Backstage Java

Making a Difference in Metaprogramming

Zachary Palmer, Scott F. Smith

Hong Hai Chu

May 8, 2012

Table of contents

Metaprogramming

- C preprocessor macros
- C++ templates
- (Scheme macros)

Weakness: absence of contextual information at the call site

Backstage Java (BSJ)

BSJ metaprograms have the following properties:

- Non-local changes are effected without incurring confusing side-effects
- Execution order is dependency-driven to retain determinism in light of non-locality
- Conflicts between independent metaprograms are automatically detected

Difference-based metaprogramming (DBM) approach

Metaprograms as transformation generators:

- AST is instrumented to record an edit script of all changes which are made to it
- BSJ compiler prepares a different input AST for each metaprogram
- Input AST contains only those changes in the edit scripts of the metaprogram's dependencies
- Metaprogram conflicts are detected when two edit scripts fail to merge over the same AST

Comparable Java Example

```
public class Person implements Comparable<Person> {
  private String givenName;
  private String middleName;
  private String surname;
  public int compareTo(Person other) {
    int c;
    c = this.surname.compareTo(other.surname);
    if (c != 0) return c;
    c = this.givenName.compareTo(other.givenName);
    if (c != 0) return c;
    c = this.givenName.compareTo(other.middleName);
    if (c != 0) return c;
    return 0;
  }
}
```

Figure 3. Comparable Java Example

Comparable BSJ Example

```
#import static com.example.bsj.Utils.*;
public class Person {
  private String givenName;
  private String middleName;
  private String surname;
  [:
    generateComparedBy(context,
      <:surname:>, <:givenName:>, <:middleName:>);
  :]
}
```

Figure 1. Comparable BSJ Example

What's really inside

```
public class Utils {
  public static void generateComparedBy(Context<?,?> context,
      IdentifierNode... vars) {
    ClassDeclarationNode n = context.getAnchor().
      getNearestAncestorOfType(ClassDeclarationNode.class);
    BsjNodeFactory factory = context.getFactory();
    BlockStatementListNode list =
        factory.makeBlockStatementListNode();
    list.add(<:int c;:>);
    for (IdentifierNode var : vars) {
      list.add(<:c = this.~:var.deepCopy(factory):~.
        compareTo(other.~:var:~);:>);
      list.add(<:if (c != 0) return c;:>);
    }
    list.add(<:return 0;:>);
    n.getBody().getMembers().addLast(<:
        public int compareTo(Person other) { ~:list:~ }
      :>);
    n.getImplementsClause().addLast(<: Comparable
        <~:n.getIdentifier().deepCopy(factory):~> :>);
  }
}
```

Figure 2. BSJ Utility Class

Scope limitation

```
public class Example {
  [:
    // next statement produces an error!
    context.getAnchor().getNearestAncestorOfType(
      CompilationUnitNode.class).getTypeDecls().add(
        <:class Extra {}:>);
  :]
}
```

Figure 4. Limiting the Scope of Change

Dual-write conflict

```
public class SimpleConflict {
  public void foo() {
    [:
      // #depends a; /* uncomment to resolve conflict */
      context.getPeers().add(0, <:System.out.print("A");:>);
    :]
    [:
      // #target a; /* uncomment to resolve conflict */
      context.getPeers().add(0, <:System.out.print("B");:>);
    :]
  }
}
```

Figure 5. Simple Conflict

Dual-write conflict

```
public class SimpleConflict {
  public void foo() {
    [:
      // #depends a; /* uncomment to resolve conflict */
      context.getPeers().add(0, <:System.out.print("A");:>);
    :]
    [:
      // #target a; /* uncomment to resolve conflict */
      context.getPeers().add(0, <:System.out.print("B");:>);
    :]
  }
}
```

Figure 5. Simple Conflict

It is sufficient to execute each metaprogram over the original AST in turn

Read-write conflict

```
class X {
  [:
    /* For each class defined in this compilation unit, adds a
     * field to this class of the same name but lower-cased. */
    // #depends foo; /* uncomment to include y field in X */
     ...
  :]
}
[:
  #target foo;
  context.addAfter(<:class Y{}:>);
:]
```

Figure 6. Apparent Read-Write Conflict Example

Read-write conflict

```
class X {
  [:
    /* For each class defined in this compilation unit, adds a
     * field to this class of the same name but lower-cased. */
    // #depends foo; /* uncomment to include y field in X */
     ...
  :]
}
[:
  #target foo;
  context.addAfter(<:class Y{}:>);
:]
```

