Backstage Java

Making a Difference in Metaprogramming

Zachary Palmer
The Johns Hopkins University
zachary.palmer@jhu.edu

Scott F. Smith
The Johns Hopkins University
scott@jhu.edu

Abstract

We propose Backstage Java (BSJ), a Java language extension which allows algorithmic, contextually-aware generation and transformation of code. BSJ explicitly and concisely represents design patterns and other encodings by employing *compile-time metaprogramming*: a practice in which the programmer writes instructions which are executed over the program's AST during compilation. While compile-time metaprogramming has been successfully used in functional languages such as Template Haskell, a number of language properties (scope, syntactic structure, mutation, etc.) have thus far prevented this theory from translating to the imperative world. BSJ uses the novel approach of *difference-based* metaprogramming to provide an imperative programming style amenable to the Java community and to enforce that metaprograms are consistent and semantically unambiguous. To make the feasibility of BSJ metaprogramming evident, we have developed a compiler implementation and numerous working code examples.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords Java, metaprogramming, macros, design patterns, software merging, difference-based metaprogramming

1. Introduction

When selecting features for a programming language, it's impossible to keep everyone happy. Inevitably, a programmer will want for an abstraction which would make the implementation of a program much easier, more concise, and more maintainable. When such abstractions are not provided by the language, programmers resort to implementing each case manually. Design patterns in Java are one example of this strategy: while each pattern describes a solution to a problem abstractly, each instance of a design pattern in code must be reified with context-specific information [6].

This manual implementation is tedious, however, and macro systems are often employed to reduce its maintenance burden. One simple example is the token sequence replacement system such as in C preprocessor macros. But such macro systems have many well-known flaws: because operations occur on the token level, for instance, macros which produce low-precedence operators at a call site with high-precedence operators can introduce subtle bugs.

More advanced generative techniques can be seen in C++ templates and Scheme macros. C++ templates, while originally intended for type and function definitions, are Turing-complete [22] and can be used as a compile-time code generation language. Because this was never the intended use of the system [15], the resulting templates have an obtuse and unintuitive syntax. Scheme macros operate by AST substitution (unlike most macro systems) and have a much more tractable syntax. Indeed, language extensions have been constructed in Scheme using the macro system.

One weakness of all of the aforementioned systems is the absence of contextual information at the call site. Object-oriented languages such as Java require changes to multiple AST subtrees to express more complex properties (e.g., that a given class of objects is ordered by a specific member field). OJ [20] is a Java language extension which permits the definition of modifiers which abstract the notion of design patterns. The OJ compiler provides a meta-object protocol (MOP) which executes code associated with these modifiers at compile time, allowing it to effect *non-local changes*: alterations to the AST in positions other than that of the call site itself. This form of *ad hoc* abstraction allows for greater semantic meaning than any statically-typed macro system, permitting the generation of well-typed proxies, adapters, and other common design patterns.

Unfortunately, OJ is very permissive in this regard; it is not immediately clear from the call site which portions of code could be affected. A keyword modifying a local variable, for instance, could make modifications to any AST node, including those contained in other compilation units. This behavior could easily lead to code with confusing side-effects. While the imperative programming community expects to write code in the style of mutation, this mutation must be managable and well-understood.

The primary weakness of OJ, however, is that of metaprogram execution order. If two transformations operate over the same region of code, changing their execution order may change the resulting AST. OJ does not clearly define its expansion order, meaning that an OJ program may have multiple valid meanings. This form of nondeterminism is unacceptable from a compiler.

The functional programming community has seen significant benefits from the more disciplined technique of *compile-time metaprogramming* as a means of *ad hoc* abstraction. In languages such as MacroML [7] and Template Haskell [17], Scheme-like code generation is reconciled with a static type system. Compile-time metaprograms may also inspect definitions which are in scope at the call site, providing valuable contextual information. Determinism is enforced by forbidding non-local changes and by executing metaprograms in a predefined order dictated by syntactic structure. But this theory is non-trivial to apply to imperative and object-oriented languages like Java for precisely that reason: expressing non-trivial facts about Java classes in an imperative style is not possible without non-local changes (as in the ordering example mentioned above).

We introduce Backstage Java (BSJ), a language which brings the sophistication of compile-time metaprogramming to Java. BSJ metaprograms have the following properties:

(a) Non-local changes are effected without incurring confusing side-effects

(b) Execution order is dependency-driven to retain determinism in light of non-locality

(c) Conflicts between independent metaprograms are automatically detected

To achieve the above, BSJ uses a *difference-based* metaprogramming approach which, to our knowledge, is novel in metaprogramming literature. Rather than viewing metaprograms as program transformations, we view them as transformation *generators*: the AST is instrumented to record an *edit script* of all changes which are made to it in a strategy similar to operation-based merging [11, 12]. Then, rather than executing each metaprogram against a single AST in turn, the BSJ compiler prepares a different input AST for each metaprogram. That input AST contains only those changes in the edit scripts of the metaprogram's dependencies, guaranteeing that the metaprogram cannot see changes applied by metaprograms on which it does not depend. Metaprogram conflicts are detected when two edit scripts fail to merge over the same AST.

This metaprogramming model has several advantages. Non-local changes, necessary for metaprogramming in an imperative style, are possible. The input for each metaprogram is understood simply in terms of its declaration, making metaprogram semantics much clearer. Order of execution may be controlled but is inferred by default, allowing flexibility without placing unnecessary burdens on the metaprogrammer. In the event of ambiguity, the system can easily identify the pair of metaprograms involved and the AST subtree on which they disagree.

```
 1 #import static com.example.bsj.Utils.*;
 2 public class Person {
 3   private String givenName;
 4   private String middleName;
 5   private String surname;
 6   [:
 7     generateComparedBy(context,
 8       <:surname:>, <:givenName:>, <:middleName:>);
 9   :]
10 }
```

Figure 1. `Comparable` BSJ Example

This paper describes the design, semantics, and implementation of BSJ. Section 2 provides brief discussions of implementation challenges and solutions. Section 3 describes how non-local changes are made safe by the implementation of a metaprogram edit script algebra and conflict detection system (which is proven correct in Appendix A). Section 4 details BSJ quasiquoting syntax and the use of a type family to disambiguate it. Section 5 discusses the BSJ language specification and the compiler reference implementation. We conclude with an analysis of related work in Section 6 and a discussion of future research in Section 7.

2. Overview

To discuss our difference-based metaprogramming model, we will provide source examples in BSJ. Sections 2.1 and 2.2 provide a general introduction to BSJ syntax; Section 2.3 introduces difference-based metaprogramming and highlights its importance. The remainder of the overview is devoted to explaining other ambiguities which must be resolved in BSJ and outlining our solutions to them.

2.1 Basic Execution Semantics

Figure 1 presents a class in BSJ which contains three string fields representing a person's names. A metaprogram appears in this class which makes the class comparable first by surname, then by given name, and finally by middle name. A utility method is used to abstract the concept of a lexicographical class ordering; this method can be seen in Figure 2. Java code equivalent to Figure 1 can be seen in Figure 3

The delimiters `[:` (line 6) and `:]` (line 9) of Figure 1 mark the beginning and end of an explicit BSJ metaprogram. The intervening lines form the body of the metaprogram (lines 7 and 8). The compilation process strips the metaprograms from the AST before they are executed, leaving behind a single AST leaf node – called an *anchor* – which is provided to the metaprogram as input.

During compilation, the statements in the metaprogram's body are executed. In the case of the metaprogram in Figure 1, this will result in the addition of a `compareTo` method to the metaprogram's class. The environment in which the metaprogram is executed includes a binding for the variable `context` (used on line 7), which is a reference allowing access to the metaprogram's anchor, the node factory, and other metaprogramming facilities.

```
1  public class Utils {
2    public static void generateComparedBy(Context<?,?> context,
3        IdentifierNode... vars) {
4      ClassDeclarationNode n = context.getAnchor().
5        getNearestAncestorOfType(ClassDeclarationNode.class);
6      BsjNodeFactory factory = context.getFactory();
7      BlockStatementListNode list =
8          factory.makeBlockStatementListNode();
9      list.add(<:int c;:>);
10     for (IdentifierNode var : vars) {
11       list.add(<:c = this.~:var.deepCopy(factory):~.
12         compareTo(other.~:var:~);:>);
13       list.add(<:if (c != 0) return c;:>);
14     }
15     list.add(<:return 0;:>);
16     n.getBody().getMembers().addLast(<:
17         public int compareTo(Person other) { ~:list:~ }
18       :>);
19     n.getImplementsClause().addLast(<: Comparable
20         <~:n.getIdentifier().deepCopy(factory):~> :>);
21   }
22 }
```

Figure 2. BSJ Utility Class

```
1  public class Person implements Comparable<Person> {
2    private String givenName;
3    private String middleName;
4    private String surname;
5    public int compareTo(Person other) {
6      int c;
7      c = this.surname.compareTo(other.surname);
8      if (c != 0) return c;
9      c = this.givenName.compareTo(other.givenName);
10     if (c != 0) return c;
11     c = this.givenName.compareTo(other.middleName);
12     if (c != 0) return c;
13     return 0;
14   }
15 }
```

Figure 3. `Comparable` Java Example

The delimiters `<:` and `:>` denote a *code literal* (quasiquote) and allow metaprograms to express ASTs in the form of a code fragments. These code literals appear on line 8 of Figure 1 as well as in several places in Figure 2. In the latter figure, the delimiters `~:` and `:~` are *splicing operators* and permit the code literal to be used as a template which is then instantiated by metaprogram expressions. In this case, the arguments passed to the `generateComparedBy` method on line 8 of Figure 1 will replace the splices on lines 8 and 9 of Figure 2, generating one AST representing an assignment to a variable `c` for each identifier specified in the arguments.

BSJ AST nodes can access their parent nodes.[1] This permits the metaprogram in Figure 1 to obtain a reference to the class declaration in which it is contained. This is critical for the behavior of `generateComparedBy` because it must insert a `compareTo` implementation into the class body as well as a type in the `implements` clause. It is in this fashion that BSJ permits non-local changes to be inserted at well-defined positions in the AST. This is a necessity in a Java-like language as some semantic facts (such as the comparability of a class) must affect multiple subtrees of the AST consistently. In the example code in Figure 3, both the `implements` clause on line 1 and the `compareTo` definition starting on line 4 were added by a single metaprogram. In contrast, the simplest and most convenient macro system approach would require two cooperating macros, resulting in information duplication as well as the opportunity for the two macros to become out of sync.

[1] For this to be possible, each node object in the AST must be unique. This motivates the `deepCopy(factory)` constructions in Figure 2.

```
1  public class Example {
2    [:
3      // next statement produces an error!
4      context.getAnchor().getNearestAncestorOfType(
5        CompilationUnitNode.class).getTypeDecls().add(
6          <:class Extra {}:>);
7    :]
8  }
```

Figure 4. Limiting the Scope of Change

```
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

To prevent metaprograms from being difficult to understand, they are limited to affecting only the declaration in which they appear. In Figure 4, for example, the metaprogram at line 2 attempts to insert a new top-level class into the same file as the `Example` class. Because the metaprogram is positioned inside of the `Example` class, it is unable to affect the compilation unit outside of its declaration. Metaprograms are also forbidden from modifying compilation units other than the one in which they appear, although it is possible to insert new compilation units into an existing package.

2.2 Execution Order

Order of execution is an important consideration in any metaprogramming environment but is especially critical when admitting non-local changes. Figure 5 demonstrates the importance of execution order. Each metaprogram attempts to insert a statement at the beginning of the method. Without execution ordering, the meaning of the method is ambiguous: at runtime, it may print either "`BA`" or "`AB`" depending on the order in which the transformations executed. Any such case in which the order of metaprogram execution is unspecified but would change the resulting AST is known as a metaprogram *conflict*. This particular case is known as a *dual-write conflict* because the metaprograms are conflicting over the semantics of their write operations.

Languages such as Scheme and Template Haskell use a simple mechanism for the order of metaprogram execution: metaprograms are executed in the order in which they appear in the AST. This is intuitive in functional languages, as this is the direction in which scoped bindings flow. By

```
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

contrast, many constructs in the Java language (such as non-initializing declarations in the body of a type) have no relevant ordering; assigning meaning to the order of these declarations would be confusing and unintuitive. Furthermore, it may be undesirable; a programmer may wish the first metaprogram in Figure 5 to be able to see and react to the changes made by the second metaprogram (or even by a metaprogram in another compilation unit).

In BSJ, the programmer may resolve the conflict in Figure 5 by explicitly specifying an ordering between the metaprograms; uncommenting the first line of each metaprogram would be sufficient. The preamble clause `#target a` indicates that the metaprogram is a member of the *target* named `a`; the preamble clause `#depends a` indicates that the metaprogram should wait until all metaprograms in target `a` have executed.

While this dependency-driven ordering of execution is a valuable tool, it is not a complete solution. In practice, large numbers of metaprograms can appear in an object program and each metaprogram may have subtle side effects; thus, we cannot expect the metaprogrammer to identify conflicts manually. The BSJ compiler provides a conflict detection system (described in Section 3) which guarantees that any BSJ code which does not have exactly one valid Java equivalent will produce a compilation error. This feature is necessary for the scalability of the language and frees up the programmer to focus on developing the object program.

2.3 Difference-Based Metaprogramming

Two categories of potential conflict other than dual-write conflicts exist: *read-write* conflicts and *injection* conflicts. Injection conflicts are outlined in Section 2.5 and discussed in depth in Section 3.5. An example of an apparent read-write conflict, on the other hand, appears in Figure 6. The semantics of the first metaprogram are explained in a comment for the sake of simplicity.

In previous examples we have presented, it is sufficient to execute each metaprogram over the original AST in turn; dual-write conflicts could be detected by noticing when one metaprogram writes to the same AST node as another. In Figure 6, however, this is not true. If we execute the top metaprogram first, the body of `X` contains one variable declaration (`x`); otherwise, it contains two (`x` and `y`). In neither case do we see a dual-write conflict, but the output would still be ambiguous if the code were allowed to compile.

A previous iteration of BSJ attempted to solve this problem by monitoring the behavior of metaprograms, producing an error if a metaprogram reads a variable that another metaprogram changed. In our experience, however, the constraints describing what a metaprogram considers significant are quite complex; often, a metaprogram would iterate over a list (such as the body of a class declaration) and only act on those elements which were fields. Because the conflict detection system was unable to determine that the non-field elements were not used, this led to numerous false conflicts.

Instead, BSJ solves this problem by using *difference-based metaprogramming*. Instead of executing each of the metaprograms in Figure 6 over the same AST in some sequence, we execute each over a copy of that AST. The AST is instrumented to produce an *edit script*: a description of the changes that the metaprogram made. Software merging techniques are later used to combine these edit scripts and produce the final program. Because each metaprogram executes over a copy of the original AST, neither metaprogram can see the changes created by the other. In our example, the system would always generate code in which `X` had a single member field (`x`). The AST provided to a metaprogram reflects the edit scripts from that metaprogram's dependencies; thus, the programmer could choose to generate code in which `X` has the member fields `x` and `y` by uncommenting the `#depends` clause, making the first metaprogram depend upon the second.

