( call(* genWhileStmt(..)) )
{ counter++; }
```

- Query Shadows

```
if (!cflowstack1.getThreadStack().isEmpty())
  CountGenExpr.aspectOf().before$2(...);
genStmt(...);
```

- Update Shadows

```
context = ...;
cflowstack1.getThreadStack().push(context);
genWhileStmt(...);
cflowstack1.getThreadStack().pop();
```

Examples: when can you eliminate shadows?

- Never matches

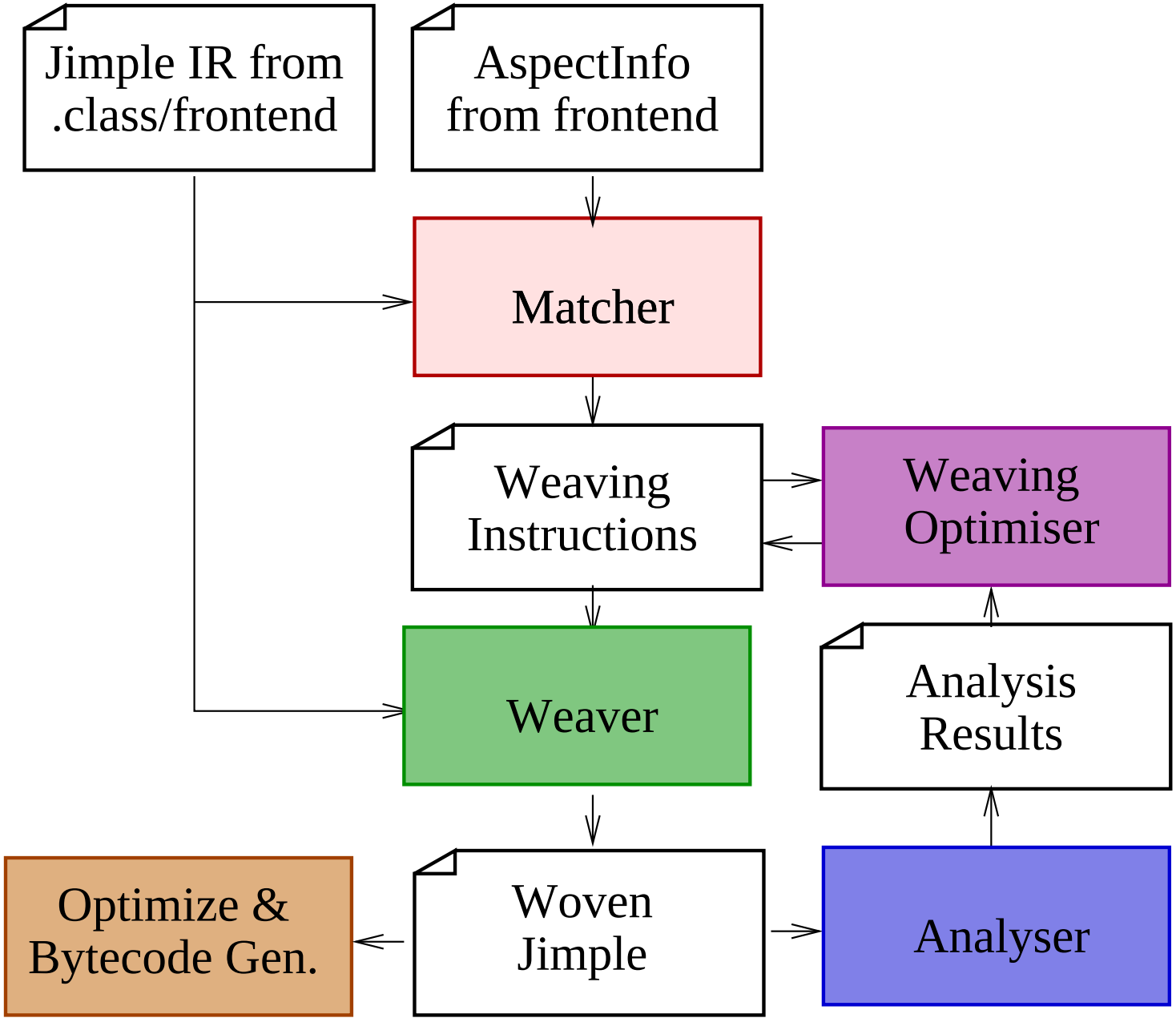
```
public static void main(...)
 { ...
   output = genStmt(body,true);
   ...
 }
```

- Always matches or possibly matches