Figure 6. Apparent Read-Write Conflict Example

Solution: difference-based metaprogramming

Dependency graph

```
[: #target a; :]                    // metaprogram 1
[: #target a; :]                    // metaprogram 2
[: #target a, b; :]                 // metaprogram 3
[: #target c; #depends a; :]        // metaprogram 4
[: #target c; #depends b; :]        // metaprogram 5
[: #depends c; :]                   // metaprogram 6
[: :]                               // metaprogram 7
```

Figure 13. Execution Order Example

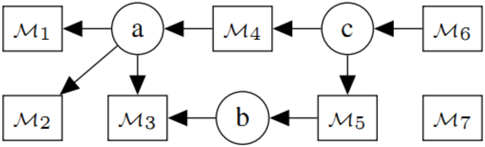
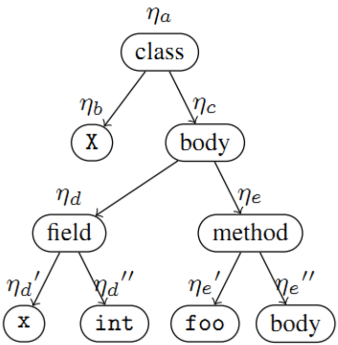


Figure 14. Example Dependency Graph

Edit script

```
public class X {
  private int x;
  public void foo() { }
}
```

(a) Example Source



(b) Simplified Example AST

η_a	$\mapsto$	id $\mapsto$ η_b body $\mapsto$ η_c
η_b	$\mapsto$	identifier $\mapsto$ "X"
η_c	$\mapsto$	$[(\triangleright, \mathcal{M}, \mathcal{S}), (\eta_d, \mathcal{M}, \mathcal{S}),$ $(\eta_e, \mathcal{M}, \mathcal{S}), (\triangleleft, \mathcal{M}, \mathcal{S})]$
η_d	$\mapsto$	id $\mapsto$ $\eta_d{}'$ type $\mapsto$ $\eta_d{}''$
$\eta_d{}'$	$\mapsto$	identifier $\mapsto$ "x"
$\eta_d{}''$	$\mapsto$	type $\mapsto$ INT
η_e	$\mapsto$	id $\mapsto$ $\eta_e{}'$ body $\mapsto$ $\eta_e{}''$
$\eta_e{}'$	$\mapsto$	identifier $\mapsto$ "foo"
$\eta_e{}''$	$\mapsto$	$[(\triangleright, \mathcal{M}, \mathcal{S}), (\triangleleft, \mathcal{M}, \mathcal{S})]$

(c) Record Encoding

Ackermann function

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Java version

```java
public class AckermannFunction {
  public static final AckermannFunction SINGLETON =
    new AckermannFunction();
  private AckermannFunction() {}
  private static class EvaluateCacheKey {
    private BigInteger m;
    private BigInteger n;
    ...
  }
  private Map<EvaluateCacheKey, BigInteger> evaluateCache =
      new HashMap<EvaluateCacheKey, BigInteger>();
  public BigInteger evaluate(BigInteger m, BigInteger n) {
    final EvaluteCacheKey key = new EvaluateCacheKey(m, n);
    ...
    return result;
  }
  public BigInteger calculateEvaluate(BigInteger m,
      BigInteger n) {
    if (m.compare(BigInteger.ZERO) < 0 ||
        n.compare(BigInteger.ZERO) < 0)
      throw new ArithmeticException("Undefined");
    if (m.equals(BigInteger.ZERO))
      return n.add(BigInteger.ONE);
    if (n.equals(BigInteger.ZERO))
      return evaluate(m.subtract(BigInteger.ONE), 1);
    return evaluate(m.subtract(BigInteger.ONE),
      evaluate(m, n.subtract(BigInteger.ONE)));
  }
}
```

(a) Abbreviated Java Source

BSJ version

```
@@MakeSingleton
public class AckermannFunction {
  @@Memoized @@BigIntegerOperatorOverloading
  public BigInteger evaluate(BigInteger m, BigInteger n) {
    if (m < 0 || n < 0)
      throw new ArithmeticException("Undefined");
    if (m == 0) return n + 1;
    if (n == 0) return evaluate(m - 1, 1);
    return evaluate(m - 1, evaluate(m, n - 1));
  }
}
```

(b) BSJ Source

Meta-annotation ordering

```
#import edu.jhu.cs.bsj.stdlib.metaannotations.*;
@@GenerateEqualsAndHashCode
@@GenerateToString
@@ComparedBy({<:rank:>,<:suit:>})
@@GenerateConstructorFromProperties
public class Card {
  public enum Rank { TWO, ..., KING, ACE }
  public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
  @@Property(readOnly=true) private Rank rank;
  @@Property(readOnly=true) private Suit suit;
}
```

