Adding Trace Matching with Free Variables to AspectJ

Chris Allan, Pavel Avgustinov, Sascha Kuzins, Oege de Moor, Damien Sereni, Ganesh Sittampalam and Julian Tibble (Oxford)

Laurie Hendren and Ondřej Lhoták (McGill)

Aske Simon Christensen (Aarhus)

Introduction

AspectJ: An Aspect-Oriented Extension of Java

- Define patterns on run-time events
- Match patterns to events at run-time
- Execute extra code when matching events occur

Introduction

AspectJ: An Aspect-Oriented Extension of Java

- Define patterns on run-time events
- Match patterns to events at run-time
- Execute extra code when matching events occur

Tracematches

Match whole *execution histories*, not just events

AspectJ Lingo

Join Point = event (call, execution, field set / get)

Pointcut = pattern on join points

Advice = extra code to run

AspectJ: Advice

```
aspect Autosave {
   int count = 0;
   after(): call(* Command.execute(..))
        { count++; }

   after(): call(* Application.save()) || call(* Application.autosave())
        { count = 0; }

   before(): call (* Command.execute(..))
        { if (count > 4)
             Application.autosave(); }
}
```

AspectJ: Join Points

enter call Command.execute()

  enter execution Command.execute()

    (…)

  exit execution Command.execute()

exit call Command.execute()

enter call Application.save()

  enter execution Application.save()

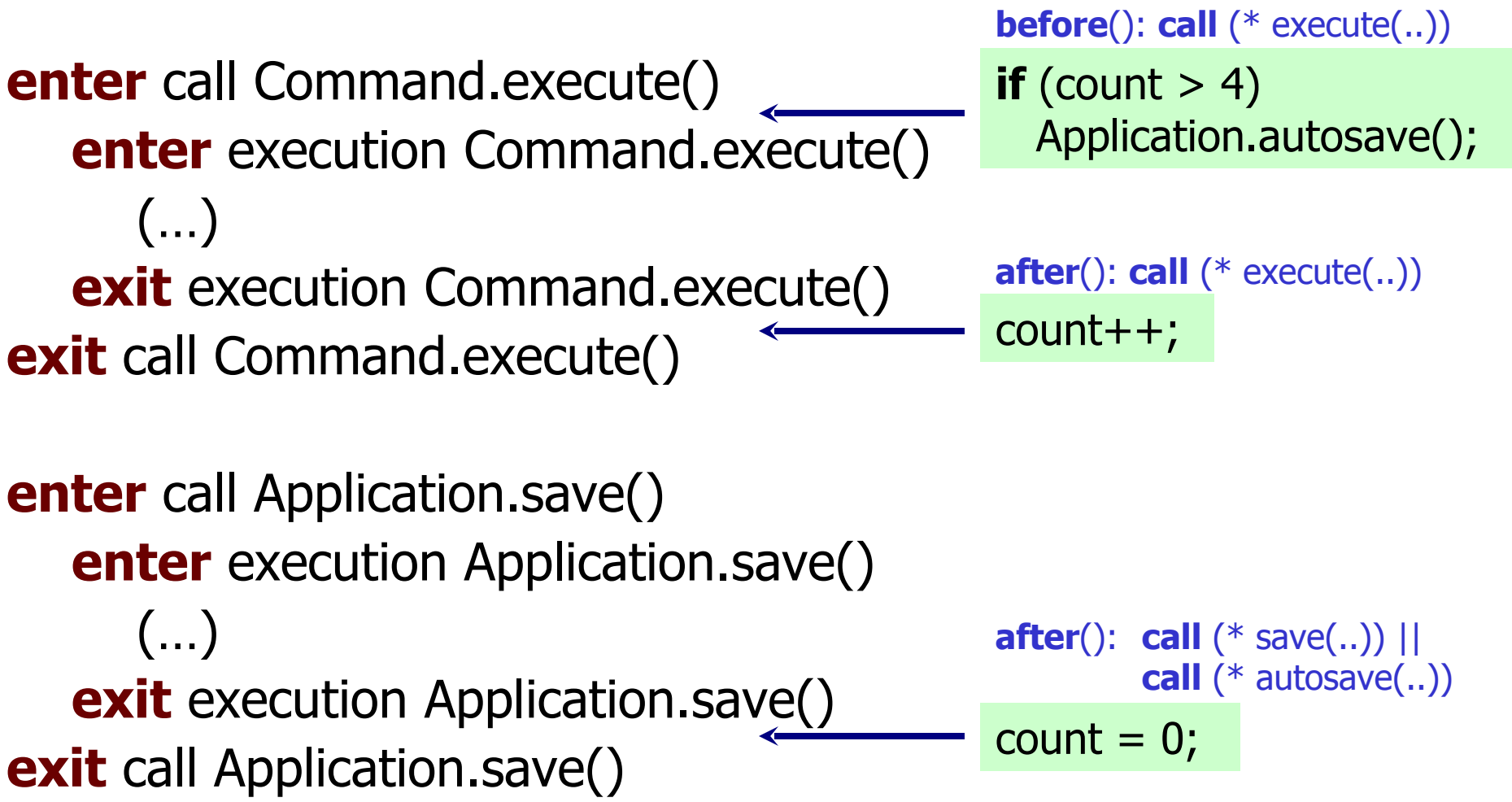
    (…)

  exit execution Application.save()

exit call Application.save()

```
Command c;
(…)
c.execute();
Application.save();
```