This approach has two profound advantages. First, read-write conflicts are impossible in this system: by definition, a metaprogram can only observe changes applied by its dependencies. Second, reasoning about metaprograms becomes considerably easier. The behavior of a metaprogram can now be understood strictly in terms of its declaration and those of its dependencies; it is not necessary to consider any other code to understand the effect that a metaprogram will have.

It is possible using this approach that a programmer may misunderstand the semantics of the program in Figure 6, expecting it to result in a class with two member fields. While this potential for confusion exists, we view the small learning curve introduced by the dependency system's semantics as worth the considerable advantages described above.

2.4 Meta-Annotations

The explicit metaprogram syntax witnessed thus far (delimited by the `[:` and `:]` operators) is suitable for individual metaprograms. But a metaprogrammer often wishes to abstract over these metaprograms (as well as their dependencies) to represent general coding concepts. For this purpose, BSJ includes *meta-annotations*.

Recall that Java annotations are constructs prefixed with `@` which may appear as modifiers on declarations. In general, annotations have no semantic meaning; instead, preprocessors or reflective code are expected to react to them.

```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

```java
@@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

Figure 7. Ackermann Function

BSJ meta-annotations are similar but are used by the BSJ compiler and are not available at runtime. We use the prefix `@@` for disambiguation. BSJ also permits meta-annotations to represent metaprograms, encouraging a declarative metaprogramming style.

We introduce meta-annotations with a simple example: the task of writing a class to calculate the well-known Ackermann function. This function grows quickly in some cases, so we will use `BigInteger` to represent its inputs and results. Calculation of the Ackermann function is expensive, so we will use memoization for performance.

An abbreviated version of a Java source code solution appears in Figure 7a. The full source is sixty lines. By comparison, the BSJ implementation of this same routine appears in Figure 7b. The latter variation is more concise and more readable than its Java counterpart; the patterns used in the code (singleton and memoization) are explicitly represented and existing Java syntax is semantically extended to make `BigInteger` operations more legible. The BSJ standard libraries, utilized in these figures, provides definitions of the meta-annotations such as `@@Memoized`. BSJ programmers are encouraged to define their own meta-annotations for domain-specific patterns or other similar uses.

```java
1  #import edu.jhu.cs.bsj.stdlib.metaannotations.*;
2  @@GenerateEqualsAndHashCode
3  @@GenerateToString
4  @@ComparedBy({<:rank:>,<:suit:>})
5  @@GenerateConstructorFromProperties
6  public class Card {
7    public enum Rank { TWO, ..., KING, ACE }
8    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
9    @@Property(readOnly=true) private Rank rank;
10   @@Property(readOnly=true) private Suit suit;
11 }
```

Figure 8. BSJ Playing Card Class

```java
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() { }
}
```

Figure 9. `@@GenerateConstructorFromProperties`

Meta-annotations also generalize over targets and dependencies. Consider the playing card class shown in Figure 8. The metaprograms which generate the getters for properties (on lines 9 and 10) must execute before the metaprogram which generates the constructor (on line 5). This dependency is declared in the meta-annotations' definitions, reducing clutter at the call site.

Meta-annotations are defined by creating an implementation of the interface `BsjMetaprogramMetaAnnotation` which specifies methods defining the metaprogram's targets, dependencies, and execution behavior. Meta-annotations execute in the same fashion as explicit metaprograms and obey the same conflict detection rules.

For an example, consider the abbreviated definition of `@@Property` appearing in Figure 9. This meta-annotation abstracts the Java property idiom: the creation of getter and setter methods for a field. The arguments to the super constructor specify that each use of this meta-annotation is a member of the `property` target; this allows other metaprograms to depend on this meta-annotation.

2.5 Metaprogram Injection

Because BSJ metaprograms can insert nodes into the AST, they can create and insert new metaprograms. This results in a very natural expression of complex pattern definitions. For example, consider the Ackermann implementation in Figure 7b. The `@@Memoized` meta-annotation is responsible for much of the generated code because it must create a data structure representing the cache key: a tuple of two `BigIntegers`. If the compiler expands the

```
@@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

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

Figure 11. Injection Conflict Example

`@@Memoized` meta-annotation first, the AST would then resemble Figure 10. (`@@Memoized` is italicized to indicate that its metaprogram has executed.) The newly-inserted code uses meta-annotations to define the cache key data structure.

The process of adding new metaprograms during the execution of a metaprogram is termed *metaprogram injection*. Injection is common among metaprogramming systems but introduces complications for a dependency system. For example, consider the program in Figure 11. The top metaprogram injects a metaprogram in target y; the bottom metaprogram depends on target y. If the metaprograms execute from top to bottom, everything is fine; we will conclude that the injected metaprogram should run before the bottom one.

But at the beginning of compilation, it would also be legal for the bottom metaprogram to execute before the top one; after all, the top metaprogram is not in target y and the injected metaprogram is not yet present. After executing the bottom metaprogram, we would execute the top metaprogram only to discover that the target y is no longer satisfied! To remain consistent, the compiler must produce an error not only when such an event occurs, but when such an event *could have occurred*; that is, even if we executed the top metaprogram first, we must eventually realize that the potential for an injection conflict existed and produce an error. Our technique for doing this is described in Section 3.5.

2.6 Code Literal Disambiguation

A simple example of code literals is demonstrated in Figure 12a: the sole line uses code literal syntax to create an AST representing a variable declaration. The code in Figure 12b shows an equivalent expression using a BSJ node factory. As in other languages with quasiquoting, the code literal form is much more readable.

While code literals are clearly desirable, the complexity of the Java grammar makes parsing code literals difficult. The code literal shown in Figure 12a has more than one interpretation: it could be a local variable declaration, a class member field, or an interface constant. Metaprogramming

```
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

environments often address this problem either by providing different quasiquoting syntax for each form [2, 17, 21] or by restricting the forms that can be used [10, 26]. Unfortunately, BSJ would need dozens of different quasiquoting forms; the singleton token sequence x could represent an identifier, an enum constant declaration, or fourteen other parses.

It has been shown [3] that this problem can be solved by using the type context of the surrounding metaprogram code to disambiguate the code literal. In this approach, each of the possible parses is *lifted* into metaprogram code that constructs the indicated AST. This forest of parses is then typechecked and, if exactly one parse successfully typechecks, it is taken as the meaning of the code literal.

BSJ uses a similar approach but defers the lifting until after typechecking is complete. Code literals are represented by a type family which directly represents the ambiguity in the parse. This technique more directly integrates with the original language's type system and provides opportunities for lazy evaluation and other performance optimizations. Section 4 describes this process in detail.

3. Edit Scripts and Conflicts

In this section, we characterize difference-based metaprogramming and define its semantics. We observe that traditional metaprograms are perceived as functions converting one object program to another. That is, suppose $\mathbb{P}$ to be the set of all programs ρ; then traditional metaprograms ψ are viewed as having the type $\mathbb{P} \to \mathbb{P}$. The metaprograms $\psi_1, \ldots, \psi_n$ are then composed over an initial input program ρ, producing the resulting program ρ'. This also means, however, that each ψ_i receives as its input the result of $\psi_1, \ldots, \psi_{i-1}$ executed over ρ, which leads to the possibility of the read-write conflicts discussed in Section 2.3.

In a difference-based metaprogramming environment such as BSJ, however, we separate metaprogram logic into two parts: analysis and modification. Analysis logic involves the metaprogram examining the object program to make decisions about what actions to take; modification logic involves taking action and changing the object program. A BSJ metaprogram ϕ represents the analysis logic and has the type $\mathbb{P} \to \Delta_{\mathbb{P}}$, where $\Delta_{\mathbb{P}}$ is the set of all *edit scripts* $\bar{\delta}$. An edit script represents the modification logic as a series of transformations $\delta_1, \ldots, \delta_n$ which are performed over a program. We define a function $\vartheta(\delta, \rho) = \rho'$ which applies the transformation of an edit script element δ to a program ρ, resulting in a program ρ'. For convenience, we define $\vartheta(\bar{\delta},\rho) = \vartheta(\delta_n, (\cdots\vartheta(\delta_1,\rho)))$. Clearly, ϑ has the type $\mathbb{P} \to \mathbb{P}$ if δ is fixed. But unlike a traditional metaprogram, ϑ can only perform write operations based on the edit script element; it would not inspect the object program or make any decisions during its application.

Edit scripts are generated by providing a specific object program as input to each metaprogram ϕ which reflects ϕ's dependencies.. While the full discussion of dependencies is deferred until Section 3.1, we can informally define the *dependencies* of ϕ as the set of metaprograms which participate in targets on which ϕ depends (and the dependencies of those metaprograms, recursively). We model the AST nodes representing the original source code as being inserted by an "origin" metaprogram on which all other metaprograms depend. The input to ϕ is then $\vartheta(\bar{\delta}_n, (\cdots\vartheta(\bar{\delta}_1,\emptyset)))$ where $\emptyset$ represents an empty AST, $\bar{\delta}_i$ is the edit script produced by ϕ_i, and ϕ_1 is the origin metaprogram.

Once the edit scripts $\bar{\delta}_i$ for each metaprogram ϕ_i has been calculated, we apply them to the original program ρ to obtain the resulting program ρ'. To do so, we must select an order of application. We always do so in a fashion that observes the dependencies that each metaprogram has declared.

Definition 3.1. Given metaprograms $\{\phi_1,\ldots,\phi_n\}$, an *ordering* of these metaprograms $\bar{k} = k_1,\ldots,k_n$ is some permutation of the integers $1,\ldots,n$. The ordering $\bar{k}$ is *respectful* iff, for any metaprogram ϕ_j which depends on a target containing ϕ_i, the value i preceeds the value j in $\bar{k}$.

We then determine the final object program ρ' by calculating $\vartheta(\bar{\delta}_{k_n}, (\cdots\vartheta(\bar{\delta}_{k_1},\emptyset)))$ using a respectful ordering $\bar{k}$.

Our key objective is to ensure that all BSJ programs are unambiguous: each BSJ program either corresponds to exactly one output program or it produces a compilation error. Since the ordering in which the metaprograms' edit scripts are composed is selected arbitrarily by the compiler (such that it preserves declared dependencies), we must ensure that it has no impact on the output; every respectful ordering should produce the same result. In pursuit of this, we provide a formal definition of a metaprogram conflict below. This definition uses a concept of program equivalence; we define program equivalence to resolve a special case in Section 3.4, but a reader may safely proceed by assuming equivalence ($\cong$) to be identity until then.

Definition 3.2. Let ϕ and ϕ' be two metaprograms in a compilation of a program ρ involving n metaprograms. Let $\bar{\delta}$ and $\bar{\delta}'$ be the edit scripts produced by those metaprograms, respectively. Let $\bar{k}$ be any respectful ordering of the n metaprograms. A *conflict* exists if there exists any other respectful ordering $\bar{k}'$ of the n metaprograms such that $\vartheta(\bar{\delta}_{k_n}, (\cdots\vartheta(\bar{\delta}_{k_1},\emptyset))) \ncong \vartheta(\bar{\delta}_{k'_n}, (\cdots\vartheta(\bar{\delta}_{k'_1},\emptyset)))$.

We separate conflicts into the three categories discussed in Section 2. We specify an edit script algebra for modeling difference-based metaprogramming execution in Section 3.3 and show that this metaprogramming model makes read-write conflicts impossible. We then describe a *merge* operation in Section 3.4; this operation simultaneously performs the composition of a series of edit scripts and analyzes them to determine if any dual-write conflicts exist. Section 3.5 describes injection conflicts in detail and provides a graph analysis sufficient to detect them.

3.1 The Dependency Graph

For purposes of execution ordering and conflict detection, metaprograms and their targets are organized into a bipartite dependency graph. Each metaprogram is assigned a unique identity; we use the terms $\mathcal{M}_1$, $\mathcal{M}_2$, etc. to range over metaprogram identities and the terms `a`, `b`, etc. to range over metaprogram targets. Because the input for a metaprogram ϕ is defined in terms of the output of its dependencies, those dependencies will always be executed prior to ϕ. In effect, the dependency graph defines a partial order over the metaprograms which is used to determine order of execution. Formally, we define a dependency graph as follows:

Definition 3.3. A *dependency graph* $G = \langle M, T, E_M, E_T\rangle$, where M is a finite set of metaprogram nodes, T is a set of target nodes, E_M is a set of directed edges from metaprogram nodes to target nodes, and E_T is a set of directed edges from target nodes to metaprogram nodes. Each edge in E_M and in E_T is either an *explicit* edge or an *implicit* edge.

An acyclic dependency graph is said to be *well-formed*. Executing a metaprogram from a dependency graph which is not well-formed eventually becomes impossible without violating at least one dependency constraint; as a result, the construction of such dependency graphs results in a compile error. Hereafter, all discussion of dependency graphs will assume that they are well-formed.

To facilitate this discussion, we turn to the contrived example shown in Figure 13. This source file contains seven metaprograms, all of which perform no operations when executed (as they contain no statements). The `#target` and `#depends` clauses declare execution order. This order is enforced by constructing a dependency graph G as shown in Figure 14. Metaprograms themselves are shown as square nodes in the graph; metaprogram targets are shown as circles.

This graph defines a number of facts about execution order. We can see, for instance, that $\mathcal{M}_6$ should execute after $\mathcal{M}_3$ (because a path exists from $\mathcal{M}_6$ to $\mathcal{M}_3$). We formalize metaprogram dependencies with the following definition.

