Meta-AspectJ

COMP6209

Julian Rathke
jr2@ecs.soton.ac.uk
Lecture Goals

• sketch Meta-AspectJ language
  – mostly by examples
• show use of AspectJ as intermediate language
  – for program transformation / generation
• introduce *quote/unquote* mechanism
Meta-AspectJ

Meta-AspectJ is a language tool for generating Java and AspectJ programs using code templates.

- *meta-programming* language
- built on top of AspectJ (hence Java)
- guarantees syntactic hygiene
  - use of typed syntax trees
  - SafeGen extension guarantees referential hygiene

Smaragdakis et al., 2004
Meta-programming requires representing programs as data objects.

Definition: A *meta-program* is a program that *represents and manipulates programs as data objects at its run-time*.

- parser, compiler, …
- profiler, partial evaluator, …
- code generator

Definition: A *meta-programming language* provides language support to represent programs as data objects (*reflection*).

- Java: Reflection API
- Prolog: `clause/3, assert/1, retract/1`
Meta-programming requires representing programs as data objects. (cont’d)

Program representation methods:

• text ⇒ bad
  
  ```
  main(char* a) {
    char* a="main(char* a){char* a=%c%s%c;printf(a,46,a,46);}";
    printf(a,46,a,46);
  }
  ```

• objects ⇒ better, but still clumsy
  – typically as *abstract syntax tree* (AST)
  – typically based on compiler AST exposed by reflection
  – typically built using constructors

• AST with quote/unquote
Quote/unquote representation of programs as data objects.

Basic Idea 1: *Let the compiler build the AST!*

- use *concrete syntax* instead of AST constructors
- tell compiler to return AST (instead of compiling): *quote* program text

\[
\text{VarDec } v = \left[ \text{int } i; \right];
\]

AST

\[
\begin{align*}
\text{VarDec} & \quad \text{name : "i"} \\
\text{Type} & \quad \text{name : "int"}
\end{align*}
\]

Note different language levels:
1. object language (AspectJ)
2. glue language (Meta-AspectJ)
3. meta language (Meta-AspectJ)
Quote/unquote representation of programs as data objects. (cont’d)

Basic Idea II: *Splice meta-level values into AST!*

- turn meta-language computation results into objects
- tell compiler to copy AST: *unquote* expressions (vars)

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

- meta-level expression / variable
- unquote brackets / operator

```
ClassDec name : “Foo”
VarDec name : “i”
Type name : “int”
```

members type
Quote/unquote representation of programs as data objects. (cont’d)

Additional operators:

• *emit*: turn data object into source code (program text)
• *eval*: turn data object into object code
• *run*: execute data object (runtime code generation)
Quote/unquote in Meta-AspectJ

More complicated in languages with richer syntax:

• *quote* must parse program text
  – multiple-entry parser for different syntactic categories
  – different quote operators:
    
    ```
    Dcl v = `(Dcl)[ int x=0; ];
    ```

• *unquote* must ensure well-formed syntax-trees

```
Stm s = `(Stm)[ if (#v) x++; ];
Dcl c = `(Dcl)[ class C { #v } ];
```
Quote/unquote in Meta-AspectJ

More complicated in languages with richer syntax:

• *quote* must parse program text
  – multiple-entry parser for different syntactic categories
  – different quote operators:
    ```
    Dcl v = `(Dcl)[ int x=0; ];
    ```

• *unquote* must ensure well-formed syntax-trees
  – syntactic categories of unquoted vars must be known
    ```
    Stm s = `(Stm)[ if (#(Dcl)v) x++; ]; // bad
    Dcl c = `(Dcl)[ class C { #(Dcl)v } ]; // OK
    ```
Quote/unquote in Meta-AspectJ

More complicated in languages with richer syntax:

• *quote* must parse program text
  – multiple-entry parser for different syntactic categories
  – different quote operators:
    
    ```java
    infer v = '[ int x=0; ];
    ```

• *unquote* must ensure well-formed syntax-trees
  – syntactic categories of unquoted vars must be known
    ```java
    infer s = '[ if (#v) x++; ]; // bad
    infer c = '[ class C { #v } ]; // OK
    ```

• Meta-AspectJ supports type inference of quotations
  – parser becomes type-dependant
  – sometimes need tagging to disambiguate
Meta-programming requires manipulating programs as data objects.

- *quote/unquote* is a representation mechanism only
- Meta-programming needs support for
  - parsing existing programs
  - traversing ASTs
  - pattern matching over ASTs
  - manipulating ASTs

⇒ this can be the most complex part of a generator!
⇒ *accidental* complexity: not original purpose
Meta-AspectJ builds on AspectJ to avoid accidental complexity.