```
public static String genWhileStmt(Stmt s, boolean v)
 { outcond = genExpr(s.getExpr(),v);
   outbody = genBody(s.getBody(),v);
   ...
 }

public static String genBody(Stmt s, boolean v)
   { out = genStmt(s,v);
     ...
   }
```

Need to know the call graph, but weaving changes it



Interprocedural analyses for `cflow`

mayCflow: Set of statements whose execution **may** be within given cflow

mustCflow: Set of statements which execute **only within** given cflow

necessaryShadows: Set of stack updates observable by some stack query

- implemented using Jedd [PLDI2004]
- even though the sets are large, the BDDs represent them efficiently
- using a points-to based call graph instead of a CHA-based call graph leads to slightly more accurate and faster cflow analyses

Benchmarks using `cflow`

- **figure:** small benchmark, part of AspectJ tutorial
- **quicksort:** small example used in previous work on cflow analysis [Sereni and de Moor, AOSD2003]
- **sablecc:** count allocations in a phase of a compiler generator
- **ants:** identify allocations in a key simulator loop
- **LOD:** Law of Demeter checker [Lieberherr, Lorenz and Wu, AOSD2003]
- **Cona:** Aspects for Contracts [Skotiniotis and Lorenz, OOPSLA2004(companion)]

Static measurements of `cflow` optimizations

- Intraprocedural optimizations
  - only the **ants** benchmark needs a stack, rest can use counters
  - numbers of stacks/counters greatly reduced (a lot of sharing)
- Interprocedural optimizations
  - all query and update shadows removed, except for 1 in the **sablecc** benchmark

Speedups due to `cflow` optimizations

Benchmark	no-opt (sec)	share (sec)	share+ cntrs (sec)	share+ cntrs+ reuse (sec)	share+ cntrs+ reuse × no-opt	+inter-proc (sec)	+inter-proc × intra
figure	1072.2	238.3	90.3	20.3	(52.82)	1.96	(10.38)
quicksort	122.3	75.1	27.9	27.4	(4.46)	27.3	(1.00)
sablecc	29.0	29.1	22.8	22.5	(1.29)	20.4	(1.10)
ants	18.7	18.8	18.7	17.9	(1.04)	13.1	(1.37)
LoD-sim	1723.9	46.6	32.8	26.2	(65.80)	23.7	(1.11)
LoD-weka	1348.7	142.5	91.9	75.2	(17.93)	66.3	(1.13)
Cona-stack	592.8	80.1	41.2	27.4	(21.64)	23.1	(1.19)
Cona-sim	75.8	75.3	73.8	72.0	(1.05)	73.6	(0.98)

Conclusions - Recap

- **abc** is a new compiler for AspectJ which is extensible and optimizing.
- **abc** uses Polyglot to build an extensible front-end and Soot to build an extensible back-end.
- It is worth thinking about AspectJ-specific optimizations, and **abc** has implemented optimizations (PLDI05) to handle major overheads identified in the OOPSLA04 study.
- Need a special strategy for interprocedural analysis, since weaving can make non-trivial changes to the call graph.

What's next?

- Both the **abc** team and others are implementing new extensions of AspectJ.
  - *tracematches* to match on related traces of events (OOPSLA05)
  - adding and extending *open modules* for AspectJ (AOSD06)
- Apply interprocedural analysis to statically check more dynamic pointcuts and optimize tracematches.
- Enable more powerful static checking of **`declare`** constructs.
- Lots more work by the **abc** team to come ... we welcome users and benchmarks!

`http://aspectbench.org`