Definition 3.4. Let $\mathcal{M}$ and $\mathcal{M}'$ be metaprograms in a dependency graph G. $\mathcal{M}$ *depends* on $\mathcal{M}'$ (equiv., $\mathcal{M} \rightsquigarrow \mathcal{M}'$) if a path exists in the dependency graph from $\mathcal{M}$ to $\mathcal{M}'$. Otherwise, $\mathcal{M}$ does not depend on $\mathcal{M}'$ (equiv., $\mathcal{M} \not\rightsquigarrow \mathcal{M}'$). We write $\mathcal{M} \overset{=}{\rightsquigarrow} \mathcal{M}'$ to represent $\mathcal{M} \rightsquigarrow \mathcal{M}' \vee \mathcal{M} = \mathcal{M}'$ and we write $\mathcal{M} \overset{=}{\not\rightsquigarrow} \mathcal{M}'$ to represent $\neg(\mathcal{M} \overset{=}{\rightsquigarrow} \mathcal{M}')$.

```
[: #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

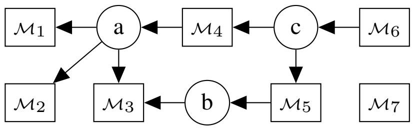

$$\begin{tikzcd}
\boxed{\mathcal{M}_1} & \text{a} \arrow[l] \arrow[dl] \arrow[d] & \boxed{\mathcal{M}_4} \arrow[l] & \text{c} \arrow[l] \arrow[d] & \boxed{\mathcal{M}_6} \arrow[l] \\
\boxed{\mathcal{M}_2} & \boxed{\mathcal{M}_3} & \text{b} \arrow[l] & \boxed{\mathcal{M}_5} \arrow[l] & \boxed{\mathcal{M}_7}
\end{tikzcd}$$

Figure 14. Example Dependency Graph

$\mathcal{M}$ is *ordered* with respect to $\mathcal{M}'$ (equiv., $\mathcal{M}$ and $\mathcal{M}'$ are *ordered*, $\mathcal{M} \leftrightsquigarrow \mathcal{M}'$) if $\mathcal{M} \rightsquigarrow \mathcal{M}'$ or if $\mathcal{M}' \rightsquigarrow \mathcal{M}$. Any two metaprograms which are not ordered are *unordered* (equiv., $\mathcal{M} \not\leftrightsquigarrow \mathcal{M}'$). We write $\mathcal{M} \underline{\leftrightsquigarrow} \mathcal{M}'$ to represent $\mathcal{M} \leftrightsquigarrow \mathcal{M}' \vee \mathcal{M} = \mathcal{M}'$ and we write $\mathcal{M} \not\underline{\leftrightsquigarrow} \mathcal{M}'$ to represent $\neg(\mathcal{M} \underline{\leftrightsquigarrow} \mathcal{M}')$.

We also reason about dependencies over targets. Given any target `t` in a dependency graph, $\mathcal{M}$ *depends* on target `t` if a path exists in the dependency graph from $\mathcal{M}$ to `t`. $\mathcal{M}$ is a *member* of target `t` (equiv., $\mathcal{M} \in$ `t`) if a path exists in the dependency graph from `t` to $\mathcal{M}$.

Based on this definition, it should be evident that any respectful ordering of a set of metaprograms is a topological sort of its dependency graph.

Any metaprogram which does not declare any dependencies implicitly depends on the target $\boldsymbol{\alpha}$, the only member of which is the origin metaprogram $\mathcal{M}_\alpha$. We use $\mathcal{M}_\alpha$ to represent the start of compilation. Intuitively, $\mathcal{M}_\alpha$ can be seen as the metaprogram which loads the initial sources for compilation. The edges leading into and out of $\boldsymbol{\alpha}$ are all implicit, the meaning of which is discussed below in Section 3.5.

The dependency graph is monotonically increasing during compilation. This is because metaprograms can create and insert other metaprograms into the AST (a process we term *metaprogram injection*); each new metaprogram appears in the dependency graph after its injector terminates. Metaprogram injection is the motivation for implicit edges and is discussed in Section 3.5. As indicated above, a compile error occurs if such an injection introduces a dependency graph cycle.

The definition of the dependency graph permits a metaprogrammer to express an explicit ordering between any two metaprograms. The bipartite nature of the graph and the existence of targets allows metaprograms to be grouped as a form of indirection. While the dependency graph permits the programmer to express ordering constraints between metaprograms, we must also be able to detect conflicts when ordering constraints are missing. The following sections demonstrate how this is accomplished.

$$\begin{array}{lll}
\eta & & \textit{node nonces} \\
l & & \textit{labels} \\
u & & \textit{native immutable values} \\
\mathcal{M} & & \textit{metaprogram identifiers} \\
\mathcal{K} ::= & l \mid \eta & \textit{record labels} \\
\mathcal{R} ::= & \{\overline{\mathcal{K} \mapsto \hat{v}}\} & \textit{records} \\
\mathcal{S} ::= & \{\overline{\mathcal{M}}\} & \textit{metaprogram sets} \\
\overset{\vartriangle}{\eta} ::= & \eta \mid \triangleright \mid \triangleleft & \textit{list nonces} \\
\mathring{\eta} ::= & (\overset{\vartriangle}{\eta}, \mathcal{M}, \mathcal{S}) & \textit{list element} \\
\mathcal{L} ::= & [\overline{\mathring{\eta}}] & \textit{lists} \\
v ::= & \eta \mid u \mid \mathcal{R} \mid \mathcal{L} & \textit{values} \\
\hat{v} ::= & v \mid \cancel{b} & \textit{optional value} \\
\mathring{\delta} ::= & \overset{R}{+}\eta(\overline{l = \hat{v}}) & \textit{record node creation} \\
 & \mid \overset{L}{+}\eta & \textit{list node creation} \\
 & \mid \eta.l \leftarrow \hat{v} & \textit{record assignment} \\
 & \mid \eta : \eta \hookleftarrow \overset{\vartriangle}{\eta} & \textit{list add-before} \\
 & \mid \eta : \overset{\vartriangle}{\eta} \hookrightarrow \eta & \textit{list add-after} \\
 & \mid \eta : \downarrow \eta & \textit{list removal} \\
\delta ::= & \mathcal{M} \succ \mathring{\delta} & \textit{edit script element} \\
e ::= & \hat{v} \mid \delta\ e & \textit{expression}
\end{array}$$

Figure 15. Edit Script Algebra Syntax

3.2 Edit Script Algebra

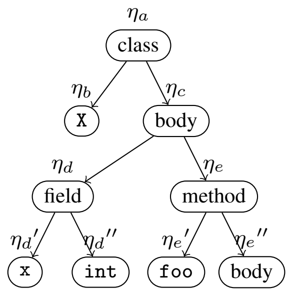
As stated above, a metaprogram's logic can be separated into analysis and modification. The objective of an edit script ($\bar{\delta} : \Delta_{\mathbb{P}}$) is to distill the modification logic of a given metaprogram over a known input program separated from its analysis logic. This edit script is then applied to the original program along with the edit scripts of other metaprograms to produce the final output.

In difference-based metaprogramming, we model metaprograms as having type $\mathbb{P} \to \Delta_{\mathbb{P}}$. But BSJ, a difference-based metaprogramming environment, has syntax that allows programmers to directly modify the AST; this behavior corresponds to the type $\mathbb{P} \to \mathbb{P}$ in our algebraic notation. The BSJ compiler reference implementation resolves this issue by executing metaprograms over heavily instrumented ASTs. These ASTs capture the behavior of the BSJ metaprogram by recording each mutation event; a sequence of such events composes the programmatic representation of an edit script. In this fashion, the BSJ reference implementation uses the algebra described here to drive its compilation process.

Figure 15 describes the syntax of our edit script algebra. This algebra is language-agnostic, but we relate it to BSJ throughout this section to provide practical examples and appeal to intuition. The grammar uses the notation $\overline{k \mapsto v}$

```
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

Figure 16. Example Record Encoding

to denote a sequence $k_1 \mapsto v_1, \ldots, k_n \mapsto v_n$. We use this notation to reflect the embedding of sequences within other sequences; for instance, $\overline{k \mapsto v}, k' \mapsto v'$ denotes the sequence $k_1 \mapsto v_1, \ldots, k_n \mapsto v_n, k' \mapsto v'$.

Node nonces, labels, native immutable values, and metaprogram identifiers may have any coherent form. In the case of BSJ, for instance, u represents values of types such as `String`, `int`, or the API-specific `BsjSourceLocation`. In this algebra, values include native immutables, node nonces, records, and lists. We treat records as maps, using $\mathcal{K} \mapsto \hat{v} \in \mathcal{R}$ to denote that a mapping exists in a map and using $\mathcal{R}[\![\mathcal{K} \mapsto \hat{v}]\!]$ to denote map substitution.

Our algebra seeks to represent object programs as structured data; the most common way to do so is in the form of an abstract syntax tree. We choose a somewhat unusual representation of this syntax tree: each node is mapped via a nonce to its structural representation. This can be viewed as an AST in which the children of each node are boxed references. In our algebra, every edit script only affects one AST node at a time; the tree-structured relationship between the nodes is semantically irrelevant. As a result, this record encoding considerably simplifies our semantics. For clarity, an example visualization of a simplified BSJ AST in this record encoding appears in Figure 16.

Our representations of AST nodes in the algebra are divided into two categories: record nodes and list nodes. Record nodes are constructs which are described by a fixed number of mappings from property labels to values. For example, nonces η_a, η_b, and η_d in Figure 16 all refer to record nodes; η_a, for instance, has the properties `id` and `body`.

By contrast, list nodes are AST nodes whose sole purpose is to contain a variable number of child nodes in a specific order. The nonces η_c and $\eta_e{}''$ in Figure 16 refer to list nodes. List nodes are distinct entities in order to prevent false positives in the conflict detection system which would be invoked by the simplistic representation of lists as a doubly-linked series of record nodes. This is discussed in detail when we present the algebra's semantics in Section 3.3.

Our algebra describes each list element as a three-tuple. The first element in this tuple is a "list nonce", which is either a node nonce (indicating the position of a node in the list) or a marker representing the beginning ($\triangleright$) or end ($\triangleleft$) of the list. The tuple also contains the ID of the metaprogram that inserted the node and a set of metaprograms which have deleted the element. The action of removing a node from a list simply marks its element as deleted for reasons that, as above, will become clear in Section 3.3.

In light of the above structure, we define the formal representation of a program as follows:

Definition 3.5. A *program* $\rho = \{\overline{\eta \mapsto \hat{v}}\}$ where, for all mappings $\eta \mapsto \hat{v}$, either $\hat{v} = \{\overline{l \mapsto \hat{v}'}\}$ (for record nodes) or $\hat{v} = [\overline{\mathring{\eta}}]$ (for list nodes).

As an example, the AST in Figure 16b is encoded as the record shown in Figure 16c.

Every edit script in the algebra consists of at most six types of edit script elements. Each edit script element is annotated with the ID of the metaprogram which produced it, although we often elide this prefix in cases where it is unimportant. That is, if it is irrelevant that $\mathcal{M} \succ \mathring{\delta}$ was created by metaprogram $\mathcal{M}$, we simply write $\mathring{\delta}$ instead.

The record and list node creation elements have the effect of mapping new nodes into a program. The record node creation element accepts as input a series of initial mappings. The list node creation element does not require such mappings; newly created list nodes are always initialized to contain two elements. The first element of a new list node contains the beginning marker; the second contains the ending marker. These markers are always present in a list to guarantee further list operations some points of reference.

The record assignment element changes a mapping in a specific record node. The newly-mapped value is always an optional value; as a result, it may be represented by $\not{v}$.

Two types of list addition element exist: add-before and add-after. All additions to a list are specified relative to an element which already exists in the list. This is the primary motivation for the beginning and end markers included in every list: such markers guarantee that list addition has a reference point even in an empty list. Our choice of notation is intended to reflect the operation being performed. The arrow in a list addition routine always points towards the newly inserted element and away from the reference point. For instance, $\eta : \eta' \hookleftarrow \overset{\vartriangle}{\eta}''$ indicates that η' was inserted into the list η immediately before the element containing $\overset{\vartriangle}{\eta}''$ (which may be a beginning or end marker). On the other hand, $\eta : \overset{\vartriangle}{\eta}'' \hookrightarrow \eta'$ indicates that η' was inserted after the element containing $\overset{\vartriangle}{\eta}''$. The removal element's notation mirrors that of the addition elements; while the addition elements show the arrow originating from the bottom of the line, the list removal element's arrow points in that direction.

```java
public class EditScriptExample {
  private int foo() {}
  [:
    BsjNodeFactory factory = context.getFactory();
    for (MethodDeclarationNode m : context.getPeers().
        filterByType(MethodDeclarationNode.class)) {
      MethodDeclarationNode m2 = m.deepCopy(factory);
      m2.setIdentifier(factory.makeIdentifierNode(
        m2.getIdentifier().getIdentifier()+"A"));
      context.addAfter(m2);
    }
  :]
  [:
    BsjNodeFactory factory = context.getFactory();
    for (MethodDeclarationNode m : context.getPeers().
        filterByType(MethodDeclarationNode.class)) {
      MethodDeclarationNode m2 = m.deepCopy(factory);
      m2.setIdentifier(factory.makeIdentifierNode(
        m2.getIdentifier().getIdentifier()+"B"));
      context.addAfter(m2);
    }
  :]
}
```

Figure 17. Edit Script Example Code

$$\bar{\delta}_{\mathcal{M}_1} \begin{cases} \overset{R}{+}\eta_a(\text{identifier} = \texttt{fooA}) & \text{create identifier} \\ \overset{L}{+}\eta'_a & \text{create empty body} \\ \overset{R}{+}\eta''_a(\text{id} = \eta_a, \text{body} = \eta'_a) & \text{create } \texttt{fooA} \text{ method} \\ \eta_c : \eta''_a \hookleftarrow \triangleleft & \text{add } \texttt{fooA} \text{ to class} \end{cases}$$

$$\bar{\delta}_{\mathcal{M}_2} \begin{cases} \overset{R}{+}\eta_b(\text{identifier} = \texttt{fooB}) & \text{create identifier} \\ \overset{L}{+}\eta'_b & \text{create empty body} \\ \overset{R}{+}\eta''_b(\text{id} = \eta_b, \text{body} = \eta'_b) & \text{create } \texttt{fooB} \text{ method} \\ \eta_c : \eta''_b \hookleftarrow \triangleleft & \text{add } \texttt{fooB} \text{ to class} \end{cases}$$

Figure 18. Example Edit Scripts

As an example of the edit script algebra in use, we provide the simple program in Figure 17. Both metaprograms in the figure perform similar operations. The first creates a copy of every method in the class, suffixing `A` onto the method name. The second metaprogram suffixes `B` instead.

In a traditional metaprogramming environment, we would always see methods `foo`, `fooA`, and `fooB`. However, we would also see either `fooBA` or `fooAB` depending on the metaprograms' execution order. This is an example of how a traditional metaprogramming model produces a read-write conflict: the metaprogram which is executed last is basing its behavior on the metaprogram which is executed first even though the two metaprograms are unordered.

In a difference-based metaprogramming environment, however, each metaprogram is executed independently over a copy of the program to calculate its edit script. Consider the case in which each BSJ metaprogram in Figure 17 is executed over its original AST. The first metaprogram produces method `fooA` and the second metaprogram produces method `fooB`. Because neither metaprogram can see the method produced by the other, neither `fooBA` nor `fooAB` would be created. These facts are represented algebraically by the edit scripts in Figure 18: $\bar{\delta}_{\mathcal{M}_1}$ represents the behavior of the first metaprogram and $\bar{\delta}_{\mathcal{M}_2}$ represents the behavior of the second.

If we had desired the `fooAB` method in Figure 17, we could simply make the second metaprogram in our example depend upon the first. When the second metaprogram executed, it would be able to see the changes applied by the first metaprogram and react accordingly. This is not a conflict because a dependency exists between the two metaprograms; as a result, every respectful ordering will ensure that the first metaprogram executes before the second.

Gathering edit scripts instead of executing the metaprogram directly over a changed tree trivially obviates read-write conflicts: because a metaprogram's input only includes the scripts from its dependencies, it cannot see the effects of unordered metaprograms and is therefore unable to react to them. But the mere gathering of edit scripts does not prevent dual-write conflicts. Dual-write conflicts are captured in the form of merge failures as described in the next section.

3.3 Edit Script Semantics

This section defines the semantics of the edit script algebra. We relate operations that BSJ metaprograms can perform on a node to edit script elements (δ). We also describe the evaluation semantics $\Rightarrow$ (as in $\delta(\rho) \Rightarrow \rho'$) for applying an edit script element to a program.

When a node is created in a BSJ metaprogram, the node factory generates an appropriate edit script element representing node creation. Any method invocations which alter that node (such as a property setter or a list addition) generate edit script elements to capture the modification logic of those actions. Because every child of a BSJ node is either itself a node or is an immutable value, all changes to the tree will occur through the AST API; thus, every change is recorded in the form of an edit script element.

