A Language Workbench for Creating Production-Ready Extensions to AspectJ

Arik Hadas
Dept. of Mathematics and Computer Science
The Open University of Israel

Advisor:
David H. Lorenz
“Explicit join points looks interesting, let's evaluate it”

abc? AWESOME?
Spoofax? xtext?
Our Research Goal

- Tool for the development, evaluation and production of extensions for AspectJ
  - Like abc
- Workbench, not a compiler
  - Provide common editing tools
  - Compatible with AOP development tools
- Generate production-ready extensions
  - Work with a commonly used version of AspectJ
  - Proper support for programming in multiple extensions simultaneously
Limitations of the AspectBench Compiler (abc)

- Used to be the default choice for implementing AspectJ extensions
- Not suitable for development of new extensions
  - Does not work with recent versions of AspectJ
- Not suitable for evaluation of new extensions
  - Does not provide development tools
  - No support for advanced weaving semantics
Language Workbench (LW) for Java

- DSL Code
- Java
- Bytecode

Language Workbench

javac
Language Workbench (LW) for Java

- DSL Code
- Language Workbench
- Java
-.javac
- Bytecode
Will It Work for AspectJ?

DSL Code
Extension Code
Java
Bytecode

Language Workbench

javac

No Aspects
Will It Work for AspectJ?

DSL Code
Extension Code

Java

Language Workbench

javac

Bytecode

No Aspects
Replacing javac with ajc

Language Workbench

Extension Code

Java AspectJ

javac ajc

Bytecode Woven Bytecode

No Multiple DSALs
AOP Composition Framework (CF)

- To work with multiple AspectJ extensions simultaneously, one will need to define:
  - Weaving semantics for co-advising
  - Weaving semantics for foreign advising
- CF Allows to define the required semantics
  - As opposed to ajc
- CF does not provide editing tools
<table>
<thead>
<tr>
<th></th>
<th>Language Workbench</th>
<th>AOP Composition Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tools for creation &amp; usage of languages</td>
<td>🟢</td>
<td>🟥</td>
</tr>
<tr>
<td>Defining weaving semantics needed for DSALs</td>
<td>🟥</td>
<td>🟢</td>
</tr>
<tr>
<td></td>
<td>Language Workbench</td>
<td>AOP Composition Framework</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Tools for creation &amp; usage of languages</td>
<td>😊</td>
<td>😞</td>
</tr>
<tr>
<td>Defining weaving semantics needed for DSALs</td>
<td>😞</td>
<td>😊</td>
</tr>
</tbody>
</table>
Naive Combination of LW and CF

- DSAL Code
- AspectJ
- Woven Bytecode
But We Still Lack AOP Tools..
Traditional LW Architecture

DSAL Code

Language Workbench

Code Transformation

Composition Framework

AspectJ

Woven Bytecode
Our Workbench Architecture

DSAL Code

Language Workbench

AspectJ DSAL Code

Code Transformation

Composition Framework

Woven Bytecode

Compliance with AOP tools
Our Workbench Architecture

- Standalone DSAL compiler
- Can generate debugging & browsing information

DSAL Code

Code Transformation

Composition Framework

Woven Bytecode

Compliance with AOP tools
Validation

- We implemented a workbench
- We Implemented third-party extensions that were proposed to AspectJ
  - COOL
  - Closure Join Points (CJP)
  - Explicit Join Points (EJP)
- Available as an open source
  - https://github.com/OpenUniversity
Our Workbench Implementation

DSAL Code

Spoofax

DSAL Code

AWESOME*

Woven Bytecode
Enhancing AWESOME

AspectJ Code

ajc

aspectjtools

aspectjweaver

Extension Code

AWESOME*

Pluggable tools

Code Transformation

Pluggable weaver
AWESOME's Weaving Model

List<BcelShadow> **around**(MultiMechanism mm, LazyClassGen clazz):
  **reify** Class(mm,clazz) { ... }

**public** List<IEffect> **match**(BcelShadow shadow) { ... }

**public** List<IEffect> **order**(BcelShadow shadow, List<IEffect> effects) { ... }

**void around**(MultiMechanism mm, List effects, BcelShadow shadow):
  **execution**(void MultiMechanism.**mix**(List, BcelShadow)) { ... }
Extended Weaving Model

public void preweave(List<ResolvedType> types) { ... }

List<BcelShadow> around(MultiMechanism mm, LazyClassGen clazz):
  reify Class(mm, clazz) { ... }

public List<IEffect> match(BcelShadow shadow) { ... }

public List<IEffect> order(BcelShadow shadow, List<IEffect> effects) { ... }

void around(MultiMechanism mm, List effects, BcelShadow shadow):
  execution (void MultiMechanism.mix(List, BcelShadow)) { ... }

public void preweave(List<ResolvedType> types) { ... }
So I've been asked to implement EJP...
CJP – Grammar Definition

| Expr ::= ... | ClosureJoinpoint. |
| StmtExpr ::= ... | ClosureJoinpoint. |
| ClosureJoinpoint ::= |
| “exhibit” ID “(” [ParamList] “)” Block |
| “(” [ArgList] “)” |
| “exhibit” ID Block. |
| AspectMember ::= ... | JoinpointDecl. |
| JoinpointDecl ::= |
| “joinpoint” Type ID “(” [ParamList] “)” [ThrowsList]. |
| AdviceDecl ::= ... | CJPAdviceDecl. |
| CJPAdviceDecl ::= |
| CJPAdviceSpec ::= |
| Type “before” ID “(” [ParamList] “)” |
| Type “after” ID “(” [ParamList] “)” |
| Type “after” ID “(” [ParamList] “)” |
| “throwing” “(” [Param] “)” |
| Type “around” ID “(” [ParamList] “)” |

Figure 9: Syntax for Closure Joinpoints, as a syntactic extension to AspectJ (shown in gray)

**sorts** JoinpointDeclaration

**context-free syntax**

"exhibit" MethodName "(" {FormalParam ","}* ")" Block
"(" {Expr ","}* ")" ->
Expr{cons("ClosureJoinpoints")}

"exhibit" MethodName Block ->
Expr {cons("ShortClosureJoinpoints")}

JoinpointDeclaration -> AspectBodyDec
"joinpoint" ResultType Id "(" {FormalParam ","}* ")"
Throws? ";" ->
JoinpointDeclaration{cons("JoinpointDeclaration")}
(Anno | MethodMod)* CJPAdviceSpec Throws? Block ->
AdviceDec {cons("CJPAdvice")}

"before" Id "(" {FormalParam ","}* ")" ->
CJPAdviceSpec {cons("CJPBefore")}

"after" Id "(" {FormalParam ","}* ")" ->
CJPAdviceSpec {cons("CJPAfter")}

"after" Id "(" {FormalParam ","}* ")" "returning"
CJPSingleParam?
->CJPAdviceSpec {cons("CJPAfterReturning")}

"after" Id "(" {FormalParam ","}* ")" "throwing"
CJPSingleParam?
->CJPAdviceSpec {cons("CJPAfterThrowing")}

"(" FormalParam? ")" -> CJPSingleParam
{cons("CJPSingleParam")}
ResultType "around" Id "(" {FormalParam ","}* ")"
-> CJPAdviceSpec {cons("CJPAround")}

**lexical syntax**

"exhibit" -> Keyword
"joinpoint" -> PseudoKeyword
CJP – Grammar Definition

Expr ::= ... | ClosureJoinpoint.
StmtExpr ::= ... | ClosureJoinpoint.
ClosureJoinpoint ::= "exhibit" ID "(" [ParamList] ")" Block
                    "(" [ArgList] ")" | "exhibit" ID Block.

"exhibit" MethodName "(" {FormalParam ","}* ")" Block "(" {Expr ","}* ")"
  -> Expr {cons("ClosureJoinpoints")}

"exhibit" MethodName Block -> Expr {cons("ShortClosureJoinpoints")}
But It Will Not Compile..

package research;

class HelloWorld {
    public static void main(String[] args) {
        exhibit say(String message) {
            System.out.println("Hello, " + message);
        }("World");
    }
}

package research;

aspect Impact {
    joinpoint void say(String message);
    after say(String message) {
        System.out.println("It did a " + message + " of good.");
    }
}
closure-to-java-impl =
  ?ShortClosureJoinpoints(<or(?MethodName(Id(jp_name)), ?MethodName(_, Id(jp_name))), block);
  !Invoke(
    Method(
      NewInstance(
        None(),
        ClassOrInterfaceType(TypeName(Id("JoinpointWrapper")), None())
      ),
      Some(
        ClassBody(
          [ MethodDec(
              MethodDecHead(
                [MarkerAnno(TypeName(Id("Closure"))), Public()],
              None(),
              Void(),
              Id(jp_name),
              [],
              None()
            ),
            block
          ]
        )
      ),
      None(),
      Id(jp_name)
    )
  )
)}
CJP – Replacing ajc with AWESOME*

```java
package research;

public class HelloWorld {
    public static void main(String[] args) {
        exhibit say(String message) {
            System.out.println("Hello, " + message);
        }("World");
    }
}

package research;

aspect Impact {
    joinpoint void say(String message) {
        System.out.println("It did a " + message + " of good.");
    }
}
```

Hello, World
It did a World of good.
CJP – Behind the Scenes

Original Code

```java
package research;

public class HelloWorld {
    public static void main(String[] args) {
        exhibit say(String message) {
            System.out.println("Hello, " + message);
        }("World");
    }
}
```

Transformed Code

```java
package research;

import closures.runtime.*;
import org.aspectj.lang.annotation.*;
import org.aspectj.lang.*;

public class HelloWorld {
    public static void main(String[] args)
    {
        new JoinpointWrapper()
        {
            @Closure public void say(String message)
            {
                System.out.println("Hello, " + message);
            }
            .say("World");
        }
    }
}
```
CJP Implementation

- Passed all tests from original prototype
  - Few invalid tests were fixed
- CJP programs runnable in Eclipse
  - Looks like regular AspectJ project
- Non trivial extension
  - Used context-aware code transformations
Context-aware Code Transformation

```java
package research;

public class HelloWorld {
    public static void main(String[] args) {
        exhibit say(String message) {
            System.out.println("Hello, " + message);
            return 8;
        }("World");
    }
}
```

```java
package research;

aspect Impact {
    joinpoint int say(String message); 
    after say(String message) {
        System.out.println("It did a " + message + " of good.");
    }
}
```
Context-aware Code Transformation

Need to know about the joinpoint declaration when transforming the base code!
Another example: COOL

```java
package base;

public class BoundedStack implements Stack {
    protected Object[] buffer;
    private int usedSlots = 0;

    public BoundedStack(int capacity) {
        this.buffer = new Object[capacity];
    }

    public Object pop() {
        Object result = buffer[usedSlots - 1];
        usedSlots--;
        buffer[usedSlots] = null;
        return result;
    }

    public void push(Object obj) {
        Multiple markers at this line
        - implements base.Stack.push
        - advised by injar aspect: BoundedStackCoord.cool
    }
}
```

```java
package base;

public class BoundedStack {

    public coordinator base.BoundedStack {

        selfexec {push(java.lang.Object), pop()};
        mutex {push(java.lang.Object), pop()};

        condition full = false, empty = true;
        int top = 0;

        push(java.lang.Object):
            requires (!full);
            on_entry {top = top + 1;}
            on_exit {
                empty = false;
                if (top == buffer.length) full = true;
            }

        pop():
            requires (!empty);
            on_entry {top = top - 1;}
            on_exit {
                full = false;
                if (top == 0) empty = true;
            }
    }
```
AJDT Markers for COOL

```java
package base;

public class BoundedStack implements Stack {
    protected Object[] buffer;

    private int usedSlots = 0;

    public BoundedStack(int capacity) {
        this.buffer = new Object[capacity];
    }

    public Object pop() {
        Object result = buffer[usedSlots - 1];
        usedSlots--;
        buffer[usedSlots] = null;
        return result;
    }

    public void push(Object obj) {
        // Multiple markers at this line
        - implements base.Stack.push
        - advised by injar aspect: BoundedStackCoord.cool
    }
}
```
Another example: EJP

- Implemented features that were omitted in original prototype
  - Pointcut arguments
  - Policy enforcement
- Used the 'preweave' extension in the AWESOME's weaving model
Using the preweave phase

```java
package ex_pointcutargs;

public class Main {
    public static void main(String[] args) {
        new Main().foo();
    }

    public void foo() {
        System.out.println("at foo");
        pointcutargs mm():call(* goo(..));
        goo();
    }

    public void goo() {
        System.out.println("at goo");
    }
}
```

```java
package ex_pointcutargs;

aspect Aspect {
    public joinpoint void jp() pointcutargs mm();
    before(): jp.mm() {
        System.out.println("calling " + "something that was added to aa.mm");
    }
}
```

Empty pointcut

Extending pointcut in base code
Related Work

• Language Workbenches
  - [Fowler, 2005] Language workbenches: The killer-app for domain specific languages.

• The AspectBench Compiler

• AOP Composition Frameworks
## Tools Comparison

<table>
<thead>
<tr>
<th></th>
<th>abc</th>
<th>AWESOME</th>
<th>Spoofax</th>
<th>Workbench</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tools for custom syntax definition</td>
<td>V</td>
<td>X</td>
<td>X</td>
<td>V</td>
</tr>
<tr>
<td>Extensible Java/AspectJ syntax</td>
<td>V</td>
<td>X</td>
<td>X</td>
<td>V</td>
</tr>
<tr>
<td>Tools for code transformation</td>
<td>V</td>
<td>X</td>
<td>X</td>
<td>V</td>
</tr>
<tr>
<td>Editing tools for end-programmers</td>
<td>V</td>
<td>X</td>
<td>X</td>
<td>V</td>
</tr>
<tr>
<td>Ability to define the weaving semantics required for DSAL</td>
<td>V</td>
<td>X</td>
<td>X</td>
<td>V</td>
</tr>
<tr>
<td>Works with a recent version of AspectJ</td>
<td>V</td>
<td>X</td>
<td>X</td>
<td>V</td>
</tr>
<tr>
<td>Compliance with AJDT</td>
<td>V</td>
<td>X</td>
<td>X</td>
<td>V</td>
</tr>
</tbody>
</table>
Conclusion

- A novel design for a workbench that produces first-class AspectJ extensions
  - A modern alternative to abc
  - AOP composition framework used as a back-end to achieve first-class DSL
  - DSAL code passed to the back-end to achieve first-class AOP language

- Validation
  - Prototype comprising Spoofax and AWESOME*
  - Plug-ins for COOL, EJP and CJP

- Future Work
  - Evaluate AspectJ extensions in real-world cases
Thank You!

Arik Hadas  
Dept. of Mathematics and Computer Science  
The Open University of Israel  
arik.hadas@openu.ac.il
https://github.com/OpenUniversity