AspectJ: Matching



Trace Matching

Tracematches match on the entire execution history of the program

Related Work:

Walker and Viggers
Douence *et al*
Bockisch *et al*
Bodden and Stolz
Martin *et al*
Goldsmith *et al*

Contributions:

- Trace matching with free variables
- Semantics of tracematches
- Implemented in the *abc* compiler
- Eliminating memory leaks

Traces

```
enter call Command.execute()
    enter execution Command.execute()
        (…)
    exit execution Command.execute()
exit call Command.execute()

enter call Application.save()
    enter execution Application.save()
        (…)
    exit execution Application.save()
exit call Application.save()
```

Trace =

sequence of join point **enter** / **exit** events

Example: Autosave

```
tracematch() {

    sym save after:
            call ( * Application.save() )
        || call ( * Application.autosave() );

    sym action after:
        call ( * Command.execute() );


    action [5]
        { Application.autosave(); }

}
```

Example: Autosave

```
tracematch() {

    sym save after:
            call ( * Application.save() )
        || call ( * Application.autosave() );

    sym action after:
        call ( * Command.execute() );

    action [5]
        { Application.autosave(); }

}
```

Symbols = pointcuts

Pattern = regexp over symbols

Matching

```
sym action after:
   call ( * Command.execute() );
```

exit call Command.execute()

```
sym save after:
   call (* Application.save())
|| call (* Application.autosave())
```

exit call Application.save()
exit call Application.autosave()

action [5]

The pattern matches traces:

ending with 5 events matching action

with *no* events matching save in between

Matching with Free Variables

```
tracematch(Subject s, Observer o) {

    sym create_observer
        after returning(o):
            call ( Observer.new(..) )
        && args (s);

    sym update_subject after:
            call ( * Subject.update(..) )
        && target (s);

    create_observer update_subject *
        { o.update_view(); }

}
```

Matching with Free Variables

```
tracematch(Subject s, Observer o) {

    sym create_observer
        after returning(o):
            call ( Observer.new(..) )
        && args (s);

    sym update_subject after:
            call ( * Subject.update(..) )
        && target (s);

    create_observer update_subject *
        { o.update_view(); }

}
```

```
o = new Observer(s);
```

```
s.update(..);
```

Matching with Free Variables

create_observer binds the Observer o and Subject s
update_subject binds the Subject s

```
create_observer  update_subject *
```

Matches a trace if there is a *consistent* binding of o and s
each symbol must bind s to the same value

There can be several such bindings:

run the body once for each set of bindings

After an update to *s*, the body is run for each *o* observing *s*

Example: DB Connection Pooling

```
public aspect DBConnectionPooling {

pointcut connectionCreation(String url, String uid, String password) : …;
pointcut connectionRelease(Connection connection) : …;

    Connection tracematch
        (Connection connection, String url, String uid, String password) {

        sym get_connection1 after returning(connection):
            connectionCreation(url, uid, password);

        sym get_connection2 around(url, uid, password):
            connectionCreation(url, uid, password);

        sym release_connection before:
            connectionRelease(connection);

        get_connection1 release_connection get_connection2
            { return connection; }

    }
…
```

Example: DB Connection Pooling

```
public aspect DBConnectionPooling {
    ...
    Connection tracematch
        (Connection connection, String url, String uid, String password) {
        sym get_connection1 after returning(connection):
            connectionCreation(url, uid, password);
        sym get_connection2 around(url, uid, password):
            connectionCreation(url, uid, password);
        sym release_connection before:
            connectionRelease(connection);
        get_connection1 release_connection get_connection2
            { return connection; }
    }
    void around() : connectionRelease(*) { /* Do Nothing */ }
}
```

Semantics and Implementation

Matching Semantics
No Free Variables

```
tracematch () {
  sym F before: call(* f());
  sym G before: call(* g());
  F G +   { }
}
```

Filter out all events that do not match any symbol in the tracematch

The last event must match a symbol

```
enter call f();
enter call f();
enter call g();
enter call g()
```

The tracematch applies if:

some *suffix* of the *filtered* trace is matched by a word in the pattern

~~enter call f();~~
enter call f();
enter call g();
enter call g()

matched by

F;
G;
G

Matching Semantics
Free Variables

```
tracematch (Object x) {
  sym F before: call(* f()) && target(x);
  sym G before: call(* g()) && target(x);
  F G +   { }
}
```

Apply *all* possible substitutions, then match as before

Trace	Filtered, x=o	Filtered, x=q
o.f()	o.f()	~~o.f()~~
q.f()	~~q.f()~~	q.f()
o.g()	o.g()	~~o.g()~~
q.f()	~~q.f()~~	q.f()
o.g()	o.g()	~~o.g()~~