List nodes in the BSJ API may be changed either through the addition or removal of their child elements. Removal is specified in terms of the child to remove. Addition is specified in terms of a reference child (which is already in the list), the new child to add, and a direction (before or after). The `java.util.List` interface is implemented in the API over list nodes, but the implementation translates such calls to a series of these primitives; for instance, an invocation of `List.add` generates an edit script element of the form $\eta : \eta' \hookleftarrow \triangleleft$. We do this to map the relatively full-featured Java `List` interface onto our algebra. In our experience, metaprograms operating over an AST's list are concerned with relative operations (such as "insert print statement before variable assignment") rather than with absolute operations (such as "insert print statement at index 5").

Application of edit script elements to programs is defined in terms of the big-step semantics in Figure 19. The evaluation relation ($\Rightarrow$) relates algebraic expressions to their values. Expressions include edit script application; values include object programs. The ϑ function is defined in terms of this algebra:

Definition 3.6. $\vartheta(\delta, \rho) = \rho'$ iff $\delta\ \rho \Rightarrow \rho'$.

These semantics make use of a list search function Σ defined in the discussion below.

Record Node Creation Rule

$$\frac{\eta \mapsto \hat{v} \notin \rho \qquad \rho\llbracket \eta \mapsto \{\overline{l \mapsto \hat{v}}\} \rrbracket \Rightarrow \rho'}{(\overset{R}{+}\eta(\overline{l = \hat{v}}))\ \rho \Rightarrow \rho'}$$

List Node Creation Rule

$$\frac{\eta \mapsto \hat{v} \notin \rho \qquad \rho\llbracket \eta \mapsto [(\triangleright, \mathcal{M}, \emptyset), (\triangleleft, \mathcal{M}, \emptyset)] \rrbracket \Rightarrow \rho'}{(\mathcal{M} \succ \overset{L}{+}\eta)\ \rho \Rightarrow \rho'}$$

Record Assignment Rule

$$\frac{\eta \mapsto \mathcal{R} \in \rho \qquad \rho\llbracket \eta \mapsto \mathcal{R}\llbracket l \mapsto \hat{v} \rrbracket \rrbracket \Rightarrow \rho'}{(\eta.l \leftarrow \hat{v})\ \rho \Rightarrow \rho'}$$

List Add Before Rule

$$\frac{\begin{array}{c} \overset{\triangle}{\eta}_3 \neq \triangleright \\ \eta_1 \mapsto \mathcal{L} \in \rho \qquad \mathring{\eta}_3 = \Sigma(\overset{\triangle}{\eta}_3, \mathcal{M}, \mathcal{L}) \qquad \mathcal{L} = [\overline{\mathring{\eta}'}, \mathring{\eta}_3, \overline{\mathring{\eta}''}] \\ \mathcal{L}' = [\overline{\mathring{\eta}'}, (\eta_2, \mathcal{M}, \emptyset), \mathring{\eta}_3, \overline{\mathring{\eta}''}] \qquad \rho\llbracket \eta_1 \mapsto \mathcal{L}' \rrbracket \Rightarrow \rho' \end{array}}{(\mathcal{M} \succ \eta_1 : \eta_2 \looparrowleft \overset{\triangle}{\eta}_3)\ \rho \Rightarrow \rho'}$$

List Add After Rule

$$\frac{\begin{array}{c} \overset{\triangle}{\eta}_3 \neq \triangleleft \\ \eta_1 \mapsto \mathcal{L} \in \rho \qquad \mathring{\eta}_3 = \Sigma(\overset{\triangle}{\eta}_3, \mathcal{M}, \mathcal{L}) \qquad \mathcal{L} = [\overline{\mathring{\eta}'}, \mathring{\eta}_3, \overline{\mathring{\eta}''}] \\ \mathcal{L}' = [\overline{\mathring{\eta}'}, \mathring{\eta}_3, (\eta_2, \mathcal{M}, \emptyset), \overline{\mathring{\eta}''}] \qquad \rho\llbracket \eta_1 \mapsto \mathcal{L}' \rrbracket \Rightarrow \rho' \end{array}}{(\mathcal{M} \succ \eta_1 : \overset{\triangle}{\eta}_3 \looparrowright \eta_2)\ \rho \Rightarrow \rho'}$$

List Remove Rule

$$\frac{\begin{array}{c} \eta_1 \mapsto \mathcal{L} \in \rho \\ \mathring{\eta}_2 = (\eta_2, \mathcal{M}', \mathcal{S}) = \Sigma(\eta_2, \mathcal{M}, \mathcal{L}) \qquad \mathcal{L} = [\overline{\mathring{\eta}'}, \mathring{\eta}_2, \overline{\mathring{\eta}''}] \\ \mathcal{L}' = [\overline{\mathring{\eta}'}, (\eta_2, \mathcal{M}', \mathcal{S} \cup \{\mathcal{M}\}), \overline{\mathring{\eta}''}] \qquad \rho\llbracket \eta_1 \mapsto \mathcal{L}' \rrbracket \Rightarrow \rho' \end{array}}{(\mathcal{M} \succ \eta_1 : \downarrow \eta_2)\ \rho \Rightarrow \rho'}$$

Recursive Application Rule

$$\frac{\delta'\ e \Rightarrow \rho \qquad \delta\ \rho \Rightarrow \rho'}{\delta\ (\delta'\ e) \Rightarrow \rho'}$$

Value Rule

$$\frac{}{\hat{v} \Rightarrow \hat{v}}$$

Figure 19. Edit Script Semantics

Certain properties of this algebra are worth observing. First, recall that programs ρ are merely mappings $\mathcal{R}$ from nonces to node structures. As a result, each nonce added by the creation rules must be unique. This property is enforced by the rules themselves as they will not overwrite a mapping for an existing node; thus, any sequence of edit script elements applied to a program which contains two creations of the same nonce will not evaluate.

Record assignment is simplistic compared to the other rules in this algebra. We locate the record corresponding to the nonce for which the assignment is taking place. In that record, we substitute the label's current mapping for one which reflects the new value. We then substitute the resulting record into the program to obtain our new program.

The list addition rules appear somewhat more complex. Addition to a list always occurs relative to another element which is already in the list. This is the motivation for the begin ($\triangleright$) and end ($\triangleleft$) elements: because they are always present in the list, they provide available reference points even if the list is empty. Addition before the begin element or after the end element are, of course, prohibited. Each list cell is a triple composed of the list nonce it represents, the metaprogram responsible for inserting it, and a set of metaprograms which have marked it as deleted.

Because every list operation uses a list nonce as a reference point, that list nonce must uniquely identify a cell in the list for the metaprogram applying the operation. To do so, we define the search function Σ:

Definition 3.7. $\Sigma(\overset{\triangle}{\eta}, \mathcal{M}, \mathcal{L}) = \mathring{\eta}$ iff $\{(\overset{\triangle}{\eta}', \mathcal{M}', \mathcal{S}) : \mathcal{L} \mid \mathcal{M} \underline{\rightsquigarrow} \mathcal{M}' \wedge \forall \mathcal{M}'' \in \mathcal{S}. \mathcal{M} \not\underline{\rightsquigarrow} \mathcal{M}''\} = \{\mathring{\eta}\}$.

The intuition of the Σ function is to provide a view of the list consistent with the dependencies of metaprogram $\mathcal{M}$. Any elements added by metaprograms that $\mathcal{M}$ can see will not be considered by the search. Any elements added by a metaprogram that $\mathcal{M}$ *can* see but removed by a metaprogram that $\mathcal{M}$ *cannot* see will still be considered. This is necessary to ensure that the metaprogram cannot observe any list operations performed by other metaprograms which are not its dependencies; otherwise, a read-write conflict would be possible.

For example, suppose that metaprogram $\mathcal{M}_1$ adds an element η_i to a list; that $\mathcal{M}_2 \rightsquigarrow \mathcal{M}_1$ and $\mathcal{M}_2$ removes the element from the list; and that $\mathcal{M}_3 \rightsquigarrow \mathcal{M}_1$ and $\mathcal{M}_4 \rightsquigarrow \mathcal{M}_2$. In this case, the list would contain a cell $(\eta_i, \mathcal{M}_1, \{\mathcal{M}_2\})$. Because $\mathcal{M}_4$ depends on $\mathcal{M}_2$, it *should not* see the element in the list. Because $\mathcal{M}_3$ depends on $\mathcal{M}_1$ but not on $\mathcal{M}_2$, it *should* see the element in the list. We must be able to satisfy both of these conditions at the same time. This is the purpose of the Σ function: because the only deleting metaprogram of the cell is $\mathcal{M}_2$ and because $\mathcal{M}_3 \not\rightsquigarrow \mathcal{M}_2$, the cell is not considered deleted for $\mathcal{M}_3$. Similarly, we use the inserting metaprogram's identifier to guarantee that some other metaprogram $\mathcal{M}_5$ which is not ordered with these other metaprograms cannot see the element in the list (because $\mathcal{M}_5 \not\rightsquigarrow \mathcal{M}_1$).

In light of the above, both list addition operations are fairly simple. We locate the list structure in the program and we locate the appropriate cell in the list. We then define a new list with the new cell in the appropriate location and substitute it back into the original program. The list removal rule works in much the same way, but it does not actually remove the cell from the list; it is the pervue of each metaprogram to decide (by use of the Σ function) whether or not each removal applies, so the removal rule simply updates the deletion set for that cell.

3.4 Merging Edit Scripts

While read-write conflicts are obviated by the extraction of edit scripts as described in Section 3.2, dual-write conflicts are still possible. The conflict shown in Figure 5, for instance, is a type of dual-write conflict. The relevant portions of the edit scripts are shown in Figure 20. In these scriptlets, η_c is taken to be the nonce of the method body and η_a and η_b are taken to be the nonces of the code literal expressions.

$\cdots$	creation of code literal expression
$\mathcal{M} \succ \overset{R}{+}\eta_a(\cdots)$	create `print` method invocation
$\mathcal{M} \succ \eta_c : \triangleright \looparrowright \eta_a$	add method invocation to body
$\cdots$	creation of code literal expression
$\mathcal{M}' \succ \overset{R}{+}\eta_b(\cdots)$	create `print` method invocation
$\mathcal{M}' \succ \eta_c : \triangleright \looparrowright \eta_b$	add method invocation to body

Figure 20. Simple Conflict Edit Scriptlets

In light of the semantics of edit scripts described in Figure 19, it is quite clear that these metaprograms are in conflict: their execution order will affect the ordering of η_a and η_b in the final program. But the very similar edit scripts presented in Figure 18 are not in conflict. The order in which they are executed will affect the ordering of the methods in the resulting program's class declaration but, unlike statements in a method, that ordering is not significant in Java.

Complicating matters is the fact that this ordering significance is not a property of the list itself; it is a property of the node in the list. In Java, for instance, the ordering of elements in the body of a class declaration is largely insignificant: a method can appear before or after another method without affecting the semantics of the program. But the ordering of some of those elements may matter: the ordering of initializers *is* significant.

Our algebra accommodates this node-based significance in ordering by introducing the order-dependence function ω. This function is simply an abstract fixed predicate over node nonces. When the algebra is applied to a specific language, the ω predicate should match those nodes whose order in a list is significant.

Definition 3.8. A node represented by the nonce η is *order-dependent* iff $\omega(\eta)$. For convenience, we say that the nonce is order-depdendent if the node is order-dependent. The order-dependence of a node is constant.

Our definition of order-dependence now allows us to specify the equivalence relation between programs:

Definition 3.9. Two programs $\rho = \{\overline{\eta \mapsto \hat{v}}\}$ and $\rho' = \{\overline{\eta \mapsto \hat{v}'}\}$ are *equivalent* (equiv., $\rho \cong \rho'$) iff all of the following are true:

- $\eta \mapsto \mathcal{R} \in \rho \Leftrightarrow \eta \mapsto \mathcal{R} \in \rho'$
- $\eta \mapsto \mathcal{L} \in \rho \Leftrightarrow \eta \mapsto \mathcal{L}' \in \rho'$. Further, $E = \{\mathring{\eta}' : \mathcal{L} \mid \mathring{\eta}_j = (\eta', \mathcal{M}, \emptyset)\} = \{\mathring{\eta}' : \mathcal{L}' \mid \mathring{\eta}_j = (\eta', \mathcal{M}, \emptyset)\}$. Finally, for all $(\mathring{\eta}', \mathring{\eta}'') \in E \times E$, if $\omega(\eta') \wedge \omega(\eta'')$, then $\mathring{\eta}'$ appears before $\mathring{\eta}''$ in $\mathcal{L}$ iff it also does in $\mathcal{L}'$.

For any two edit scripts $\bar{\delta}$ and $\bar{\delta}'$, the expression $\bar{\delta} \cong \bar{\delta}'$ indicates that $\forall \rho \in \mathbb{P}.(\vartheta(\bar{\delta}, \rho)) \cong (\vartheta(\bar{\delta}', \rho))$.

For convenience, we use $\bar{\delta}\ (\bar{\delta}'\ \rho) \cong \bar{\delta}'\ (\bar{\delta}\ \rho)$ to denote that $\vartheta(\bar{\delta}, \vartheta(\bar{\delta}', \rho)) \cong \vartheta(\bar{\delta}', \vartheta(\bar{\delta}, \rho))$.

Intuitively, the above definition states that two programs are equivalent as long as they are identical except for the relative ordering of order-independent elements in their lists.

Record Assignment Conflict Rule

$$\frac{\hat{v} \neq \hat{v}'}{\eta.l \leftarrow \hat{v} \not\leftrightarrow \eta.l \leftarrow \hat{v}'}$$

Add Before Conflict Rule

$$\frac{\omega(\eta_2) \qquad \omega(\eta_2')}{\eta_1 : \eta_2 \looparrowleft \overset{\triangle}{\eta}_3 \not\leftrightarrow \eta_1 : \eta_2' \looparrowleft \overset{\triangle}{\eta}_3}$$

Add After Conflict Rule

$$\frac{\omega(\eta_2) \qquad \omega(\eta_2')}{\eta_1 : \overset{\triangle}{\eta}_3 \looparrowright \eta_2 \not\leftrightarrow \eta_1 : \overset{\triangle}{\eta}_3 \looparrowright \eta_2'}$$

Unordered Creation Conflict Rule

$$\frac{\delta = \overset{R}{+}\eta(\overline{l \mapsto \hat{v}}) \vee \delta = \overset{L}{+}\eta \qquad \eta \in \delta'}{\delta \not\leftrightarrow \delta'}$$

Figure 21. Conflict Semantics

Using the above definitions, we can now combine the edit scripts produced by each metaprogram. Dual-write conflicts are detected by defining the *element conflict* relation ($\not\leftrightarrow$); we define this relation as the least symmetric relation following the rules shown in Figure 21. These rules use the notation $\eta \in \delta$, which indicates that η appears in the element δ. For instance, $\eta \in (\eta : \downarrow \eta')$ but $\eta'' \notin (\eta : \downarrow \eta')$.

We then wish to use this element conflict relation to determine whether or not two edit scripts can be composed without ambiguity. First, we define a predicate which describes whether or not two edit scripts are *mergable*.

Definition 3.10. Two edit scripts $\bar{\delta}$ and $\bar{\delta}'$ are *mergable* (equiv., $\bar{\delta} \overset{?}{\rightarrowtail} \bar{\delta}'$) iff $\forall(\mathcal{M} \succ \delta, \mathcal{M}' \succ \delta') \in \bar{\delta} \times \bar{\delta}'.\mathcal{M}' \underline{\rightsquigarrow} \mathcal{M} \vee \neg(\delta \not\leftrightarrow \delta')$.

