Easy Language Extension with Meta-AspectJ

Yannis Smaragdakis
Georgia Tech

Research work joint with Shan Shan Huang, David Zook

(Apologies to Chris Lengauer)

Interests: C++ templates, DSLs, meta-programming tools, automatic testing, binary transformations, component software, OO language implementation, code generation for distributed computing

Introduction

How to use Meta-AspectJ to implement simple (annotation-based) extensions to Java

- Meta-AspectJ (MAJ): a language for generating AspectJ (and, of course, Java) programs
- Simple, unobtrusive DSLs are important:
  - Full language extensibility can be confusing and complex.
  - Java 1.5 annotations can express simple DSLs
- Applications: implementing annotation-based DSLs with MAJ is simple
  - Interface filler and connection pooling example DSLs

Introduction

How to use Meta-AspectJ to implement simple (annotation-based) extensions to Java

- Meta-AspectJ (MAJ): a language for generating AspectJ (and, of course, Java) programs
- Simple, unobtrusive DSLs are important:
  - Full language extensibility can be confusing and complex.
  - Java 1.5 annotations can express simple DSLs
- Applications: implementing annotation-based DSLs with MAJ is simple
  - Interface filler and connection pooling example DSLs

Basic Operators

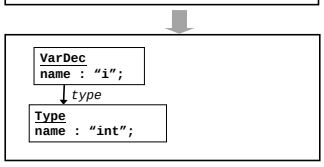
- Operators:
  - quote:

```
VarDec v = `[ int i; ];
```


Basic Operators

- Operators:
  - quote:

```
VarDec v = `[ int i; ];
```



Basic Operators

- Operators:
  - quote:

```
VarDec v = `[ int i; ];
```

  - unquote:

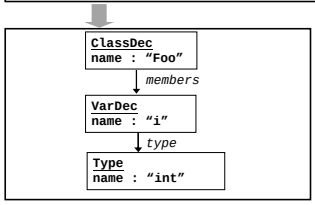
```
Class cl = Foo.class;
ClassDec c =`[class #[cl.getName()]{ #v }];
```

Basic Operators

- Operators:
  - quote:
  - unquote:

```
VarDec v = `[ int i; ];
```

```
Class cl = Foo.class;
ClassDec c =`[class #[cl.getName()]{ #v }];
```



Structured Meta-Programming

- Complete Type System for code constructs:
  - Structured: *well-typed generator implies syntactically correct generated programs*

```
VarDec v = `[ int x; ];

Expr e = `[ if (#v) x++; ];        // BAD!

ClassDec c = `[ class C { #v } ];  // Ok
```

  - MAJ type system will reject the syntactically illegal tree

Type Inference

```
VarDec v = `[ int i; ];
ClassDec c =
   `[   class Foo {
           #v
        }
    ];
```


Type Inference

```
infer v = `[ int i; ];
infer c =
   `[   class Foo {
           #v
        }
    ];
```

- MAJ supports "infer" keyword
- AST types inferred at variable initialization
- In fact, much more is inferred:

```
VarDec v = `(VarDec)[ int i; ];
ClassDec c =
    `(ClassDec)[ class Foo {
                    #(VarDec)[v]
                 }
];
```


DSLs

How to use Meta-AspectJ to implement simple (annotation-based) extensions to Java

- Meta-AspectJ (MAJ): a language for generating AspectJ (and, of course, Java) programs
- Simple, unobtrusive DSLs are important:
  - Full language extensibility can be confusing and complex.
  - Java 1.5 annotations can express simple DSLs
- Applications: implementing annotation-based DSLs with MAJ is simple
  - Interface filler and connection pooling example DSLs

A DSL Implementation Philosophy

- "If you have a choice, design a generator as a tool, not as a language"
  - user should not be locked in
    - program in extended language should also be valid for unaware base-language tools (e.g., compilers, debuggers)
    - programmers should feel that generated code can be replaced by a hand-written library at will
  - arbitrary language extension is confusing
    - "tower of Babel"
    - add-on language constructs typically do not nest or compose (horrible language design)

Front-End: Java Annotations

- In Java 1.5 one can annotate definitions with user-defined annotations

```
@TwoDim ({3, 7})
class A { ... }
```