Figure 8. BSJ Playing Card Class

User defined meta-annotation

```
public class Property
    extends AbstractBsjMetaprogramMetaAnnotation {
  public Property() {
    super(Arrays.asList("property"), Arrays.<String>asList());
  }
  protected void execute(Context... context) {
    /* code here to insert getters and setters */ ...
  }
  private boolean readOnly = false;
  @BsjMetaAnnotationElementGetter
  public boolean getReadOnly() { return this.readOnly; }
  @BsjMetaAnnotationElementSetter
  public void setReadOnly(boolean r) { this.readOnly = r; }
  @Override public void complete() { }
}
```

Memoize expansion

```
@@MakeSingleton
public class AckermannFunction {
  @@GenerateEqualsAndHashCode @@GenerateConstructorFromProperties
  private static class EvaluateCacheKey {
    @@Property(readOnly=true) private BigInteger m;
    @@Property(readOnly=true) private BigInteger n;
  }
  @@Memoized @@BigIntegerOperatorOverloading
  public BigInteger evaluate(BigInteger m, BigInteger n) { ··· }
  ...
}
```

Figure 10. Ackermann Function - Memoize Expansion

Injection conflict

```
[:
  #target x;
  context.addAfter(<: [: #target y; :] :>);
:]
[: #target y; :]
[: #depends y; :]
```

Figure 11. Injection Conflict Example

Injection conflict

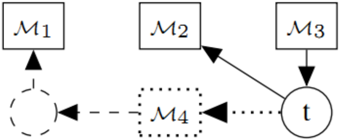


Figure 22. Example Injection Conflict

Injection conflict

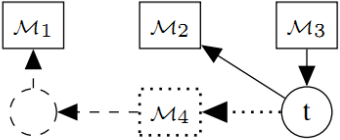


Figure 22. Example Injection Conflict

Produce a compile error if any legal execution orders (including those we are not running) become invalid during the process of compilation.

Restrictions

- **Unordered metaprograms must never communicate.** This includes *all* forms of communication. For instance, a metaprogram must not write to a file that another metaprogram reads or use the same global variables that another metaprogram uses.
- **Metaprograms may not use any external resource to obtain access to AST nodes.** The creation of new nodes must be done via the node factory provided by the metaprogram's context; access to existing nodes must be obtained by following references from the metaprogram's anchor.

Code literal ambiguation

```
LocalVariableDeclarationNode n1 = <: int x = 0; :>;
```

(a) With Code Literals

```
LocalVariableDeclarationNode n2 =
  factory.makeLocalVariableDeclarationNode(
    factory.makePrimitiveTypeNode(PrimitiveType.INT),
      factory.makeVariableDeclaratorListNode(
        factory.makeVariableDeclaratorNode(
          factory.makeIdentifierNode("x"),
          factory.makeIntLiteralNode(0))));
```

(b) Without Code Literals

Figure 12. With and Without Code Literals

Code literal ambiguation

```
LocalVariableDeclarationNode n1 = <: int x = 0; :>;
```

(a) With Code Literals

```
LocalVariableDeclarationNode n2 =
  factory.makeLocalVariableDeclarationNode(
    factory.makePrimitiveTypeNode(PrimitiveType.INT),
      factory.makeVariableDeclaratorListNode(
        factory.makeVariableDeclaratorNode(
          factory.makeIdentifierNode("x"),
          factory.makeIntLiteralNode(0))));
```

(b) Without Code Literals

Figure 12. With and Without Code Literals

There is more than one interpretation:

- a local variable
- a class member field
- an interface constant

Code lifting

1. Code literals are represented by a type family which directly represents the ambiguity in the parse.
2. Forest of parses is then typechecked and, if exactly one parse successfully typechecks, it is taken as the meaning of the code literal.
3. Each of the possible parses is lifted into metaprogram code that constructs the indicated AST.

What has been done

The existing implementation is comprised of three parts:

- BSJ API: diagnostic interfaces, utilities, and over 200 types of AST nodes,
- Reference implementation of the API: 54000 lines of code and 193000 lines of generated code,
- BSJ standard libraries:
  - meta-annotations such as @@Memoized and @@BigIntegerOperatorOverloading,
  - meta-annotations implementing design patterns such as Builder, Observer and Proxy,
  - useful code manipulations such as loop unrolling and method delegation

Future work

- BSJ metaprograms are currently quite verbose
- The call sites for BSJ always include explicit delimiter syntax
- An IDE comparable to those available to Java programmers (Eclipse plugin) is necessary