Intuitively, two edit scripts are mergable if (1) the order in which they should be concatenated is fixed because of the dependency graph or (2) the composition of the two edit scripts commutes up to equivalence.

As described above, the mergable relation $\overset{?}{\rightarrowtail}$ makes use of the element conflict relation $\not\leftrightarrow$ defined in Figure 21. This latter relation holds in the cases in which dual-write conflicts can occur. Node creation between unordered metaprograms, for example, only causes a conflict when they attempt to create nodes using the same nonce. In practice, this should never occur; the BSJ compiler guarantees that every AST node in a compilation is given a unique ID.

List insertion between unordered metaprograms only conflicts when the lists and reference points are the same and both elements are ordered. This ordering exception is made to ensure that two edit script elements are considered non-conflicting as long as the programs they produce are equivalent (even if their ASTs are distinct). Making this exception for lists considerably improves the programming experience of languages such as BSJ, as it allows metaprogrammers to focus on the semantics (rather than the syntax) of the object program.

Removal of a node from a list never causes a conflict because it can always be merged. If two metaprograms remove the same element from a list, the merge of those operations should simply remove that element from the list.

We then define the *merge* operator $\rightarrowtail$ which successfully combines two edit scripts if and only if they are mergable.

Definition 3.11. The *merge* between two edit scripts $\bar{\delta}$ and $\bar{\delta}'$ (equiv. $\bar{\delta} \rightarrowtail \bar{\delta}'$) is defined as:
$\bar{\delta} \rightarrowtail \bar{\delta}' = \delta_1, \ldots, \delta_n, \delta'_1, \ldots, \delta'_m$ iff $\bar{\delta} \overset{?}{\rightarrowtail} \bar{\delta}'$
If $\neg(\bar{\delta} \overset{?}{\rightarrowtail} \bar{\delta}')$, then this merge is said to *fail*.

Our key objective, as stated at the beginning of Section 3, is that to ensure that a compilation only produces an output program when every respectful ordering of metaprograms is equivalent. We state this objective as Theorem 1:

Theorem 1. *Given metaprograms $\mathcal{M}_\alpha, \mathcal{M}_1, \ldots, \mathcal{M}_n$ in a dependency graph G, let $\bar{\delta}_\alpha, \bar{\delta}_1, \ldots, \bar{\delta}_n$ be the edit scripts produced by those metaprograms. If any respectful ordering $\bar{k}$ of these metaprograms successfully merge $\bar{\delta}_\alpha \rightarrowtail \bar{\delta}_{k_1} \rightarrowtail \cdots \rightarrowtail \bar{\delta}_{k_n}$, then every respectful ordering $\bar{k}'$ of these metaprograms produces an equivalent object program.*

A proof of this theorem appears in Appendix A.

Once all metaprograms have been executed and their edit scripts have been produced, the compiler chooses a respectful ordering of metaprogram edit scripts $\bar{k}$ and computes $\bar{\delta}^* = \bar{\delta}_{k_1} \rightarrowtail \cdots \rightarrowtail \bar{\delta}_{k_n}$. By Theorem 1 above, every such respectful ordering produces an equivalent program. If the merge is successful, the resulting $\bar{\delta}^*$ is applied to the original AST and the result is serialized into Java source code for compilation. If this merge is unsuccessful, a compile-time error is generated and compilation halts.

We have thus accomplished our key objective. The BSJ compiler follows the algebra above and this algebra ensures that its behavior is always deterministic: if no element conflict is found during the merge of a given respectful ordering of edit scripts, then every respectful ordering produces the same program (up to equivalence). As a result, the programmer can take successful compilation to mean that the BSJ program has exactly one meaning.

We can relate the merge operator to software merging, which can be categorized in several different ways [12]: the program representation (textual, syntactic, semantic, or structural), available information about the transformation process (state-based, changed-based, or operation-based), and conflict detection strategy (ad-hoc, conflict tables, conflict sets, etc.). The algebra presented here represents programs syntactically and the merge operator describes an operation-based merge over edit script elements. The element conflict relation corresponds to a simple conflict table.

3.5 Injection Conflicts

Metaprogram injection, discussed in Section 2.5, gives rise to a particularly eldritch problem known as an *injection conflict*. An injection conflict occurs when a metaprogram injected into the AST could result in violating a previously-satisfied dependency constraint. Like other conflicts, the presence of an injection conflict in the dependency graph causes a compilation error.

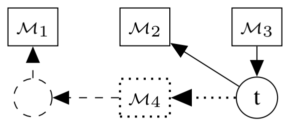


Figure 22. Example Injection Conflict

Figure 22 contains a dependency graph with an injection conflict. At the start of compilation, three metaprograms ($\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$) are known to exist; $\mathcal{M}_4$ does not yet exist and none of the dashed or dotted lines are present. This graph demonstrates that $\mathcal{M}_3$ depends on target `t` and therefore must not execute until after $\mathcal{M}_2$ executes. There are three execution orders that observe that constraint: $\{\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3\}$, $\{\mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_1\}$, and $\{\mathcal{M}_2, \mathcal{M}_1, \mathcal{M}_3\}$.

Presume that $\mathcal{M}_1$, when executed, adds $\mathcal{M}_4$ to the dependency graph. Also presume that $\mathcal{M}_4$ declares that it is a member of target `t`. This information is represented by the dotted portions of Figure 22. In this case, $\mathcal{M}_3$ should not run until after $\mathcal{M}_4$ runs. Since $\mathcal{M}_4$ cannot run until it is created by $\mathcal{M}_1$, we can see that $\mathcal{M}_3$ should not run until $\mathcal{M}_1$ runs. This invalidates the execution order $\{\mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_1\}$, but we cannot know this until compilation is partially complete; if we had initially chosen that execution order, we will find ourselves inserting $\mathcal{M}_4$ after $\mathcal{M}_3$ has already executed! Furthermore, we cannot simply detect when such an ordering is chosen and produce a compile error; we must produce a compile error if *any* legal execution orders (including those we are not running) become invalid during the process of compilation. Otherwise, an arbitrary execution order decision by the compiler determines whether compilation succeeds or fails and we no longer have unambiguous compilation.

Recall from Definition 3.3 that all edges in the dependency graph are either explicit or implicit. Edges representing declared dependencies are always explicit. In the case of metaprogram injection, however, an implicit target is created which corresponds to the injecting metaprogram. The injected metaprogram then gains an implicit dependency upon that target and the injecting program implicitly becomes a member of it. This models the fact that the injected metaprogram (such as $\mathcal{M}_4$) cannot execute until after the injecting metaprogram (such as $\mathcal{M}_1$) is finished.

Next, we must define the condition which prevents an injection conflict between two nodes. This condition is termed an *injection dependency* and is defined below. Note that injection dependencies are only defined over a well-formed dependency graph.

Definition 3.12. Let the following path-like relations be defined between nodes in a dependency graph G.

- $\mathcal{M} \overset{\bullet}{\rightsquigarrow} \mathcal{M}'$: A path exists from $\mathcal{M}$ to $\mathcal{M}'$ in G which consists entirely of explicit edges.

- $\mathcal{M}\overset{\circ}{\twoheadrightarrow}\mathcal{M}'$: A path exists from $\mathcal{M}$ to $\mathcal{M}'$ in G which consists of an implicit edge from $\mathcal{M}$ to some target and an implicit edge from that target to $\mathcal{M}'$.
- $\mathcal{M}\overset{\bullet/\circ}{\rightsquigarrow}\mathcal{M}'$: An injection dependency exists between $\mathcal{M}$ and $\mathcal{M}'$ in G: for each metaprogram $\breve{\mathcal{M}}$ in G, let $S_{\breve{\mathcal{M}}} = \{\breve{\mathcal{M}}' \mid \breve{\mathcal{M}}\overset{\circ}{\twoheadrightarrow}\breve{\mathcal{M}}'\}$. Then, $\mathcal{M}\overset{\bullet/\circ}{\rightsquigarrow}\mathcal{M}'$ iff $(\exists\, \mathcal{M}\overset{\bullet}{\rightsquigarrow}\mathcal{M}') \vee (|S_{\mathcal{M}}| > 0 \wedge \forall \mathcal{M}'' \in S_{\mathcal{M}}, \mathcal{M}''\overset{\bullet/\circ}{\rightsquigarrow}\mathcal{M}')$.

The definition of $\mathcal{M}\overset{\bullet/\circ}{\rightsquigarrow}\mathcal{M}'$ identifies the conditions necessary to ensure that the ordering information present in the dependency graph was already present before injection. The first term of the conjunction indicates that no new information is introduced if $\mathcal{M}$ and $\mathcal{M}'$ were already ordered. The second term expresses a constraint meant to handle cases of recursive injection: no new ordering is introduced if we explicitly depend upon whatever our injector depends upon (or whatever those metaprograms depend upon, recursively). This recursion is clearly well-founded for a well-formed dependency graph as such graphs are acyclic.

We now define an injection conflict as follows:

Definition 3.13. An injection conflict exists from $\mathcal{M}$ over $\mathcal{M}'$ with the nodes in $S_{\mathcal{M}'}$ iff $|S_{\mathcal{M}'}| > 0$ and $\forall \mathcal{M}'' \in S_{\mathcal{M}'}, \neg\mathcal{M}\overset{\bullet/\circ}{\rightsquigarrow}\mathcal{M}''$

Intuitively, Definition 3.13 reads as follows: if any metaprogram $\mathcal{M}$ depends on another metaprogram $\mathcal{M}'$ and that metaprogram $\mathcal{M}'$ was generated (directly or indirectly) by some metaprograms in $S_{\mathcal{M}'}$, then $\mathcal{M}$ must not be a candidate for execution until $\mathcal{M}'$ exists. We ensure that $\mathcal{M}$ is not executed until $\mathcal{M}'$ exists by ensuring that $\mathcal{M}$ cannot run until after some $\mathcal{M}'' \in S_{\mathcal{M}'}$ runs.

For example, consider Figure 22. We can now see that an injection conflict exists from $\mathcal{M}_3$ over $\mathcal{M}_4$ with $\mathcal{M}_1$. We have that $S_{\mathcal{M}_4} = \{\mathcal{M}_1\}$ and so $|S_{\mathcal{M}_4}| > 0$; we also have that $\neg\mathcal{M}_3\overset{\bullet/\circ}{\rightsquigarrow}\mathcal{M}_1$ because $\neg\mathcal{M}_3\overset{\bullet}{\rightsquigarrow}\mathcal{M}_1$ and because $\mathcal{M}_3$ is not an injected metaprogram (meaning that $|S_{\mathcal{M}_3}| = 0$).

We have thus reached our goal of detecting injection conflicts independent of the chosen execution order. Also, the phrasing of the injection conflict conveys a solution which can be communicated to the programmer: if $\mathcal{M}$ has an injection conflict over $\mathcal{M}'$ with $S_{\mathcal{M}''}$, then the conflict can be resolved by explicitly declaring any dependency such that $\mathcal{M} \rightsquigarrow \mathcal{M}''$ for $\mathcal{M}'' \in S_{\mathcal{M}'}$.

While the BSJ reference implementation treats an injection conflict as an error, we observe that this is not strictly necessary. In cases in which we chose an execution order that triggered the conflict, our difference-based metaprogramming model allows us to simply recompute the edit scripts for any affected metaprograms and their dependents. Consider, for instance, the case in which we have the graph in Figure 22 and we chose the execution order $\{\mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_1\}$. Upon executing $\mathcal{M}_1$ and discovering the injection conflict, it would be possible to prepare another input AST for $\mathcal{M}_3$ (this time including the edit script from $\mathcal{M}_4$) and recompute its edit script, discarding the old one. In effect, we would then have the execution order $\{\mathcal{M}_2, \mathcal{M}_1, \mathcal{M}_4, \mathcal{M}_3\}$. This would not be possible in a traditional metaprogramming model, as it would not be possible in the general case to undo the changes a metaprogram made to the AST. The BSJ reference implementation does not include this feature – we believe that the metaprogrammer would wish to fix the inconsistency – but this solution helps to illustrate the nature of injection conflicts.

3.6 Restrictions

Our compiler reference implementation uses the edit script algebra to ensure unambiguous output (although a proof of correctness of the reference implementation is clearly beyond the scope of this paper). This property of unambiguity relies upon the compiler's ability to isolate metaprograms from each other. Because BSJ metaprograms are capable of escaping the JVM in several ways (I/O, JNI, etc.), perfect conflict identification would require significant information flow instrumentation from the JVM and underlying operating system. In order to make implementation tractable and prevent excessive restrictions, two sacrifices were made. These represent potential *false negatives* in the BSJ conflict detection system, so metaprogrammers should observe them with care:

- **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.

These two restrictions prove easy to follow in practice; accidental violation, for instance, does not appear to be a concern. Pending further experience, we accept these restrictions in exchange for the conflict detection behavior that they provide.

4. Code Literals

As discussed in Section 2.6, BSJ provides a quasiquoting syntax known as a code literal. Code literals permit a very readable and terse form of AST construction. Unfortunately, the meaning of some code literals can be difficult to ascertain. The literal `<:x:>`, for instance, could be interpreted as an identifier, a simple name, a singleton list of type references (as in an `implements` clause), or any of thirteen other meanings. A parser would normally distinguish based on its internal state but, since this code has been removed from its context, this is not possible.

Previous work has explored the problem of correctly inferring the type of the code literal. MetaAspectJ [9] uses "guess-parsing": parse rules are tried in order and the first match is used. Unfortunately, when multiple rules match, the values corresponding to those parses which were not chosen cannot be expressed. A more elegant approach exists in [3]. This approach involves *lifting* the code (translating object program code to a metaprogram AST which creates it) into a set of construction forms and then eliminates those forms which do not successfully typecheck. If exactly one form typechecks correctly, this form is used as the meaning of the object program code fragment.

BSJ uses a solution similar to that of [3] but defers the lifting of code until after typechecking is complete. Instead, the code literal is directly represented as a family of types in the BSJ type system. This has two effects. First, it provides opportunities for improving the performance of the disambiguation process: BSJ does not need to parse or lift code literal productions that it knows will not successfully typecheck. Second, it becomes clearer the impact that the code literals will have on the semantics of the metaprogram in which they are used.

We begin this section by introducing selection types, the family of types which represents code literals in the BSJ language. We then explain the evaluation process for code literals. Finally, we provide an analysis of the properties of the code literal model of quasiquoting.

4.1 Selection Types

BSJ introduces a new type family over those it inherits from Java: *selection types*. Selection types are similar to Java's intersection types in that they are used in a very constrained fashion and are impossible to represent explicitly using the language's syntax. BSJ programmers should generally not require a deep understanding of selection types.

A selection type is formed over a bag of types. We use the notation $\{A,B\}^{\&}$ to represent the selection type between A and B. This selection type contains all *selection bags* which contain a value of type A and a value of type B. We use the notation $\{x,y\}_{\&}$ to denote the selection bag of the values x and y; if $x : A$ and $y : B$, then $\{x,y\}_{\&} : \{A,B\}^{\&}$. Because bags are unordered, $\{A,B\}^{\&} = \{B,A\}^{\&}$ and $\{x,y\}_{\&} = \{y,x\}_{\&}$. Selection types are not reference types and are not related via subtyping.