- Unobtrusive syntax additions
- Annotations can be read through reflection, are maintained in bytecode files
- Used commonly to implement simple DSLs
  - distributed computing and persistence (J2EE) was the major application

Translation: via Generated Aspects

- Use MAJ to generate an AspectJ aspect that transforms the original program
  - through AspectJ "introductions" of new members
  - through AspectJ code interposition at field access, method call, etc.
- AspectJ used as an "assembly language" for expressing code modifications
  - the back-end language of a generator
- *MAJ-based generators avoid the complexity of pattern matching on ASTs or flow graphs by using AspectJ*

AspectJ: Overview

- AspectJ is an extension of Java
  - It is the flagship tool of aspect-oriented programming (but you can forget that)
  - Offers a vocabulary for expressing program transformations
    - advice (modification of class *behavior*)
    - declarations (modification of class *structure*)

```
aspect CaptureUpdateCallsToA {
    static int num_updates = 0;

    pointcut updates(A a):
       target(a) && call(public * update*(..));

    after(A a): updates(a) {
        num_updates++;
    }
}
```


Applications

How to use Meta-AspectJ to implement simple (annotation-based) extensions to Java

- Meta-AspectJ (MAJ): a language for generating AspectJ (and, of course, Java) programs
- Simple, unobtrusive DSLs are important:
  - Full language extensibility can be confusing and complex.
  - Java 1.5 annotations can express simple DSLs
- Applications: implementing annotation-based DSLs with MAJ is simple
  - Interface filler and connection pooling example DSLs

Example I: Filling Empty Methods for Interface Conformance

- Common pattern in Java:

```
private class SomeListener extends BasicListener
implements MouseListener, MouseMotionListener
{
  public void mousePressed (MouseEvent event) {
    ... // do something
  }
  public void mouseDragged (MouseEvent event) {
    ... // do something
  }
  // the rest are not needed. Provide empty bodies
  public void mouseClicked (MouseEvent event) {}
  public void mouseReleased (MouseEvent event) {}
  public void mouseEntered (MouseEvent event) {}
  public void mouseExited (MouseEvent event) {}
  public void mouseMoved (MouseEvent event) {}
}
```


Simple DSL

- Use an annotation to relax interface conformance requirements
  - provide empty methods (or returning null or dummy value)

```
@Implements ({"MouseListener","MouseMotionListener"})
public class SomeListener {
  public void mousePressed (MouseEvent event) {
    ... // do something
  }
  public void mouseDragged (MouseEvent event) {
    ... // do something
  }
}
}
```

DSL Implementation in MAJ

- About 200 lines
- Use reflection, generate custom aspect
  - to add new method

```
infer newMethod =
  `[ public void #methodName (#formals) {} ];
aspectMembers.add(newMethod);
```

  - to add “implements” clause(s)

```
infer dec = `[declare parents:
  #[c.getName()] implements #[iface.getName()];];
```

- This task would be impossible with Java generics or plain AspectJ and more tedious with traditional meta-programming tools

Generated Aspect

```
public aspect SomeListenerImplementsAspect1 {
  void SomeListener.mouseClicked(MouseEvent e) {}
  void SomeListener.mouseEntered(MouseEvent e) {}
  void SomeListener.mouseExited(MouseEvent e)  {}
  void SomeListener.mouseMoved(MouseEvent e)   {}
  void SomeListener.mouseReleased(MouseEvent e){}

  declare parents:
   SomeListener implements MouseListener;
  declare parents:
   SomeListener implements MouseMotionListener;
}
```


Example II: Connection Pooling

- A DSL for specifying that objects of a type should be “pooled” and re-used

```
@pooled(mgr=pooled.PoolTypes.BASIC, max=10, min=2)
public class DBConnection {
  @constructor
  public DBConnection(String dbURI,String userName,
                      String password ) { ... }
  @request
  public void open() { ... }
  @release
  public void close() { ... }
}
```

- Entire implementation: ~400 LOC
  - approach: proxying to intermediate type

That’s all, folks!

http://www.cc.gatech.edu/maj

ERROR: undefined
OFFENDING COMMAND:

STACK: