Expressive and Safe Static Reflection with MorphJ

Shan Shan Huang
College of Computing
Georgia Institute of Technology
ssh@cc.gatech.edu

Yannis Smaragdakis
Department of Computer and Information Science
University of Oregon
yannis@cs.uoregon.edu

Abstract

Recently, language extensions have been proposed for Java and C# to support *pattern-based reflective declaration*. These extensions introduce a disciplined form of meta-programming and aspect-oriented programming to mainstream languages: They allow members of a class (i.e., fields and methods) to be declared by statically iterating over and pattern-matching on members of other classes. Such techniques, however, have been unable to safely express simple, but common, idioms such as declaring getter and setter methods for fields.

In this paper, we present a mechanism that addresses the lack of expressiveness in past work without sacrificing safety. Our technique is based on the idea of nested patterns that elaborate the outer-most pattern with blocking or enabling conditions. We implemented this mechanism in a language, MorphJ. We demonstrate the expressiveness of MorphJ with real-world applications. In particular, the MorphJ reimplementation of DSTM2, a software transactional memory library, reduces 1,107 lines of Java reflection and bytecode engineering library calls to just 374 lines of MorphJ code. At the same time, the MorphJ solution is both high level and safer, as MorphJ can separately type check generic classes and catch errors early. We present and formalize the MorphJ type system, and offer a type-checking algorithm.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords object-oriented programming, structural abstraction, class morphing, aspect-oriented programming, meta-programming, language extensions

1. Introduction

Consider the following task: how would you write a piece of code that, given *any* class `X`, returns another class that contains the exact same methods as `X`, but logs each method's return value? That is, the code is a modular representation of the functionality "logging", and abstracts over the exact methods it may be applied to.

Capturing this level of abstraction has traditionally been only possible with techniques such as meta-object protocols (MOPs) [17], aspect-oriented programming (AOP) [18], or various forms of meta-programming (e.g., reflection and ad-hoc program generation using quote primitives, string templates, or bytecode engineering). While just about any programmer can write a method that logs its return value, the techniques listed above are largely unapproachable. Additionally, these techniques bypass the guarantees offered by the type system so that there is no pre-application guarantee that the resulting code would be well-formed.

Recently, extensions have been proposed [14, 7] for Java and C#, to bridge this gap between the lack of expressiveness in mainstream languages, and the lack of type safety in the more exotic techniques mentioned above. Both extensions introduce a notion of static (as oppose to runtime) iteration over fields or methods of a class, interface, or type variable. A simple pattern language is used to match on the elements being iterated over. A declaration can be made for each element in the iteration, using pattern-matched variables. We call this type of declaration *pattern-based reflective declaration* and the general technique *class morphing*. For example, using MJ, the Java extension we introduced in [14], we can write the following generic class:

```
1 class LogMe<class X> extends X {
2    <R,A*>[m] for ( public R m(A) : X.methods )
3    public R m (A a) {
4       R result = super.m(a);
5       System.out.println(result);
6       return result;
7    }
8 }
```

This MJ generic class extends its own type parameter `X`. Lines 2-7 form a *reflective declaration block*: line 2 defines the range of elements being iterated over; lines 3-7 are the method being declared once *for each* element in the iteration range. *The iteration happens statically.* Line 2 says we want to iterate over all methods of `X` that match the pattern "`public R m (A)`", where `R`, `A`, and `m` are pattern variables (declared before the keyword `for`). `R` and `A` are pattern type variables, where `R` matches any type except `void`, and `A`, because of the `*` notation in `A`'s declaration, matches a sequence of types of any length, including zero. `m` is a name variable, and matches any identifier. Thus, this block iterates over all `public` methods of `X` that take any number of arguments, and have non-`void` returns. For each such method, it declares a method with the same signature. For example, `LogMe<java.io.File>` gives us a version of `File` with all its methods' return values logged. This task can be similarly accomplished with Compile-Time Reflection (CTR) [7], an extension proposed for C#.

MJ and CTR increase the expressiveness of their respective base languages by providing a limited form of meta-programming, through a familiar syntax accessible to the average programmer. They additionally offer a level of static type safety that has never been achieved by mechanisms like MOPs, AOP, or various forms of meta-programming. For instance, MJ preserves the separate type-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.
Copyright © 2008 ACM 978-1-59593-860-2/08/06. . . $5.00

checkability of Java generics: a generic class can be type checked separately from the types it may be instantiated with, and a well-typed generic class will never introduce a type error in its instantiated form. Thus, the class `LogMe<X>` is guaranteed to never contain type errors, no matter what `X` is instantiated to.

However, simple pattern-based reflective declaration is not expressive enough for many common tasks that are otherwise perfectly suited for this type of mechanism.

Example 1: Consider the pesky problem of defining "getter" methods for fields in a class. Currently, programmers deal with this by repeating the same boiler-plate code for each field. This seems to be a task perfectly suited for pattern-based reflective declaration. However, we cannot implement this in a type-safe way using MJ or CTR. Consider the following attempt in MJ:

```
class AddGetter<class X> extends X {
   <F>[f] for ( F f : X.fields )
   F get#f () { return super.f; }
}
```

`AddGetter<X>` defines a method `get#f()` for each field `f` in type variable `X`. `get#f` denotes an identifier that begins with the string "`get`", followed by the identifier matched by `f`.[1] However, `AddGetter<X>` is not modularly type safe—we cannot guarantee that no matter what `X` is instantiated with, `AddGetter<X>` is always well-typed! Suppose we have class `C`:

```
class C {
  Meal lunch;  ... // other methods.
  boolean getlunch() { return isNoon() ? true : false; }
}
```

`AddGetter<C>` contains method "`Meal getlunch()`", which incorrectly overrides method "`boolean getlunch()`" in its superclass, `C`. For this reason, the definition of `AddGetter<X>` does not pass MJ's type-checking.

This is an error of under-specified requirements. The definition of `AddGetter<X>` should clearly specify that it can only declare method `get#f` for those fields `f` where a conflictingly defined `get#f` does not already exist in `X`. However, the pattern-matching language in MJ (or CTR) does not allow us to specify such a condition. What we need is to place a *negative existential condition* on each field matched by the pattern: for all fields `f` of `X` such that method `get#f()` does not already exist in `X`, declare the method `get#f()`.

Example 2: Consider how one would define a class `Pair<X,Y>`, which is a container for objects `x` and `y` of types `X` and `Y`, respectively. For every non-`void` method that `X` and `Y` have in common (i.e., same method name and argument types), `Pair<X,Y>` should declare a method with the same name and argument types, but a return type that is another `Pair`, constructed from the return types of that method in `X` and `Y`.

Again, this task seems perfectly suited for pattern-based reflective declaration. Yet there is no way to define such a class using MJ (or CTR). What we need is a *positive existential condition*: for all methods in `X`, such that another method with the same name and argument types exists in `Y`, declare a method that invokes both and returns a `Pair` of their values.

Contributions: In this work, we introduce *nested patterns*. Nested patterns act as existential conditions that either enable or block the applicability of the outer-most (primary) pattern. We implemented nested patterns in our language, MorphJ, an extension of MJ [14].[2] Nested patterns make the following contributions:

- Nested patterns significantly enhance the expressiveness of pattern-based reflective declarations. Without nested patterns, languages like MJ and CTR are mostly limited to expressing wrapper-like patterns [8], such as the logging class `LogMe<X>`.
- Nested patterns also form the basis for adding more features to the MorphJ language, such as a static `if` and a static `errorif` statements. The static `if` allows code to exist or not depending on the existence or absence of other members. The static `errorif` acts as a type-cast: it allows the type system to assume the existence or absence of a member, and, if the assumption is violated, an instantiation-time error occurs.
- We demonstrate the power of nested patterns with real-world applications. We reimplemented parts of DSTM2 [10], a Java software transactional library that heavily uses reflection and bytecode engineering (with BCEL [2]) to transform sequential classes into transactional ones. The MorphJ re-implementation replaces 1,107 lines of Java reflection and BCEL library calls with 374 lines of MorphJ code. We also implemented a generic class that provides default implementations for unimplemented interface methods, for a combination of any class and any interface. The current solutions to this problem are either implemented as extensions to the underlying language itself, or use unsafe code generation with templates. We show in Section 3 how the MorphJ implementations are easy to write, easy to understand, as well as type safe.
- Nested patterns create additional challenges for type checking. We show the techniques involved to preserve the separate type-checkability of generic classes in MorphJ. We formalize our type system and prove it sound. We offer a decidable algorithm for type-checking.

In the remainder of this paper, we start by introducing the syntax and example usages of nested patterns in MorphJ (Section 2). We then show real-world applications implemented using MorphJ (Section 3). In Section 4, we discuss the type-checking issues raised by nested patterns and introduce our type rules in an informal manner. We then formalize a core subset of MorphJ and present the type rules in Section 5. We continue with a discussion on related work in Section 6 and conclude with our grand-scheme view of the evolution of programming languages and the role of MorphJ and other related mechanisms (Section 7).

2. Reflective Declarations with Nested Patterns

MorphJ extends the reflective declaration blocks of MJ [14] with nested patterns. A nested pattern has the same syntactic form as a primary pattern, but is preceded by the keywords "`some`" (for a *positive* nested pattern) or "`no`" (for a *negative* nested pattern). Like primary patterns, nested patterns can only reflect over concrete types, or type variables of the generic class. A nested pattern places a condition (nested condition) on each element matched by the primary pattern. An element must be matched by the primary pattern *and* satisfy all nested conditions to be a part of a reflective block's iteration range. We illustrate the use of nested patterns with the following examples.

2.1 Negative Nested Pattern

A negative nested pattern exerts a condition that is only satisfied if there is *nothing* in the range of elements matched by the pattern.

[1] This MJ class only defines getter methods for non-`private` fields: the semantics of pattern "`F f`" without modifier specification is that it matches all non-`private` fields. This is a limitation with the subclassing-as-extension based approach that MJ adopts.

[2] MorphJ is effectively MJ 2.0. The change of name was decided to avoid ambiguity in our efforts to release the software for wide use—"MJ" is an overloaded term for indexing and search purposes. Nevertheless, the change is also convenient in the context of this paper as it enables us to refer to old capabilities by name.

Negative nested patterns allow us to implement a modularly type safe `AddGetter<X>`:

```
1 class AddGetter<X> extends X {
2   <F>[f]for( F f : X.fields ; no get#f() : X.methods )
3   F get#f() { return super.f; }
4 }
```

The nested pattern condition on line 2 is only satisfied by those fields `f` of `X` for which there is no method `get#f()` in `X`. (The missing return type in the nested pattern is a MorphJ shorthand for matching both `void` and non-`void` return types.) Observe that this `AddGetter<X>` class will not introduce ill-typed code for *any* `X`. Potentially conflicting method declarations are prevented by the negative nested pattern. A field `f` for which a method `get#f()` already exists in `X` does not satisfy the nested condition, and thus is not in the iteration range of the reflective block.

2.2 Positive Nested Pattern

A positive nested pattern exerts a condition that is only satisfied if there is *some* element in the range matched by the pattern. A positive nested pattern allows us to define the `Pair<X,Y>` class discussed in the Introduction as follows:

```
1  public class Pair<X,Y> {
2     X x; Y y;
3     public Pair(X x, Y y) { this.x = x; this.y = y; }
4
5     <RX extends Object, RY extends Object, A*>[m]
6     for(   public RX m(A) : X.methods ;
7       some public RY m(A) : Y.methods )
8     public Pair<RX,RY> m(A args) {
9        return new Pair<RX,RY>(x.m(args), y.m(args));
10    }
11 }
```

Methods of `Pair<X,Y>` are defined using the reflective block on lines 5-10. The primary pattern on line 6 matches all non-`void` and non-primitive methods of `X`. For each such method, the positive nested condition on line 7 is only satisfied if a method with the same name and argument types also exists in `Y`. Thus, the primary and nested patterns in this class find precisely all methods that `X` and `Y` share in name and argument types. For each such method, `Pair<X,Y>` declares a method with the same name and argument types, and a body that invokes the corresponding method of `X` and `Y`. The return type is another `Pair`, constructed from the return values of the invocations. Following the same pattern, the class can be enhanced to also handle methods returning primitive types or `void`.

2.3 More Features: `if`, `errorif`

Nested patterns enable other powerful language features. The reflective declarations we have seen so far have been iteration-based: a piece of code is declared for each element in the iteration range. MorphJ also supports condition-based reflective declarations and statements. An example (from an application discussed in detail in Section 3) illustrates the usage of pure conditions in reflective declarations:

```
<R> if ( no public R restore() : X.methods )
public void restore() { ... }
```

The above reflective declaration block consists of a statically exerted condition, specified by the pattern following the `if` keyword. If the pattern condition is satisfied, the code following the condition is declared. Thus, method `void restore()` is only declared if a method `restore()`, with any non-`void` return type, does not already exist in `X`.

Another useful feature is introduced by the `errorif` keyword. `errorif` acts as a type-assertion, allowing the programmer to express facts that he/she knows are true about a type parameter. For instance, the programmer can express a mixin class that is only applicable to non-conflicting classes:

```
1 class SizeMixin<X> extends X {
2   <F> errorif ( some F size : X.fields )
3   int size = 0;
4 }
```

In this case, the programmer wants to assert that, if the parameter type `C` already contains a field named `size`, this is not an error in the definition of `SizeMixin` but in the instantiation `SizeMixin<C>`. Thus, the `errorif` construct serves as a typical type-cast: it is both an assumption that the type system can use (i.e., when checking `SizeMixin<X>` it can be assumed that `X` has no `size` field) and at the same time a type obligation for a later type-checking stage. Unlike, however, traditional type-casts that turn a static type check into a run-time type check, an `errorif` turns a modular type check into a non-modular (type-instantiation time), but still static, type check.

2.4 Semantics of Nested Patterns

Similarly to primary patterns, a nested pattern introduces an iteration. However, nested patterns are only used to return a true/false decision. For instance, in class `Pair<X,Y>`, the nested pattern iterates over all the methods in `Y` matching the pattern, but the iteration only serves to verify whether a matching method exists, not to produce different code for each matching method. Furthermore, multiple nested patterns are all nested at the same level, forming a conjunction of their conditions.

Nested patterns may use pattern variables that are not bound by any primary pattern. However, there are restrictions as to how variables bound only by nested patterns can be used in code introduced by the reflective block (i.e., the reflective declaration). Pattern variables bound by only negative nested patterns cannot be used in the reflective declaration at all. For instance, the pattern "`no public R restore()`" above bound type variable `R`. However, `R` only appears in a negative nested pattern, and thus cannot be used in the declaration of `restore()`. Intuitively, a variable in a negative nested pattern is never bound to any concrete type/identifier—no match can exist for the negative nested condition to be satisfied. Clearly, an unbound variable cannot be used in declarations.

Pattern variables that are bound by positive nested patterns, however, can be used in the reflective declaration, if we can determine that exactly one element can be matched by the nested pattern. This is the case only if all uniquely identifying parts of the nested pattern use either constants, or pattern variables bound by the primary pattern. The uniquely identifying parts of a method pattern are its name and argument types, and the uniquely identifying part of a field pattern is its name. For example, in `Pair<X,Y>`, the positive nested pattern "`some RY m(A) : Y.methods`" uses `m` and `A` in its uniquely identifying parts. Both pattern variables are bound by the primary pattern. Thus, we can use `RY` in the reflective declaration, even though it only appears in the nested pattern.

It may not be immediately obvious how `if` and `errorif` relate to nested patterns. However, the type system machinery that enables `if` and `errorif` is precisely that of nested patterns. The patterns used as `if` and `errorif` conditions are regular nested patterns (with `some` and `no`) with the same semantics and conditions (e.g., limitations on when bound variables can appear in a reflective declaration). Indeed, even our actual implementation of the static `if` statement translates it intermediately into a static `for` loop with a special "unit" value for the primary pattern condition.

We do not allow the nesting of primary patterns—i.e., it is not legal to have nested static `for` loops. However, `if` and `errorif` declarations and statements can be freely nested within the scopes of one another, or within the scope of a static `for` loop.

3. Real-World Applications

We next show two real world applications, re-implemented concisely and safely with nested patterns.

3.1 DSTM2

DSTM2 [10] is a Java library implementing object-based software transactional memory. It provides a number of "transactional factories" that take as input a sequential class, and generate a transactional class. Each factory supports a different transactional policy. The strength of DSTM2 is in its flexibility. Users can mix and match policies for objects, or define new "factories" implementing their own transactional policies.

In order to automatically generate transactional classes, DSTM2 factory classes use a combination of Java reflection, bytecode engineering with BCEL [2], and anonymous class definitions. However, the information needed for these generations is purely static and structural. The authors of DSTM2 had to employ low-level runtime techniques because the Java language does not offer enough support for compile-time transformation of classes. MorphJ, however, is a good fit for this task.

In our re-implementation of DSTM2's factories and supporting classes, 1,107 (non-commented, non-blank) lines of Java code are replaced with 374 lines of MorphJ code. For example, we replaced DSTM2's `factory.shadow.RecoverableFactory<X>` and `factory.shadow.Adaptor<X>` with the MorphJ class `Recoverable<X>` in Figure 1.

```
1  @atomic public class Recoverable<class X> extends X {
2    // for each atomic field of X, declare a shadow field.
3    <F>[f]for(@atomic F f: X.fields; no shadow#f: X.fields)
4    F shadow#f;
5
6    // for each field of X, declare a getter.
7    <F>[f]for(@atomic F f: X.fields; no get#f(): X.methods)
8    public F get#f () {
9      Transaction me  = Thread.getTransaction();
10     Transaction other = null;
11     while (true) {
12       synchronized (this) {
13         other = openRead(me);
14         if (other == null) { return f; }
15       }
16       manager.resolveConflict(me, other);
17     }
18   }
19
20   // for each field of X, declare a setter
21   <F>[f]for(@atomic F f : X.fields;
22             no set#f(F) : X.methods)
23   public void set#f ( F val ) {
24     ... // code to open transaction.
25     f = val;
26     ... // code resolving conflict.
27   }
28
29   // create backup method
30   <R>if ( no R backup() : X.methods )
31   public void backup() {
32     <F>[f] for (@atomic F f : X.fields)
33     shadow#f = f;
34   }
35
36   // create recover method
37   <R>if ( no R recover() : X.methods )
38   public void recover() {
39     // restore field values from shadow fields.
40     <F>[f] for ( @atomic F f : X.fields )
41     f = shadow#f;
42   }
43 }
```

Figure 1. A recoverable transactional class in MorphJ.

For each field of `X`, `Recoverable<X>` creates a shadow field, as well as getter and setter methods that acquire a lock from a transactional manager first, perform the read or write, and then resolve conflicts before returning. Furthermore, it creates `backup()` and `restore()` methods to backup and restore fields to and from their shadow fields.[3]

The advantage of the MorphJ implementation is two-fold. First, `Recoverable<X>` is guaranteed to never declare conflicting declarations. For example, `shadow#f` is only declared if this field does not already exist in `X`, and `backup()` is only declared if such a method does not already exist in `X`. Implementations using reflection and bytecode engineering enjoy no such guarantees, and must instead rely on thorough testing to discover potential bugs.

Secondly, class `Recoverable<X>` is easier to write and understand. For example, the code for generating a `backup()` method in DSTM2's `RecoverableFactory<X>` is illustrated in Figure 2. We invite the reader to compare the `backup()` method declaration in Figure 1 (lines 29-33) to the code in Figure 2.

```
1  public class RecoverableFactory<X>
2                               extends BaseFactory<X> {
3    ...
4    public void createBackup() {
5      InstructionList il = new
6      InstructionList(); MethodGen method =
7        new MethodGen(ACC_PUBLIC, Type.VOID, Type.NO_ARGS,
8                      new String[] { }, "backup",
9                      className, il, _cp);
10
11     for (Property p : properties) {
12       InstructionHandle ih_0 =
13         il.append(_factory.createLoad(Type.OBJECT, 0));
14       il.append(_factory.createLoad(Type.OBJECT, 0));
15       il.append(_factory
16                 .createFieldAccess(className, p.name,
17                                    p.type,
18                                    Constants.GETFIELD));
19       il.append(_factory.
20                 .createFieldAccess(className,
21                                     p.name + "$",
22                                     p.type,
23                                     Constants.PUTFIELD));
24     }
25
26     InstructionHandle ih_24 =
27             il.append(_factory.createReturn(Type.VOID));
28     method.setMaxStack();
29     method.setMaxLocals();
30     _cg.addMethod(method.getMethod());
31     il.dispose();
32   }
33 }
```

Figure 2. DSTM2 code for creating a method `backup()`.

Interestingly, the predecessor of DSTM2 is a C# software transactional memory library called SXM [9]. It was re-implemented by Fähndrich, Carbin and Larus as the "quintessential example" of Compile-Time Reflection (CTR) [7]. However, CTR's safety guarantees only concern the validity of references, and not declaration conflicts. We give a more detailed exposition of CTR in Section 6.

3.2 Default Implementations for Interface Methods

Java ensures that a class cannot be declared to "implement" an interface unless it provides implementations for all of the interface's methods. This often results in very tedious code. For instance, it is common in code dealing with the Swing graphics library to implement an event-listener interface, yet provide empty implementa-

[3] The MorphJ implementation replicates the functionality of DSTM2 factory classes, but with low-level differences. For instance, a DSTM2 factory generates transactional classes from interface definitions. It parses the interface's method names that begin with "`get`" and "`set`" to determine the field names to generate. We believe this to be a design artifact due to the constraints of the Java language. We instead statically iterate over the `@atomic` fields of classes, and generate backup fields as needed.

tions for most of the interface methods because the application does not care about the corresponding events. In response to this need, there have been mechanisms proposed [11, 20] that allow the programmer to specify that he/she wants just a default implementation for all members of an interface that are not already implemented by a class. These past solutions introduced new keywords (or Java annotations) for this specific feature. They either have no guarantee for the well-typedness of generated code [11], or require extensions to the Java type system [20]. These changes to the underlying language are required to support just this *one feature*. In contrast, we can express these language extensions as a MorphJ generic class that is guaranteed to always produce well-typed code. Below is a slightly simplified version of the MorphJ solution to this problem. (For conciseness, we elide the declarations dealing with `void`- or primitive-type-returning methods, which roughly double the code.)

```
1  class DefaultImplementation<X,interface I> implements I {
2    X x;
3    DefaultImplementation(X x) { this.x = x; }
4
5    // for all methods in I, if the same method does
6    // not appear in X, provide default implementation.
7    <R extends Object,A*>[m]for( R m (A) : I.methods ;
8                                 no R m (A) : X.methods )
9    R m (A a) { return null; }
10
11   // for all methods in X that *do* correctly override
12   // methods in I, we need to copy them.
13   <R,A*>[m]for(   R m (A) : I.methods ;
14                some R m (A) : X.methods )
15   R m (A a) { return x.m(a); }
16
17   // for all methods in X, such that there is no method
18   // in I with the same name and arguments, copy method.
19   <R,A*>[m]for( R m (A) : X.methods;
20                no m (A) : I.methods)
21   R m (A a) { return x.m(a); }
22 }
```

Class `DefaultImplementation<X,I>` copies all methods of type `X` that either correctly implement methods in interface `I`, or are guaranteed to not conflict with methods in `I`. For methods in `I` that have no counterpart in `X`, a default implementation is provided. Methods in `X` that conflict with methods in `I` (same argument types, different return) are ignored. The above code demonstrates the power of nested patterns, both in terms of expressiveness, and in terms of type safety. The application naturally calls for different handling of methods in a type, based on the existence of methods in another type. Furthermore, these declarations are guaranteed to be unique, and their uniqueness is crucially based on nested patterns.

4. Type-Checking Nested Patterns

The complexity in type-checking pattern-based reflective declarations arises from the existence of pattern type and name variables. When a declaration is made using such variables, how can we check that for all concrete types and identifiers these variables could expand to, the declaration will always be unique? Similarly, how can we check that a reference made using a name variable always refers to an entity that is declared?

The key is to treat a declaration as a *range* of declared elements. (A declaration made without pattern variables has a one-element range.) Determining the uniqueness of two declarations then reduces to determining whether their ranges are *disjoint*. Similarly, a reference is also a range. Determining whether a reference is valid then reduces to determining *subsumption*: do all entities in the reference range have corresponding entities in the declaration range?

In this section, we introduce the techniques for checking reference validity and declaration uniqueness with nested patterns. We focus on declarations and references made by reflecting over type variables: reflecting over non-variable types is simply syntactic sugar for manually inlining the declarations. We further focus on the rules for type-checking methods—rules for fields are a trivial adaptation of those for methods.

4.1 Reference Validity

Let us take another look at class `LogMe<X>` from the Introduction:

```
1 class LogMe<class X> extends X {
2    <R,A*>[m] for ( public R m(A) : X.methods )
3    public R m (A a) {
4       R result = super.m(a);
5       System.out.println(result);
6       return result;
7    }
8 }
```

How do we know that the method invocation "`super.m(a)`" (line 4) is valid? Notice that the range of `m` (i.e., all the identifiers it could expand to) is exactly the names of methods matched by the primary pattern on line 2: all non-`void` methods of `X`. This range is certainly subsumed by the range of all methods declared for `X`. Thus, we know method `m` exists, no matter what `X` is. Furthermore, how do we know we are invoking `m` with the right arguments? The type of `a` is `A`: exactly the argument type `m` of `X` is expecting.

Things get a bit more complex when a name variable bound in one reflective block references a method declared in a different reflective block. Consider the following class, which logs the arguments of methods accepting strings, before calling `LogMe` to log the return value.

```
1 class LogStringArg<class Y> {
2   LogMe<Y> loggedY;
3
4   <T>[n] for ( public T n(String) : Y.methods )
5   public T n (String s) {
6       System.out.println("arg: " + s);
7       return loggedY.n(s);
8   }
9 }
```

How do we know that `loggedY.n(s)` (line 7) is a valid reference, when the methods of `loggedY` are defined in a different class and a different reflective block? The key is to determine that the range of `n` is subsumed by the range of methods in `LogMe<Y>`. This is to say that the range of `n`'s enclosing reflective block should be subsumed by the range of `LogMe<Y>`'s declaration reflective block. Observe that the declaration block of `LogMe<Y>` is defined over methods of `Y` (after substituting `Y` for `X`), as is the reflective block enclosing `n`. Secondly, the pattern for the declaration block of `LogMe<Y>` is more general than the pattern for the reflective block enclosing `n`: the former matches all non-`void` methods, and the latter matches all non-`void` methods taking exactly one `String` argument. Thus, any method that is matched by the reference reflective block's pattern is matched by the declaration reflective block's pattern, regardless of what `Y` is. Thus, there is always a method `n` in `LogMe<Y>`.

Whether a pattern is more general than another can be systematically determined by finding a *one-way unification* from the more general pattern to the more restricted one. In a one-way unification, only pattern variables declared for the more general pattern are used as unification variables. All other pattern variables are considered constants. In this example, we can unify "`public R m(A)`" to "`public T n(String)`" using the mapping $\{$`R`$\mapsto$`T`, `m`$\mapsto$`n`, `A`$\mapsto\{$`String`$\}\}$.

We also use this unification mapping in determining whether `n` is invoked with the right argument types. We apply the mapping to the method declaration in `LogMe<Y>`, and get the declared signature "`public T n(String)`". Since `s` has the type `String`, the invocation

is clearly correct. Furthermore, we can check that the result of the invocation is of type T, which is precisely the expected return type of the method enclosing "loggedY.n(s)".

For the case of nested patterns, consider the following class:

```
1 class VoidPair<X,Y> {
2   X x; Y y; ...// constructor to initialize x and y.
3
4   <A*>[m]for (  public void m(A) : X.methods ;
5             some public void m(A) : Y.methods )
6   public void m (A a) {   x.m(a);  y.m(a);   }
7 }
```

VoidPair<X,Y> declares a method for every void method that X and Y share in name and argument types, and invokes that method on x and y. Using the reference rules described previously, we know that x.m(a) is a valid reference. Furthermore, because the pattern variables used in the positive nested pattern on line 5 are all bound by the primary pattern, we know that if the nested condition is satisfied, there is exactly one element in the range of the nested pattern, so the call y.m(a) is unambiguous. Since the types also match, y.m(a) is a valid reference, as well.

Let us now consider the general case of a reference made in one reflective block, to declarations made in another reflective block, when both blocks have nested patterns. Let R_d and R_r denote the ranges for the reflective blocks of the declaration and the reference, respectively. There are two sufficient conditions to determine that R_r is subsumed by R_d. First, the primary range of R_r must be subsumed by the primary range of R_d. Second, for all methods that are in the primary range of R_r (and thus also in the primary range of R_d), if the method satisfies the nested conditions of R_r, it should also satisfy the nested conditions of R_d. That is to say, the nested conditions of R_r should be stronger, and imply the nested conditions of R_d.

Determining that one nested condition implies another can be reduced to single range subsumption. Let $\langle N_r,T_r\rangle$ denote the range of a nested pattern N_r matching over the methods of type T_r. Let $\langle N_d,T_d\rangle$ be similarly interpreted. Let $+$ prefix a positive nested condition, and $-$ prefix a negative nested condition. We have two ways of determining that one condition implies another:

- $+\langle N_r,T_r\rangle$ implies $+\langle N_d,T_d\rangle$ if $\langle N_d,T_d\rangle$ subsumes $\langle N_r,T_r\rangle$.
- $-\langle N_r,T_r\rangle$ implies $-\langle N_d,T_d\rangle$ if $\langle N_r,T_r\rangle$ subsumes $\langle N_d,T_d\rangle$.

Intuitively, $+\langle N_r,T_r\rangle$ is satisfied if there is at least one element in $\langle N_r,T_r\rangle$. Then there is certainly at least one element in a larger range, as well. Thus, $+\langle N_d,T_d\rangle$ should be satisfied. Similar reasoning applies for the implication between two negative conditions.

To be more concrete, consider the following class:

```
8  class CallVoidsWithString<T,S> {
9    VoidPair<T,S> voidPair;
10   ... // constructor to initialize voidPair
11   [n]for (  public void n(String) : T.methods ;
12        some public void n(String) : S.methods )
13   public void n (String s) { voidPair.n(s); }
14 }
```

For every void method taking one String argument that T and S have in common, CallVoidsWithString<T,S> declares a method with the same signature, and invokes a method with the same name on voidPair, of type VoidPair<T,S>. This reference is valid if the range of the reflective block on lines 11-12 is subsumed by the range of the declaration reflective block (lines 4-5 in the definition of VoidPair).

The range of primary pattern on line 11 is subsumed by the range of declaration's primary pattern (line 4), by the one-way unification mapping $\{\texttt{m}\mapsto\texttt{n}, \texttt{A}\mapsto\{\texttt{String}\}\}$.

To check whether the nested pattern on line 5 subsumes the nested pattern on line 12, note that we first apply the unification mappings obtained from unifying the primary patterns—we only want to determine this subsumption relationship for those methods that lie in the range of both primary patterns. In our example, after applying the unification mapping to the positive nested pattern on line 5 (and also substituting S for Y), we have "public void n(String) : S.methods". This clearly subsumes "public void n(String) : S.methods" on line 12.

These two conditions guarantee us that reference voidPair.n(s) is always a valid one. It is easy to check that this is indeed the case.

The above approach generalizes to an arbitrary number of nested conditions: each nested condition in the declaration range must be implied by at least one nested condition in the reference range. A range with no nested patterns is equivalent to a range with a positive nested pattern that subsumes everything, or a negative nested pattern that is subsumed by everything. The case where there are only nested patterns (i.e., if and errorif statements) can be reduced to a range with a special primary pattern value that subsumes only itself and is subsumed only by itself.

4.2 Uniqueness of Declarations

We use range disjointness to check whether two declarations are unique. In the case of method declarations, uniqueness means two methods within the same class (including inherited methods) cannot have the same name and argument types.[4]

4.2.1 Internally Well-defined Range

A simple property to establish is that declarations introduced by the same reflective block do not conflict. Consider the following class:

```
1 class CopyMethods<X> {
2    <R,A*>[m] for( R m (A) : X.methods ; nestedConds )
3    R m (A a) { ... }
4 }
```

CopyMethods<X>'s methods are declared within one reflective block. The iteration range of this block comprises all non-void methods of X that also satisfy arbitrary nested conditions, *nestedConds*. For each of these methods, a method with identical signature is declared for CopyMethods<X>.

How do we guarantee that, given any X, the method declarations within this block are always unique? Observe that X can only be instantiated with a well-formed type (the base case being Object), and all well-formed types have unique method declarations. Thus, if the declaration block merely copies the name and argument types of methods from a well-formed type, the methods declared by this block are guaranteed to be unique, as well.

4.2.2 Uniqueness Across Ranges

When one or both methods are defined using reflective iterations, their uniqueness means that the *range* of their ⟨name, argument types⟩ tuple cannot overlap. This can be determined by a *two-way unification* of the two declarations. In a two-way unification, pattern variables from *both* reflective blocks are unification variables.