Basic Idea: **Generate Aspects!**
- generated aspects implement generator functionality
  - aspects constructed using `quote/unquote`
- AspectJ implements generator mechanism

Schema:
void generateTrivialLogger(String cn, mn) {
    infer aspectCode =
        
        'package MyPackage;
        aspect #[cn + "_" + mn + "Logger"] {
            before : call(* #cn.#mn(..)) {
                out("calling method " + #mn + 
                    " in class " + #cn);
            }
        }';

    System.out.println(aspectCode.unparse());
}
A Trickier Meta-AspectJ Example

Generate a serialization aspect:

If a class is annotated with \texttt{@remote}, then make each argument type of each method implement \texttt{java.io.Serializable}, unless it is primitive (or already does so).

\begin{verbatim}
@remote class C {
    public void m1(Car c) { ... }
    public void m2(int i, Tire t) { ... }
    public void m3(float f, Seat s) { ... }
}
\end{verbatim}

package gotech.extensions;
aspect CSerializableAspect {
    declare parents: Car implements java.io.Serializable;
    declare parents: Tire implements java.io.Serializable;
    declare parents: Seat implements java.io.Serializable;
}
public class MAJGenerate {
    public static void genSerializableAspect(Class inClass) {
        String aspectName = inClass.getName() + "SerializableAspect";
        infer serializedAspect = '[aspect #aspectName {}];
        for (int meth = 0; meth < inClass.getMethods().length; meth++) {
            Class[] methSignature = inClass.getMethods()[meth].getParameterTypes();
            for (int parm = 0; parm < methSignature.length; parm++) {
                if (!methSignature[parm].isPrimitive() &&
                    !Serializable.class.isAssignableFrom(methSignature[parm]))
                    serializedAspect.addMember(
                        '[ declare parents:
                            #[methSignature[parm].getName()] implements java.io.Serializable;
                        ]);
            } // for all params
        } // for all methods
        infer compU = '[ package gotech.extensions;
                            #serializedAspect
                        ];
        System.out.println(compU.unparse());
    }
}
Encoding meta-programming as meta-programming language constructs.

Observation: most meta-programming
• iterates over lists in AST
• checks AST types
⇒ accidental complexity

Solution: new meta-programming language (SafeGen)

```java
#defgen makeInterface (Class c) {
  interface I {
    #foreach(Method m : MethodOf(m,c)) {
      void #[m] ();
    }
  }
}
```
A Trickier SafeGen Example

Generate a delegator:

*For each class, generate a delegator that extends the class, and for each public method contains a method with the same signature that simply calls the original method.*

```java
#defgen makeDelegator( input(Class c) => !Abstract(c) ) {
#foreach( Class c : input(c) ) {
    public class Delegator extends #[c] {
        #foreach( Method m : MethodOf(m, c) & !Private(m)) {
            #[m.Modifiers] #[m.Type] #[m] ( #[m.Formals] ) {
                return super.#[m](#[m.ArgNames]);
            }
        }
    }
}
```
Safe Generators

Is the simple SafeGen generator safe?

```plaintext
#defgen makeInterface (Class c) {
  interface I {
    #foreach (Method m : MethodOf(m, c)) {
      void #[m] ();
    }
  }
}
```

No: if \( c \) contains overloaded methods, \( I \) will contain duplicate definitions.
Safe Generators (cont’d)

Safety can be checked by translation into logic:
• generators become formulas
• safety conditions become formulas

\[
\forall [m_1, m_2], \\
\quad \Rightarrow (\land (\text{method}(m_1), \\
\quad \quad \quad \text{method}(m_2), \\
\quad \quad \quad \text{equal}(\text{DeclaringClass}(m_1), \\
\quad \quad \quad \quad \text{DeclaringClass}(m_2)), \\
\quad \quad \quad \text{equal}(\text{Name}(m_1), \text{Name}(m_2)), \\
\quad \quad \quad \text{equal}(\text{Formals}(m_1), \text{Formals}(m_2))), \\
\quad \quad \quad \text{equal}(m_1, m_2))
\]

• checking becomes theorem proving
Further Reading

• Yannis Smaragdakis, Shan Shan Huang, David Zook: *Program generators and the tools to make them*. PEPM 2004:92-100.


Further Reading


Conclusions

• Combination of meta-programming and AOP
  – adds fine-grained control to AOP
  – leverages AOP as backend
• Layers of meta-programming languages can provide
  – more abstraction
  – more safety