The type rules for selection types, presented in Figure 23, are similar to those of record types in most languages. Unlike record types, selection types are unlabeled, making traditional projection impossible. Projection from a selection bag is performed implicitly: via the *selection conversion*. We write $A \twoheadrightarrow B$ to indicate that type A can be selection-converted to type B. The Selection Type Projection Rule allows a selection type to be converted to another type τ' if exactly one of its component types is a subtype of τ'. The selection conversion is legal in the conversion contexts of assignment, method invocation, and casting.[2]

SELECTION BAG TYPE RULE

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{e_1,\ldots,e_n\}_{\&} : \{\tau_1,\ldots,\tau_n\}^{\&}}$$

SELECTION TYPE REORDERING RULE

$$\frac{\begin{array}{c}\forall(k \mapsto v) \in m, (k' \mapsto v') \in m.(k = k' \Leftrightarrow v = v') \\ \tau_1 = \tau'_{m[1]} \qquad \cdots \qquad \tau_n = \tau'_{m[n]}\end{array}}{\{\tau_1,\ldots,\tau_n\}^{\&} \cong \{\tau'_1,\ldots,\tau'_n\}^{\&}}$$

EQUIVALENCE RULE

$$\frac{e' : \tau' \qquad e \cong e' \qquad \tau \cong \tau'}{e : \tau}$$

SELECTION TYPE PROJECTION RULE

$$\frac{\exists! k.\tau_k <: \tau'}{\Gamma \vdash \{\tau_1,\ldots,\tau_n\}^{\&} \twoheadrightarrow \tau'}$$

Figure 23. Selection Type Rules

The choice of notation for selection types was inspired by the additive conjunction from linear logic. An additive conjunction represents an alternative occurrence of resources, the choice of which belongs to the consumer. In this case, the selection type is analogous to the resources and the programmer is the consumer. The selection conversion is an inference mechanism for determining the choice of resource on the programmer's behalf.

4.2 Evaluating Code Literals

We can use selection types to capture the meaning of code literals. This is demonstrated by the equivalences defined in Figure 24 which use the notation from the definition below.

Definition 4.1. The following notation is used in Figure 24.

- $\mathscr{R}$ is the set of all *parse rules*: a set of functions which accept a sequence of tokens and produce a selection bag of AST node values. Each element is of the form r_π, where π is the set of output types for the rule r.
- $r_\pi(c)$ is used to describe the processing of the token sequence c by the parse rule r_π. This is an evaluation which is performed by the compiler and so can be used to affect the static typing of object program expressions.
- $x \mathrel{\cancel{\mathcal{T}}} \tau$ is used to indicate that x has the runtime type τ. This is distinct from $x : \tau$ in that the latter expresses a statically provable relationship in the object program. The prior expresses a dynamically proven relationship in the metaprogram.
- $e \overset{+}{\Rightarrow} v$ is used to indicate that metaprogram expression e evaluates to metaprogram expression v at metaprogram runtime.

[2] Chapter 5 of *The Java Language Specification* [8] provides a complete definition of conversions and conversion contexts in the Java language.

Code Literal Equivalence Rule

$$\frac{}{\Gamma \vdash \texttt{<:}c\texttt{:>} \cong \{r_\pi(c) : r_\pi \in \mathscr{R} \wedge r_\pi(c) \overset{+}{\Rightarrow} v\}_{\&}}$$

Parse Execution Type Rule

$$\frac{r_\pi(c) \overset{+}{\Rightarrow} v \qquad v \overset{+}{\nRightarrow} \tau' \qquad \tau' <: \tau \qquad \tau \in \pi}{\Gamma \vdash r_\pi(c) : \tau}$$

Figure 24. Code Literal Rules

```
1 LocalVariableDeclarationNode n1 = <: TypeNode t = <: 5 :>; :>;
2 LocalVariableDeclarationNode n2 = <: int x = y; :>;
```

Figure 25. Nested Code Literal Example

The Code Literal Equivalence Rule in Figure 24 is used to specify a selection bag to which the code literal is equivalent. If there exists exactly one value in the selection bag which has a type corresponding to the expected type in the code literal's context, the Selection Type Projection Rule would project that value, making the code literal expression equivalent to it. To accommodate these semantics, the reference implementation lifts the projected value into an expression which will, when executed, produce that value. This expression then replaces the code literal in the AST after type-checking is complete. This is to say that, after the process of lifting is complete, the code in Figure 12a will be replaced by the code in Figure 12b.

BSJ code literals are not typechecked across multiple stages. Consider the code in Figure 25. The inner code literal on line 1 contains an expression which appears to be in error; there seems to be no context into which that code can be inserted without causing a type error (because an integer literal is not a type). In line 2, however, we create an AST which expresses the instantiation of a variable `x` and its initialization to the value of `y`. Unlike some metaprogramming languages, BSJ cannot guarantee in which scope the generated code will be used; this is based on the runtime semantics of the metaprogram. For this reason, it is impossible to know at metaprogram compile time if the code literal will be inserted into a context where the type of `y` will be assignable to `int` (or even if `y` will be bound!). Likewise, it would be incorrect to assume that the name `TypeNode` in the first line is bound to the BSJ AST type of that name.

While BSJ cannot perform n-stage typechecking of metaprograms, this does not appear to be a concern in practice. Metaprograms are executed by the compiler, meaning that any errors of this nature will be detected before a binary object program is produced.

4.3 Code Splicing

Code literals in BSJ permit *splicing*. As with quasiquotes in LISP or Scheme, splicing allows the value of a metaprogram AST expression to be used in literal code. This is modeled in the BSJ AST by typing each node reference as a disjoint union between its expected type and the type `SpliceNode`;

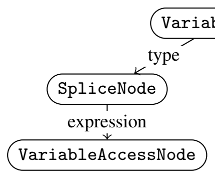


Figure 26. `<:~:vartype:~ x:>` splicing as `TypeNode`

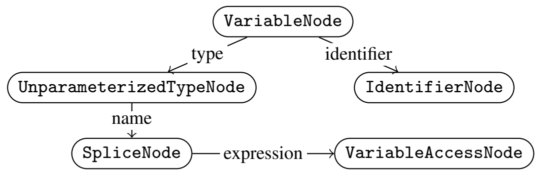


Figure 27. `<:~:vartype:~ x:>` splicing as `NameNode`

Type ::= *PrimitiveType* | *ReferenceType* | *Splice(*`TypeNode` − {`PrimitiveTypeNode` ∪ `ReferenceTypeNode`}*)*
ReferenceType ::= *UnparamType* | ⋯ | *Splice(*`ReferenceTypeNode` − {`UnparameterizedTypeNode` ∪ ⋯}*)*
UnparamType ::= *Name* | *Splice(*`UnparameterizedTypeNode`*)*
Name ::= *Identifier* | *Name* . *Identifier* | *Splice(*`NameNode`*)*
Identifier ::= *IdentifierToken* | *Splice(*`IdentifierNode`*)*

Figure 28. Splice Grammar Rule Example

for instance, `SimpleNameNode` has a reference which is effectively of type `IdentifierNode` ∪ `SpliceNode`.[3]

Consider the code literal `<:~:vartype:~ x:>` in the context of a variable declaration. The expression `vartype` refers to a variable declared in the context of the metaprogram containing the code literal. The splice clearly represents the type of the variable; the token `x` is the identifier to which it is bound (such as `int x`). Evaluating that code literal produces the AST in Figure 26.

Unfortunately, splicing is not this simple. The AST we produced in Figure 26 assumes that the splice expression is a `TypeNode`, the type of the `type` property for a `VariableNode`. Figure 27 shows the AST that would be constructed if `vartype` was bound to a `SimpleNameNode`. Thus, the AST that is built depends upon the metaprogram scope in which we interpret the splice.

To address this problem without introducing grammar ambiguity, we introduce a parameter to the splice grammar rule and use this parameter in a semantic predicate [13] determining whether or not the rule can be accepted. The argument we provide is a set of types in the form $\tau - \{\tau_1 \cup \cdots \cup \tau_n\}$. If, in the context of the metaprogram, the expression which appears in the splice is not of a type which is assignable to the argument, the splice does not parse.

For instance, consider the abbreviated grammar rules for types which appear in Figure 28. Because

[3] Java's type system does not support a general disjoint union type. In specific cases such as this one, however, they can be encoded.

`ReferenceTypeNode` is a subtype of `TypeNode`, the *Type* rule ignores splices of type `ReferenceTypeNode` and allows the *ReferenceType* rule to capture them instead. Because `UnparameterizedTypeNode` and `NameNode` are not subtypes, this precaution is not necessary for *UnparamType*. This approach permits the parsing of code literals to disambiguate splices by using the metaprogram type environment.

4.4 Analysis

There are two key effects of code literals compared with the approach outlined in [3] which we believe to be of interest. First, because we have deferred lifting the code literal until after typechecking, we are able to exploit performance optimizations which were not previously available. For instance, consider the assignment expression `IdentifierNode i = <:x:>`. Rather than evaluating each parse of the literal immediately, we can wait until a selection conversion $\tau' \twoheadrightarrow \tau$ and evaluate only those parse rules which are compatible with τ. This prevents the unnecessary parsing and lifting which would inevitably be eliminated by the typechecking process.

Additionally, the semantic impact of the BSJ code literal is more evident. Consider the following method signatures:

```
String foo(A a, B b);
int foo(B b, A a);
```

If some variable `c` of a node type `C` exists which is a subtype of both `A` and `B`, §15.12.2.5 of [8] indicates that the method invocation `String s = foo(c,c)` is ambiguous (because it is not clear which overloading is intended). If `c` : $\{\texttt{C}\}^{\&}$, then this is also true with code literals. [3], however, generalizes disambiguation up to statements when quasiquoting is used and would thus select the first method signature above (as its return type satisfies the assignment). This discontinuity arises because the disambiguation technique used in [3] does not match the one specified by the Java language. BSJ resolves this issue by integrating selection types into the conversion rules in the host language's type system.

5. Implementation

We have developed a BSJ compiler reference implementation consisting of 54,000 lines of source and 193,000 lines of generated code. This generated code largely represents the numerous design patterns which appear throughout the BSJ compiler (underscoring the need for compile-time metaprogramming in Java). The implementation is comprised of three parts. The first is the BSJ API, which all BSJ compiler implementations are expected to meet; it includes diagnostic interfaces, utilities, and over 200 types of AST nodes. The AST type hierarchy was structured to ensure that as many errors as possible are caught by the Java type system.

The second part of the compiler project is the reference implementation of the API. This implementation supports the full Java language and, while not yet complete, is capable of operating correctly on all of the examples in this paper. We have developed a Java typechecker, allowing metaprograms to be aware of and sensitive to type declarations. The typechecker evaluates lazily; metaprograms tend to typecheck only part of the AST at a time, making the lazy design considerably more efficient.

The third part of the project is a set of BSJ standard libraries, which includes meta-annotations such as `@@Memoized` and `@@BigIntegerOperatorOverloading` shown in Figure 7b. It also currently includes meta-annotations implementing design patterns such as Builder, Observer, and Proxy as well as useful code manipulations such as loop unrolling and method delegation.

6. Related Work

To our knowledge, no previous work exists which models metaprogramming using the difference-based technique we have described. However, the designs of this model and of BSJ draw heavily from the work listed below.

Template Haskell BSJ was largely inspired by existing compile-time metaprogramming literature [7, 17] and other theoretical metaprogramming work [16, 23]. In many ways, BSJ is most similar to Template Haskell: metaprograms use an embedded syntax, have a clearly-defined execution order, and have access to declarations in scope at the call site.

Unlike Template Haskell, however, BSJ allows non-local changes as discussed in Section 2.1. Non-local changes are critical for expressing complex Java abstractions (such as `@@ComparedBy`) without duplication of logic. Template Haskell, on the other hand, restricts the programmer to inserting code into the metaprogram's position in the AST.

Template Haskell provides n-stage typechecking: a metaprogram code fragment `[|x|]` cannot be constructed unless the identifier `x` is bound by the object program in the scope where the metaprogram. BSJ cannot provide n-stage typechecking because non-local changes make identifying the bindings of a code literal undecidable. We consider this tradeoff acceptable because any resulting typechecking errors will manifest at object program compilation time, allowing the programmer to address them accordingly.

Scheme Scheme and BSJ both treat code fragments as simple ASTs. Scheme macros only expand in-place, however, and have the concern of variable hygiene – preventing global names used in macros from being locally shadowed – which is addressed by built-in tools like `gensym`. BSJ metaprograms share the concern of variable hygiene but, since metaprograms may inspect their environemnts, it is possible to determine which variables are bound in scope through use of simple library functions.

Scheme has been used to demonstrate the value of abstracting design patterns using metaprogramming [25]. BSJ introduces this philosophy to the Java language using an annotation syntax which is more amenable to the Java community. BSJ metaprograms are also somewhat simpler to use than Scheme macros for non-generative purposes such as static analysis.

Groovy and OJ Groovy [5] and OJ [20] are Java-like languages which allow compile-time metaprogramming through meta-object protocols. OJ programmers write metaprogramming modules which specify new declaration-modifying keywords. Groovy is a dynamically typed Java language derivative which uses annotations to drive AST transformations (similar to BSJ meta-annotations). Both Groovy and OJ provide MOPs which allow some form of non-local changes. However, neither language is capable of detecting conflicts if they occur; thus, they admit ambiguous programs which are not detected at compile-time, a quite undesirable property.

MetaAspectJ MetaAspectJ [9] is a compile-time metaprogramming system for the AspectJ language, a Java language extension for aspect-oriented programming. Unlike the other languages listed here, MetaAspectJ attempts to perform inference over a significant number of quasiquoting forms. Unlike BSJ, quasiquotes in MetaAspectJ are disambiguated using a simple type system; it is unclear from the paper whether this type system would be sufficient to disambiguate BSJ code literals. MetaAspectJ also permits annotation-driven transformations of ASTs but, like Groovy and OJ, does not detect conflicts between metaprograms.

OpenC++ OpenC++ [4] is a C++ language extension quite similar to OJ: the programmer may use the provided MOP to enhance the syntax of the language. These new syntactic constructs are defined with handlers which specify AST transformations. OpenC++ programs are not ambiguous, but this is not by virtue of conflict detection; like Scheme, transformations are only permitted to expand in place, disallowing non-local changes.

ROSE The ROSE compiler infrastructure [14] is unique in that all of its program transformations must have a total ordering. Rather than assembling metaprogram fragments which are embedded in the object program source, ROSE requires the metaprogrammer to write a single metaprogram which executes all transformations in sequence. This technique applies to a different set of domains; while ROSE coordinates large, sweeping changes across a source base, BSJ's inline syntax is more suitable for abstracting design patterns and other descriptive code properties.

Stratego Stratego/XT [24] is a framework which abstracts the task of building program transformation systems, providing a language for specifying source-to-source transformations. While Stratego is a very general system, it is targeted toward more localized transformations than that of BSJ. It also assumes that the transformation pipeline can be expressed as a composition sequence, making it presently unsuitable for difference-based metaprogramming.