Let us start with a simple example. Consider the following class:

```
1 class DisjointDecs<X> {
2   <R>[m] for(R m (int) : X.methods; nestedConds1 )
3   R m (int i)  ...
4
5   <S>[n] for(S n (int) : X.methods; nestedConds2 )
6   S n (int i, String s)  ...
7 }
```

[4] In Java, methods in a subclass are allowed to override their counterparts in the superclass with co-variant return types. This involves a relaxation of the rules we describe in this section: return types in the subclass are allowed to be subtypes of their counterparts in the superclass.

It is easy to see that the declarations on lines 3 and 6 cannot overlap for any X. There is no unification to make the two signatures have the same ⟨name, argument types⟩ tuple, because there is simply no way to unify {int} with {int,String}.

When two method signatures do unify, there may be overlap in the declarations. However, if we can prove that overlapping elements are infeasible, then the declarations are still unique. An overlap is infeasible if the unification mapping producing the overlap, when applied to the primary and nested patterns, produces mutually exclusive conditions. Note that a non-empty primary pattern range states a condition, as well—it is a positive condition that says *some* element exists in this range.

Consider the following class:

```
1 class StillUnique<X> {
2   <A1>[m]for( String m (A1) : X.methods ; nestedConds1 )
3   void m (A1 a) { ... }
4
5   <A2>[n]for( int n (A2)    : X.methods ; nestedConds2 )
6   void n (A2 a) { ... }
7 }
```

The declared signatures on lines 3 and 6 unify with the mapping {m↦n, A1↦A2}. Applying this mapping to the primary patterns on lines 2 and 5, we get "`String n (A2) : X.methods`", and "`int n (A2) : X.methods`". Methods matched by these patterns can cause conflicting declarations. However, having at least one method in both of these ranges means that there need to be two methods in X with the same name and argument types, but different return types. This directly contradicts the fact that X is a well-formed type. Thus, this unification mapping produces mutually exclusive conditions between the two primary pattern conditions, and there are no elements that would make the mapping possible. These declarations are thus still disjoint.

There are two ways to determine whether two conditions are mutually exclusive. Using the same notation as before,

- $+\langle P_n,T_n\rangle$ and $+\langle P'_n,S_n\rangle$ are mutually exclusive if T_n is a subtype of S_n, and P_n, P'_n have unifying method name and argument types, but different return types.
- $+\langle P_n,T_n\rangle$ and $-\langle P'_n,S_n\rangle$ are mutually exclusive if $\langle P'_n,S_n\rangle$ subsumes $\langle P_n,T_n\rangle$

We apply these rules on all pairs of conditions. A single mutual exclusive pair guarantees the disjointness of ranges. We applied the first rule to prove that `StillUnique<X>` contains unique method declarations. The following example demonstrates an application of the second rule:

```
1 public class UnionOfStatic<X,Y> {
2   <A*>[m] for( static void m (A) : X.methods; nestedCond)
3   public static void m(A args) { X.m(args); }
4
5   <B*>[n] for( static void n (B)      : Y.methods ;
6              no static void n (int, B) : X.methods )
7   public static int n(int count, B args) {
8     for (int i = 0; i < count; i++) Y.n(args);
9     return count;
10  }
11 }
```

The two method declarations on lines 3 and 7 have signatures that can be unified with the mapping { A↦{int,B}, m↦n}. Applying this substitution to the primary pattern on line 2 yields "`static void n(int,B) : X.methods`". Having a method in the range of this pattern directly contradicts the condition of the negative nested pattern on line 6, which states there should be no methods in the range of "`static void n(int,B) : X.methods`". Thus, the two method declarations are unique for all X and Y.

4.2.3 Generalizations and Boundary Conditions

We have so far neglected to state the rules for when one of the names used for reference or declaration is a constant name. The range of reference and declaration with such a name contains a single element. Thus, it can always be subsumed by a range with a variable name, but it can never be disjoint from a range with a variable name.

We have also glossed over some details in the unification. In addition to unifying the pattern variables, there needs to be an additional check on type bounds. For example, a pattern variable `A extends Number` cannot be unified with variable `B extends java.io.File`, because they can never match the same types—there is no type that is a subtype of both `Number` and `java.io.File`. This detail is rigorously defined in the rules presented in Section 5.

5. Formalization

We formalize MorphJ's type system with a simplified formalism, FMJ, based on FGJ [15]. Due to space limitations, we present only the type rules most relevant to checking declarations and expressions enclosed by nested patterns. Interested readers may consult our technical report [12] for the full text of type rules and soundness proofs.

5.1 Syntax

The syntax of FMJ is presented in Figure 3. We adopt many of the notational conventions of FGJ: C,D denote constant class names; X,Y,Z denote type variables; N,P,Q,R denote non-variable types; S,T,U,V,W denote types; f denotes field names; m denotes non-variable method names; x,y denote argument names. Notations new to FMJ are: η denotes a variable method name; n denotes either variable or non-variable names; o denotes a nested pattern condition operator (either + or -) for the keywords `some` or `no`, respectively.