Operational Semantics

- Matching: Run an automaton for the pattern alongside the program
  - The automaton accumulates *bindings*
  - When a final state is reached, execute the body of the tracematch for each binding

Operational Semantics = Declarative Semantics

- Implemented in the *abc* compiler for AspectJ

Implementation Issues

Memory Usage

A naive implementation would suffer memory leaks:

Objects bound in matching cannot be reclaimed by GC

Use *weak references* to store bindings whenever possible

Some tracematches can still cause memory leaks

Compiler warning

Static analysis of the pattern to detect this

Performance : DB Connection Pooling

DBConnectionPooling :

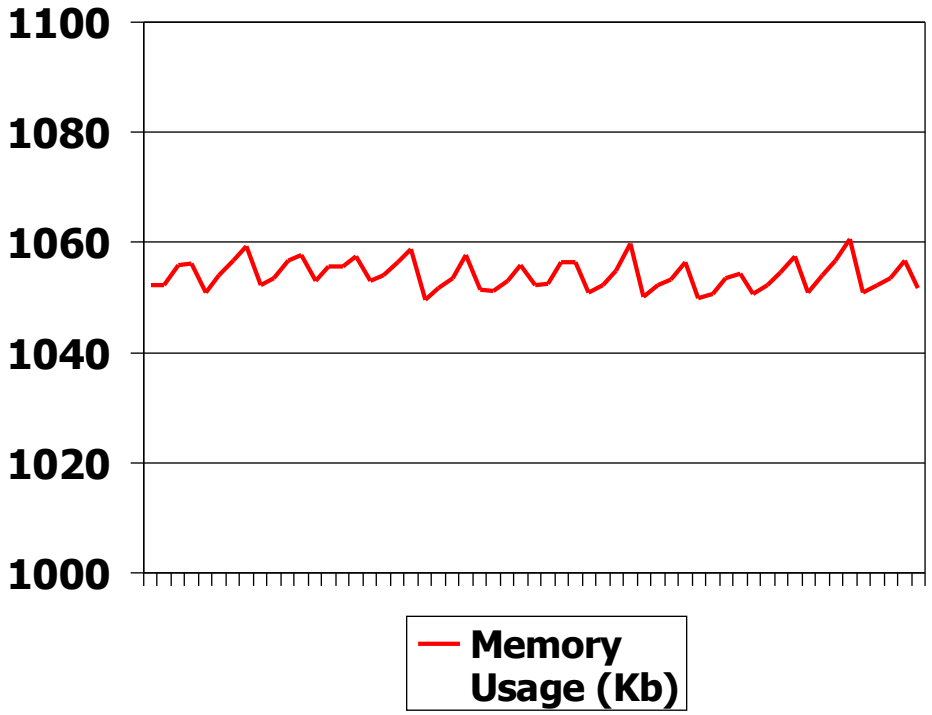
Pure Java (no pooling)	6.0s
with hand-coded pooling aspect	1.0s
with pooling tracematch	1.2s

DB Connection Pooling AspectJ (Laddad, *AspectJ in Action*)	DB Connection Pooling Tracematch
77 LOC (47 implementing pool)	21 LOC

Performance: Memory Usage

Memory usage:

JHotDraw with SafeIterators tracematch



No space leaks

Eliminating space leaks is *essential* to achieve good performance

Related Work

	PURPOSE			PATTERNS			IMPLEMENTATION		
	fault finding	functionality	AspectJ	variables	filtering	context-free	semantics	leak busting	static match
Walker&Viggers	+/-	+	+	-	-	+	-	-	-
Douence et al	+/-	+	-	+	-	-	-	-	-
Bockisch et al	+/-	+	-	+	-	+	-	-	-
Bodden & Stolz	+	-	+	+	-	-	+	+	-
Martin et al	+	-	-	+	-	+	-	-	+
Goldsmith et al	+	-	-	+	-	+	-	-	+
This paper	+/-	+	+	+	+	-	+	+	+/-

Conclusion

- Tracematches
  - Match patterns on execution history
  - Free variables to bind state
- Future work: optimising tracematches
- Tracematches are implemented as an extension of the *abc* compiler

get abc 1.1! **http://aspectbench.org**

PTQL and Tracematches

```
tracematch (Object x) {
  sym A after: call(* a()) && target(x);
  sym B after: call(* b()) && target(x);
  sym C after: call(* c()) && target(x);
  A B C   { }
}
```

```
SELECT *
FROM   MethodInvoc('*.a') A
       JOIN MethodInvoc('*.b') B
         ON A.receiver = B.receiver
       JOIN MethodInvoc('*.c') C
         ON B.receiver = C.receiver
```

o.a(); q.a(); o.b(); q.b(); q.c(); o.c()

PTQL query has 2 solutions
tracematch applies twice

o.a(); o.c(); o.b(); o.c()

Tracematch does not apply, PTQL query does