Runtime Metaprogramming Mint [26], MetaOCaml [18] and MetaML [19] are *runtime* metaprogramming extensions of Java, OCaml, and ML (respectively). Runtime metaprogramming eases the process of generating code based on runtime input (such as by inlining an input matrix into an image transformation routine for performance). The user then lifts the generated code into the current runtime. While not directly related to BSJ, runtime metaprogramming systems are based on like principles.

7. Conclusions

We have explored difference-based metaprogramming, a technique which permits a metaprogrammer to effect non-local changes without losing the crcitical guarantee of unambiguous compilation. We presented an algebra to define the semantics of this programming technique and related it to our reference implementation of BSJ, a Java language extension for difference-based metaprogramming. We also demonstrated a technique by which the meaning of code fragments in metaprograms can be inferred by introducing a type family based on linear logic. BSJ metaprograms are easier to understand and maintain than their equivalent Java counterparts due to BSJ's terse and expressive syntax.

We ensure that non-local changes do not introduce ambiguity by treating metaprograms as transformation *generators*: rather than executing each metaprogram in turn over the same object program, we use the metaprograms to produce edit scripts. These edit scripts are then merged and, in the event of a successful merge, are applied to the original object program to produce the final result. A successful merge implies that any legal ordering of the edit scripts we select will have the same effect; a merge failure indicates that two metaprograms are in conflict. A proof of these property appears in Appendix A.

We also demonstrate a variation on the disambiguation technique described in [3]. We defer lifting of object program code until after typechecking, a technique that permits us further opportunities for optimization and that, in BSJ, integrates more smoothly into the Java language specification.

7.1 Future Work

As discussed in Section 3.4, the edit script algebra presented here is strictly of a syntactic nature; it has no support for representing semantic information. Techniques for semantic software merging have already been developed [12], but it remains to be seen how those techniques would apply in a difference-based metaprogramming environment.

BSJ metaprograms are currently quite verbose and we are investigating remedies for this problem. The BSJ standard libraries may be expanded to include meta-annotations representing difference-based metaprogramming-specific design patterns; syntactic sugar should prove helpful in cases in which such libraries are insufficient. BSJ metaprograms must also use explicit type coercion on AST nodes as they inspect their environments; further extensions to the BSJ type system could prevent the need for these coercions.

The call sites for BSJ always include explicit delimiter syntax. While this conveniently emphasizes that the code may be changed at compile time, it prevents elegant construction of language extensions in BSJ. A syntax rewriter front-end (similar to Stratego [24] or Maya [1]) would be useful; this would permit new *syntactic* constructs to be reified as metaprograms in BSJ. This would also ensure that no two syntax extensions' encodings conflict with each other.

Before BSJ can achieve widespread adoption, significant library development is necessary. While part of the attraction of compile-time metaprogramming is the ability to express domain-specific encodings, BSJ programmers would benefit considerably from a rich set of general metaprogramming tools and prepared meta-annotations. BSJ will also require an IDE comparable to those available to Java programmers; to this end, we have begun development of a BSJ plugin for Eclipse. This project poses several questions: how should metaprograms be debugged? How are metaprograms' effects visualized? How do we automate refactoring when some of the target code is generated?

Acknowledgments

This work would not have been possible without the efforts of several of our colleagues. Harisanker Menon tolerated many impromptu discussions on the topic. Joseph Riley helped with the core implementation of the compiler; Uday Garikipati and Nathan Krasnopoler contributed to the BSJ standard library. An anonymous PLDI reviewer's feedback led to our investigation of the difference-based approach.

References

[1] J. Baker and W. C. Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *PLDI*, pages 270–281, 2002.

[2] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. *Software Reuse, International Conference on*, 0:143, 1998.

[3] M. Bravenboer, R. Vermaas, J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE05), volume 3676 of Lecture Notes in Computer Science*, pages 157–172. Springer, 2005.

[4] S. Chiba. A metaobject protocol for C++, 1995.

[5] Codehaus Foundation. Groovy - building AST guide, May 2010. `http://groovy.codehaus.org/Building+AST+Guide`.

[6] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994.

[7] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *MacroML. In the International Conference on Functional Programming (ICFP 01*, pages 74–85. ACM Press, 2001.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[9] S. S. Huang, D. Zook, and Y. Smaragdakis. Domain-specific languages and program generation with Meta-AspectJ. *ACM Trans. Softw. Eng. Methodol.*, 18:6:1–6:32, November 2008.

[10] S. Kamin, L. Clausen, and A. Jarvis. Jumbo: Run-time code generation for Java and its applications. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 48–56, Washington, DC, USA, 2003. IEEE Computer Society.

[11] E. Lippe and N. van Oosterom. Operation-based merging. *SIGSOFT Softw. Eng. Notes*, 17:78–87, November 1992.

[12] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28:449–462, May 2002.

[13] T. J. Parr and R. W. Quong. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In *In Computational Complexity*, pages 263–277. Springer-Verlag, 1994.

[14] D. J. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[15] A. D. Robison. Impact of economics on compiler optimization. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, JGI '01, pages 1–10, New York, NY, USA, 2001. ACM.

[16] T. Sheard. Accomplishments and research challenges in metaprogramming. In *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44, London, UK, 2001. Springer-Verlag.

[17] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, 2002.

[18] W. Taha and *et. al.* MetaOCaml: A compiled, type-safe multi-stage programming language. `http://www.metaocaml.org/`.

[19] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2): 211–242, 2000.

[20] M. Tatsubori and S. Chiba. Programming support of design patterns with compile-time reflection, 1998.

[21] L. Tratt. Compile-time meta-programming in a dynamically typed oo language. In *Proceedings of the 2005 symposium on Dynamic languages*, DLS '05, pages 49–63, New York, NY, USA, 2005. ACM.

[22] T. L. Veldhuizen. C++ templates are turing complete. Technical report, 2003.

[23] T. L. Veldhuizen. Tradeoffs in metaprogramming. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 150–159, New York, NY, USA, 2006. ACM.

[24] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT 0.9, 2004.

[25] D. von Dincklage. Making patterns explicit with metaprogramming. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 287–306, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[26] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha. Mint: Java multi-stage programming using weak separability. In *ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI '10)*, June 2010.

A. Proof of Correctness

The determinism of difference-based metaprogramming relies on the crucial property that, for all orderings $\bar{k}$ of a set of metaprograms, the sequences $\vartheta(\bar{\delta}_{k_n}, (\cdots \vartheta(\bar{\delta}_{k_1}, \emptyset))$ of the edit scripts are equivalent, either in that they produce equivalent object programs or in that they all fail to merge. This appendix provides a proof of that property based on the semantics in Section 3.

A general outline of this proof is as follows. First, we show that any pairing of edit script elements which are produced by unordered metaprograms are commutative up to equivalence. Then, we demonstrate that the edit scripts of unordered metaprograms commute up to equivalence. Finally we demonstrate that, if no conflicts exist, a merge over any respectful ordering is equivalent to a merge over every other respectful ordering.

We start by defining a lemma for the commutativity of the record substitution operator.

Lemma A.1. *The record substitution operator is commutative for different substitution keys; that is, $r[\![k \mapsto v]\!][\![k' \mapsto v']\!] = r[\![k' \mapsto v']\!][\![k \mapsto v]\!]$ when $k \neq k'$. Further, the record substitution operator is commutative when both the substitution keys and values are the same.*

Proof. Trivial by the nature of mappings. □

Next, we highlight that any edit script that performs operations over the wrong type of node will always commute with every other edit script.

Lemma A.2. *Any record assignment element which refers to a list node commutes with every other edit script element. Any list add or list remove element which refers to a record node commutes with every other edit script element.*

Proof. Let δ be an edit script of the form $\eta.l \leftarrow \hat{v}$ and assume that $\eta \mapsto \mathcal{L} \in \rho$. The application $\delta\ \rho$ diverges as no evaluation rule matches it. Thus, for any δ', we have $\delta\ (\delta'\ \rho) \cong \delta'\ (\delta\ \rho)$ (since both sides diverge).

For list add and list remove elements, the same strategy is applied with $\eta \mapsto \mathcal{R}$. □

In the definition of Lemmas A.3, A.4, A.5, A.6, and A.7, we assume that the input elements are applied to the correct types (since any case in which this is not true can be proven to commute by the above).

We can now demonstrate that individual edit script elements generated by unordered metaprograms either commute or are defined to conflict. To do so, we must analyze each pairwise case of edit script element types.

Lemma A.3. *Given any two edit script elements $\mathcal{M} \succ \delta$ and $\mathcal{M}' \succ \delta'$ where $\mathcal{M} \not\lessgtr \mathcal{M}'$ and either edit script element is a record creation or list creation element, then either $\delta\ (\delta'\ \rho) \cong \delta'\ (\delta\ \rho)$ or $\delta \not\leftrightarrow \delta'$.*

Proof. Assume w.l.o.g. that δ is either a record creation element or a list creation element containing the nonce η. If $\eta \in \delta'$, then $\delta \not\leftrightarrow \delta'$ by the Unordered Creation Conflict Rule. Otherwise, as indicated in Figure 19, the application of any type of edit script results in the substitution of exactly one nonce in the input program. Because $\eta \notin \delta'$, its semantics do not substitute η. The semantics of δ involve the substitution η and no other nonce. By Lemma A.1, these edit script elements commute. □

Lemma A.4. *Given any two edit script elements $\mathcal{M} \succ \delta$ and $\mathcal{M}' \succ \delta'$ where $\mathcal{M} \not\lessgtr \mathcal{M}'$ and one edit script element is a record assignment, either $\delta\ (\delta'\ \rho) \cong \delta'\ (\delta\ \rho)$ or $\delta \not\leftrightarrow \delta'$.*

Proof. Suppose w.l.o.g. that δ is a record assignment element of the form $\eta.l \leftarrow \hat{v}$. If δ' is a creation element, then these two elements commute by Lemma A.3. Otherwise, the application of δ' either substitutes η or it does not. If it does not, these edit script elements commute by Lemma A.1.

If it does, then δ' is either a record assignment element, a list addition element, or a list removal element. If δ' is not a record assignment element, then it commutes with δ via Lemma A.2.

Otherwise, δ' is a record assignment element of the form $\eta'.l' \leftarrow \hat{v}'$. If $\eta \neq \eta'$, then these edit script elements commute by Lemma A.1 as they replace different keys in ρ. Otherwise, if $l \neq l'$ then these edit script elements commute by Lemma A.1 as they replace different keys in the record to which η is mapped in ρ. Otherwise, we have $\eta = \eta'$ and $l = l'$. Either $\hat{v} = \hat{v}'$ (in which case both elements perform the same substitution and therefore commute by Lemma A.1) or $\hat{v} \neq \hat{v}'$ (in which case $\delta \not\leftrightarrow \delta'$ by the Value Assignment Conflict Rule in Figure 21). □

Lemma A.5. *Given any two edit script elements $\mathcal{M} \succ \delta$ and $\mathcal{M}' \succ \delta$ where $\mathcal{M} \not\lessgtr \mathcal{M}'$ and one edit script element is a list add before element, either $\delta\ (\delta'\ \rho) \cong \delta'\ (\delta\ \rho)$ or $\delta \not\leftrightarrow \delta'$.*

Proof. Suppose w.l.o.g. that $\delta = \mathcal{M} \succ \eta_1 : \eta_2 \hookleftarrow \overset{\vartriangle}{\eta}_3$. If δ' is a creation element or a record assignment element, then these elements commute by Lemma A.3 or Lemma A.4 (respectively). Otherwise, δ' is a list addition element or a list removal element. If the list affected by δ' has nonce

η_1' and $\eta_1 \neq \eta_1'$, then these two elements commute by Lemma A.1. Otherwise, $\eta_1 = \eta_1'$ and $\eta_1 \mapsto \mathcal{L} \in \rho$ and we proceed by deconstructing the type of δ'.

Suppose δ' is a list add before element of the form $\mathcal{M}' \succ \eta_1 : \eta_2' \hookleftarrow \overset{\vartriangle}{\eta}_3'$. If $\overset{\vartriangle}{\eta}_3 \neq \overset{\vartriangle}{\eta}_3'$, then these two additions occur in different positions of the list. Suppose w.l.o.g. that $\overset{\vartriangle}{\eta}_3$ appears before $\overset{\vartriangle}{\eta}_3'$ in the list. Let $\mathring{\eta}_3 = \Sigma(\overset{\vartriangle}{\eta}_3, \mathcal{M}, \mathcal{L})$ and $\mathring{\eta}_3' = \Sigma(\overset{\vartriangle}{\eta}_3', \mathcal{M}', \mathcal{L})$. If either of these two evaluations fail, at least one edit script element application does not evaluate and so we have divergence. The commutation of the elements will not affect this as the addition of an element by an unordered metaprogram does not change the result of Σ and $\mathcal{M} \not\rightleftharpoons \mathcal{M}'$.

If Σ succeeds in finding $\mathring{\eta}_3$ and $\mathring{\eta}_3'$, then we have

$$\begin{aligned}
&\delta' \; (\delta \; \{\eta_1 \mapsto [\overline{\mathring{\eta}_*}, \mathring{\eta}_3, \overline{\mathring{\eta}_*'}, \mathring{\eta}_3', \overline{\mathring{\eta}_*''}], \overline{\mathcal{K} \mapsto \hat{v}}\}) \\
\Rightarrow \;\; &\delta' \; \{\eta_1 \mapsto [\overline{\mathring{\eta}_*}, (\eta_2, \mathcal{M}, \emptyset), \mathring{\eta}_3, \overline{\mathring{\eta}_*'}, \mathring{\eta}_3', \overline{\mathring{\eta}_*''}], \overline{\mathcal{K} \mapsto \hat{v}}\} \\
\Rightarrow \;\; &\{\eta_1 \mapsto [\overline{\mathring{\eta}_*}, (\eta_2, \mathcal{M}, \emptyset), \mathring{\eta}_3, \overline{\mathring{\eta}_*'}, (\eta_2', \mathcal{M}', \emptyset), \mathring{\eta}_3', \overline{\mathring{\eta}_*''}], \overline{\mathcal{K} \mapsto \hat{v}}\}
\end{aligned}$$

and

$$\begin{aligned}
&\delta \; (\delta' \; \{\eta_1 \mapsto [\overline{\mathring{\eta}_*}, \mathring{\eta}_3, \overline{\mathring{\eta}_*'}, \mathring{\eta}_3', \overline{\mathring{\eta}_*''}], \overline{\mathcal{K} \mapsto \hat{v}}\}) \\
\Rightarrow \;\; &\delta \; \{\eta_1 \mapsto [\overline{\mathring{\eta}_*}, \mathring{\eta}_3, \overline{\mathring{\eta}_*'}, (\eta_2', \mathcal{M}', \emptyset), \mathring{\eta}_3', \overline{\mathring{\eta}_*''}], \overline{\mathcal{K} \mapsto \hat{v}}\} \\
\Rightarrow \;\; &\{\eta_1 \mapsto [\overline{\mathring{\eta}_*}, (\eta_2, \mathcal{M}, \emptyset), \mathring{\eta}_3, \overline{\mathring{\eta}_*'}, (\eta_2', \mathcal{M}', \emptyset), \mathring{\eta}_3', \overline{\mathring{\eta}_*''}], \overline{\mathcal{K} \mapsto \hat{v}}\}
\end{aligned}$$

Thus, these two list add before elements commute when they refer to different positions in the list. If they refer to the same position in the list, we have $\overset{\vartriangle}{\eta}_3 = \overset{\vartriangle}{\eta}_3'$. Either one of η_2 and η_2' is unordered or both are ordered. If both are ordered, these two edit script elements conflict by the Add Before Conflict Rule; otherwise, they commute by Definition 3.9 (program equivalence). Therefore, two list add before elements from unordered metaprograms always commute.

Otherwise, suppose δ' is a list add after element of the form $\mathcal{M}' \succ \eta_1 : \overset{\vartriangle}{\eta}_3' \hookrightarrow \eta_2'$. When these two elements refer to different positions in the list or when Σ returns different list elements for each metaprogram, the strategy described above for list add before elements suffices to show that the elements commute. In the case that $\overset{\vartriangle}{\eta}_3 = \overset{\vartriangle}{\eta}_3'$ and $\mathring{\eta}_3 = \mathring{\eta}_3'$, we have

$$\begin{aligned}
&\delta' \; (\delta \; \{\eta_1 \mapsto [\overline{\mathring{\eta}_*}, \mathring{\eta}_3, \overline{\mathring{\eta}_*'}], \overline{\mathcal{K} \mapsto \hat{v}}\}) \\
\Rightarrow \;\; &\delta' \; \{\eta_1 \mapsto [\overline{\mathring{\eta}_*}, (\eta_2, \mathcal{M}, \emptyset), \mathring{\eta}_3, \overline{\mathring{\eta}_*'}], \overline{\mathcal{K} \mapsto \hat{v}}\} \\
\Rightarrow \;\; &\{\eta_1 \mapsto [\overline{\mathring{\eta}_*}, (\eta_2, \mathcal{M}, \emptyset), \mathring{\eta}_3, (\eta_2', \mathcal{M}', \emptyset), \overline{\mathring{\eta}_*'}], \overline{\mathcal{K} \mapsto \hat{v}}\}
\end{aligned}$$

and

$$\begin{aligned}
&\delta' \; (\delta \; \{\eta_1 \mapsto [\overline{\mathring{\eta}_*}, \mathring{\eta}_3, \overline{\mathring{\eta}_*'}], \overline{\mathcal{K} \mapsto \hat{v}}\}) \\
\Rightarrow \;\; &\delta' \; \{\eta_1 \mapsto [\overline{\mathring{\eta}_*}, \mathring{\eta}_3, (\eta_2', \mathcal{M}', \emptyset), \overline{\mathring{\eta}_*'}], \overline{\mathcal{K} \mapsto \hat{v}}\} \\
\Rightarrow \;\; &\{\eta_1 \mapsto [\overline{\mathring{\eta}_*}, (\eta_2, \mathcal{M}, \emptyset), \mathring{\eta}_3, (\eta_2', \mathcal{M}', \emptyset), \overline{\mathring{\eta}_*'}], \overline{\mathcal{K} \mapsto \hat{v}}\}
\end{aligned}$$

Thus, when δ is a list add before element and δ' is a list add after element, these edit script elements commute.

Otherwise, δ' must be a list remove element of the form $\mathcal{M}' \succ \eta_1 : \downarrow \eta_3'$. We know that $\mathcal{M} \not\rightleftharpoons \mathcal{M}'$; suppose that $\eta_1 \mapsto \mathcal{L} \in \rho$. The application of δ' will add $\mathcal{M}'$ to the removal set of some cell; the application of δ will invoke $\Sigma(\overset{\vartriangle}{\eta}_3, \mathcal{M}, \mathcal{L})$. We have that $\Sigma(\overset{\vartriangle}{\eta}_3, \mathcal{M}, \mathcal{L})$ is unaffected by the actions of δ' because it is based upon the set $\{(\overset{\vartriangle}{\eta}', \mathcal{M}'', \mathcal{S}) : \mathcal{L} \mid \mathcal{M} \rightleftharpoons \mathcal{M}'' \wedge \forall \mathcal{M}''' \in \mathcal{S}. \mathcal{M} \not\rightharpoonup \mathcal{M}'''\}$. Because $\mathcal{M} \not\rightleftharpoons \mathcal{M}'$, the addition of $\mathcal{M}'$ to a removal set will not change this set. We use a similar strategy to demonstrate that $\Sigma(\overset{\vartriangle}{\eta}_3', \mathcal{M}', \mathcal{L})$ will be unaffected by δ and then use the strategy above to demonstrate commutativity between δ and δ'. □

Lemma A.6. *Given any two edit script elements $\mathcal{M} \succ \delta$ and $\mathcal{M}' \succ \delta'$ where $\mathcal{M} \not\rightleftharpoons \mathcal{M}'$ and one edit script element is a list add after element, either $\delta \; (\delta' \; \rho) \cong \delta' \; (\delta \; \rho)$ or $\delta \nleftrightarrow \delta'$.*

Proof. By the same strategy as Lemma A.5. □

Lemma A.7. *Given any two edit script elements $\mathcal{M} \succ \delta$ and $\mathcal{M}' \succ \delta'$ where $\mathcal{M} \not\rightleftharpoons \mathcal{M}'$ and one edit script element is a list removal element, $\delta \; (\delta' \; \rho) \cong \delta' \; (\delta \; \rho)$.*

Proof. Suppose w.l.o.g. that δ has the form $\mathcal{M} \succ \eta_1 : \downarrow \eta_2$. If δ' is not a list removal element, then these two elements commute by one of Lemma A.3, A.4, A.5, or A.6. Otherwise, δ' has the form $\mathcal{M}' \succ \eta_1' : \downarrow \eta_2'$. If $\eta_1 \neq \eta_1'$, then these two edit script elements commute by Lemma A.1.

Otherwise, $\eta_1 = \eta_1'$. If $\eta_2 \neq \eta_2'$, then these two edit scripts affect different portions of the list and clearly commute. Otherwise, both edit script elements have the form $\eta_1 : \downarrow \eta_2$. We observe that the semantics of this operation place the ID of the metaprogram in the removal set for the element corresponding to η_2.

In this case, suppose $\eta_1 \mapsto \mathcal{L} \in \rho$. Let $\mathring{\eta}_2 = \Sigma(\overset{\vartriangle}{\eta}_2, \mathcal{M}, \mathcal{L})$ and let$\mathring{\eta}_2' = \Sigma(\overset{\vartriangle}{\eta}_2, \mathcal{M}', \mathcal{L})$. If $\mathring{\eta}_2 \neq \mathring{\eta}_2'$ then these edit script elements replace different elements in the list (by adding to their respective removal sets) and therefore commute by Lemma A.1.

Otherwise, $\mathring{\eta}_2 = \mathring{\eta}_2'$. We show that each element's search function is unaffected by the semantics of the application of the other element. w.l.o.g., we choose to show that $\Sigma(\overset{\vartriangle}{\eta}_2, \mathcal{M}, \mathcal{L})$ is unaffected by the addition of $\mathcal{M}'$ to the removal set of $\mathring{\eta}_2$. The result of $\Sigma(\overset{\vartriangle}{\eta}_2, \mathcal{M}, \mathcal{L})$ is determined by the set $\{(\overset{\vartriangle}{\eta}', \mathcal{M}'', \mathcal{S}) : \mathcal{L} \mid \mathcal{M} \rightleftharpoons \mathcal{M}'' \wedge \forall \mathcal{M}''' \in \mathcal{S}. \mathcal{M} \not\rightharpoonup \mathcal{M}'''\}$. Because $\mathcal{M} \not\rightleftharpoons \mathcal{M}'$, this set is the same whether or not $\mathcal{M}'$ is in any element of $\mathcal{L}$.

Finally, we show that these elements commute in their behavior. The semantics of a list removal involves the addition of the metaprogram's ID to the list cell's removal set. Thus, the removal set of $\mathring{\eta}_2$ will be $\mathcal{S} \cup \mathcal{M} \cup \mathcal{M}'$ (if δ is applied first) or $\mathcal{S} \cup \mathcal{M}' \cup \mathcal{M}$ (if δ' is applied first). Because set union is commutative, these sets are equivalent. Therefore, δ and δ' are commutative. □

Lemma A.8. *Given any two edit script elements $\mathcal{M} \succ \delta$ and $\mathcal{M}' \succ \delta'$ where $\mathcal{M} \not\rightleftharpoons \mathcal{M}'$, either $\delta \; (\delta' \; \rho) \cong \delta' \; (\delta \; \rho)$ or $\delta \nleftrightarrow \delta'$.*

Proof. By Lemmas A.3, A.4, A.5, A.6, and A.7. □

Now that we have proven commutativity up to equivalence of individual edit script elements, we begin proving properties about whole edit scripts. First, we demonstrate that the merge of two edit scripts containing only elements from different, unordered metaprograms is commutative up to equivalence if the merge does not fail.

Lemma A.9. *Given any two edit scripts $\bar{\delta}$ and $\bar{\delta}'$ such that $\forall(\mathcal{M} \succ \delta, \mathcal{M}' \succ \delta') \in \bar{\delta} \times \bar{\delta}'.\mathcal{M} \not\rightleftarrows \mathcal{M}'$, $\bar{\delta} \overset{?}{\rightarrowtail} \bar{\delta}'$ implies that $\bar{\delta} \rightarrowtail \bar{\delta}' \cong \bar{\delta}' \rightarrowtail \bar{\delta}$.*

Proof. For every pair of edit script elements $\mathcal{M} \succ \delta \in \bar{\delta}$ and $\mathcal{M}' \succ \delta' \in \bar{\delta}'$, we know that $\mathcal{M} \not\rightleftarrows \mathcal{M}'$. If for any such pairing we have $\delta \not\leftrightarrow \delta'$, then the merge fails by Definition 3.11 and we are finished.

Otherwise, there exists no such pairing. By Definition 3.11, we have that $\bar{\delta} \rightarrowtail \bar{\delta}' = \delta_1, \ldots, \delta_n, \delta'_1, \ldots, \delta'_m$. By induction over Lemma A.8, we have

$$\begin{aligned}
\bar{\delta} \rightarrowtail \bar{\delta}' &= \delta_1, \ldots, \delta_{n-1}, \delta_n, \delta'_1, \delta'_2, \ldots, \delta'_m \\
&= \delta_1, \ldots, \delta_{n-1}, \delta'_1, \delta_n, \delta'_2, \ldots, \delta'_m \\
&= \delta'_1, \delta_1, \ldots, \delta_n, \delta'_2, \ldots, \delta'_m \\
&= \delta'_1, \ldots, \delta'_m, \delta_1, \ldots, \delta_n \\
&= \bar{\delta}' \rightarrowtail \bar{\delta}
\end{aligned}$$

□

We now specify an intermediate lemma which nearly completes our proof.

Lemma A.10. *Given metaprograms $\mathcal{M}_1, \ldots, \mathcal{M}_n$ in a dependency graph G, let $\bar{\delta}_1, \ldots, \bar{\delta}_n$ be the edit scripts produced by those metaprograms. For any respectful ordering $\bar{k}$ of these metaprograms, a successful merge $\bar{\delta}_1 \rightarrowtail \ldots \rightarrowtail \bar{\delta}_n$ implies that none of the metaprograms are in conflict.*

Proof. By Definition 3.2, it suffices to show that, for every pair of respectful orderings $\bar{k}$ and $\bar{k}'$, $\bar{\delta}_{k_1} \rightarrowtail \cdots \rightarrowtail \bar{\delta}_{k_n} \cong \bar{\delta}_{k'_1} \rightarrowtail \cdots \rightarrowtail \bar{\delta}_{k'_n}$.

w.l.o.g., let $\bar{k}$ and $\bar{k}'$ be any two respectful orderings over the metaprograms $\mathcal{M}_1, \ldots, \mathcal{M}_n$. Because $\bar{k}$ and $\bar{k}'$ are orderings over the same set of metaprograms, $\bar{k}'$ is a permutation of $\bar{k}$. Therefore, there exists a set $\mathcal{J}$ of lists of inversions (pairs of positions representing an adjacent transposition) which transform $\bar{k}'$ into $\bar{k}$. Let $\mathcal{I}$ be a list of inversions in $\mathcal{J}$ such that no other list in $\mathcal{J}$ has fewer members. We know that $\mathcal{I}$ does not contain any inversions between ordered metaprograms; if such an inversion existed, its reverse would also exist (since ordered metaprograms are correctly ordered in $\bar{k}'$ with respect to $\bar{k}$ by the definition of a respectful ordering) and $\mathcal{I}$ would no longer be minimal.

We proceed by induction on the length of $\mathcal{I}$. If $|\mathcal{I}|$ is zero, then $\bar{k} = \bar{k}'$ and thus $\bar{\delta}_{k_1} \rightarrowtail \cdots \rightarrowtail \bar{\delta}_{k_n} = \bar{\delta}_{k'_1} \rightarrowtail \cdots \rightarrowtail \bar{\delta}_{k'_n}$ and we are done.

Otherwise, $\mathcal{I}$ contains at least one inversion; let ι be that inversion and let $\mathcal{I}'$ be the remainder of the list. Let $\bar{k}''$ be the ordering created by applying ι to $\bar{k}'$. We know that this inversion is between unordered metaprograms, so we apply Lemma A.9 and thus have that $\bar{\delta}_{k''_1} \rightarrowtail \cdots \rightarrowtail \bar{\delta}_{k''_n} \cong \bar{\delta}_{k'_1} \rightarrowtail \cdots \rightarrowtail \bar{\delta}_{k'_n}$. Furthermore, $\mathcal{I}'$ is now a list of inversions which transforms $\bar{k}''$ into $\bar{k}$ and $|\mathcal{I}'| < |\mathcal{I}|$. Thus, by induction, $\bar{\delta}_{k_1} \rightarrowtail \cdots \rightarrowtail \bar{\delta}_{k_n} \cong \bar{\delta}_{k'_1} \rightarrowtail \cdots \rightarrowtail \bar{\delta}_{k'_n}$. □

Finally, we demonstrate the well-foundedness of our conflict detection system by proving Theorem 1 (which appears at the end of Section 3.4):

Proof. Let $\mathcal{M}'_1 = \mathcal{M}_\alpha$ and, for all $i \in 1 \ldots n$, let $\mathcal{M}'_{i+1} = \mathcal{M}_i$. We have that, for all $i \in 2 \ldots m$, $\mathcal{M}'_m \rightsquigarrow \mathcal{M}'_1$ (because of the nature of $\mathcal{M}_\alpha$). By Lemma A.10, none of the metaprograms $\mathcal{M}'_1, \ldots, \mathcal{M}'_n$ are in conflict. By Definition 3.2 this implies that, for every pair of respectful orderings $\bar{k}$ and $\bar{k}'$, we have $\vartheta(\bar{\delta}_{k_n}, (\cdots \vartheta(\bar{\delta}_{k_1}, \emptyset)) \cong \vartheta(\bar{\delta}_{k_n}, (\cdots \vartheta(\bar{\delta}_{k_1}, \emptyset))$. □

Theorem 1 demonstrates that every ordering of edit scripts is equivalent if any of them successfully merge. This gives us the guarantee that any metaprogram execution which follows the semantics of this algebra will always produce an unambiguous output. If we assume that the BSJ reference compiler is a faithful implementation of this algebra, then every BSJ program which compiles successfully is unambiguous.