T	::=	X | N
N	::=	C<$\overline{\texttt{T}}$>
CL	::=	class C<$\overline{\texttt{X}}$◁$\overline{\texttt{N}}$>◁ N {$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; $\overline{\texttt{M}}$}
	|	class C<$\overline{\texttt{X}}$◁$\overline{\texttt{N}}$>◁ T {$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; $\overline{\mathfrak{M}}$}
M	::=	T m ($\overline{\texttt{T}}$ $\overline{\texttt{x}}$) {↑e;}
$\mathfrak{M}$	::=	<$\overline{\texttt{Y}}$◁$\overline{\texttt{P}}$>for($\mathbb{M}_p$;$o\mathbb{M}_n$) U η ($\overline{\texttt{U}}$ $\overline{\texttt{x}}$) {↑e;}
o	::=	+ | -
$\mathbb{M}$	::=	V η ($\overline{\texttt{V}}$):X.methods
e	::=	x | e.f | e. n ($\overline{\texttt{e}}$) | new C<$\overline{\texttt{T}}$>($\overline{\texttt{e}}$)
n	::=	m | η

Figure 3. Syntax

We use the shorthand $\overline{\texttt{T}}$ for a sequence of types $\texttt{T}_0,\texttt{T}_1,\ldots,\texttt{T}_n$, and $\overline{\texttt{x}}$ for a sequence of unique variables $\texttt{x}_0,\texttt{x}_1,\ldots,\texttt{x}_n$. We use $\bullet$ to denote an empty sequence. We use : for sequence concatenation, e.g. $\overline{\texttt{S}}:\overline{\texttt{T}}$ is a sequence that begins with $\overline{\texttt{S}}$, followed by $\overline{\texttt{T}}$. We use $\in$ to mean "is a member of a sequence" (in addition to set membership). We use … for values of no particular significance to a rule. ◁ and ↑ are shorthands for the keywords `extends` and `return`, respectively. Note that all classes must declare a superclass, which can be `Object`.

FMJ formalizes some core Mystique features that are representative of our approach. One simplification is that we allow only one nested pattern per reflective block. This does not change the essence of our type system, since we can emulate multiple nested patterns using one nested pattern that reflects over an intermediate type, defined itself using reflective declarations. (I.e., we can simulate i nested patterns with $i-1$ intermediate types with one

nested pattern each.) Another simplification is that we do not allow a nested pattern to use any pattern type or name variables not bound by its primary pattern. We also do not formalize reflecting over a statically known type, or using a constant name in reflective patterns. These are decidedly less interesting cases from a typing perspective. The zero or more length type vectors `T*` are also not formalized. These type vectors are a mere matching convenience. Thus, safety issues regarding their use are covered by non-vector types. We do not formalize reflectively declared fields—their type-checking is a strict adaptation of the techniques for checking methods. Lastly, static name prefixes, casting expressions and polymorphic methods are not formalized.

FGJ does not support method overloading, and FMJ inherits this restriction. Thus, a method name alone uniquely identifies a method definition. Since we allow no fresh name variables in nested patterns, there can be only one name variable in a reflective block. We use η for this name variable, and a reflective definition must also use this same name variable. This results in a small simplification over the informal rules in Section 4 but leaves their essence intact.

A program in FMJ is an $(\mathtt{e}, CT)$ pair, where `e` is an FMJ expression, and CT is the class table. We place some conditions on *CT*: every class declaration `class C...` has an entry in *CT*; `Object` is not in *CT*. The subtyping relation derived from *CT* must be acyclic, and the sequence of ancestors of every instantiation type is finite. (The last two properties can be checked with the algorithm of [1] in the presence of mixins.)

5.2 Typing Judgments

There are three environments used in our typing judgments:

- Δ: Type environment. Maps type variables to their upper bounds.
- Γ: Variable environment. Maps variables (e.g., `x`) to their types.
- Λ: Reflective iteration environment. Λ has the form $\langle R_p, oR_n\rangle$, where R_p is the primary pattern, and oR_n the nested pattern. o can be + or -.
  - R_p has the form $(\mathtt{T}_1,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)$. $\mathtt{T}_1$ is the type over whose methods R_p iterates. We call it the *reflective type* of R_p. $\overline{\mathtt{Y}}$ are pattern type variables, bounded by $\overline{\mathtt{P}}$, and $\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0$ the method pattern.
  - R_n has a similar form: $(\mathtt{T}_2,\ \overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0)$. However, note the lack of pattern type variables, due to the (formalism-only) simplification that the nested pattern not use pattern type variables not already bound in the primary pattern.

  There is no nesting of reflective loops. Thus, Λ contains at most one $\langle R_p, oR_n\rangle$ tuple.

We use the $\mapsto$ symbol for mappings in the environments. For example, $\Delta{=}\mathtt{X}{\mapsto}\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ means that $\Delta(\mathtt{X}){=}\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$. Every type variable must be bounded by a non-variable type. The function $bound_\Delta(\mathtt{T})$ returns the upper bound of type `T` in Δ. $bound_\Delta(\mathtt{N}){=}\mathtt{N}$, if `N` is not a type variable. And $bound_\Delta(\mathtt{X}){=}bound_\Delta(\mathtt{S})$, where $\Delta(\mathtt{X}){=}\mathtt{S}$.

In order to keep our type rules manageable, we make two simplifying assumptions. To avoid burdening our rules with renamings, we assume that pattern type variables have globally unique names (i.e., are distinct from pattern type variables in other reflective environments, as well as from non-pattern type variables). We also assume that all pattern type variables introduced by a reflective block are bound (i.e., used) in the corresponding primary pattern. Checking this property is easy and purely syntactic.

The core of our type system is in determining reflective range subsumption and disjointness. Thus, we begin our discussion with a detailed explanation of the rules for subsumption and disjointness.

5.2.1 Subsumption and Disjointness

The range of a reflective environment, $\langle R_p, oR_n\rangle$, comprises methods in the primary range R_p, that also satisfy the nested condition oR_n. The nested condition $+R_n$ (or $-R_n$) is satisfied if there is at least one method (or no method, resp.) in the range of R_n. We call ranges of R_p and R_n *single ranges*. In this section, we explain the rules for determining the following three relations:

- $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda\sqsubseteq_\Lambda\Lambda'$. Range of Λ is subsumed by the range of Λ', under the assumptions of type environment Δ and the unifying type substitutions of $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$.
- $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1\sqsubseteq_R R_2$. Single range R_1 is subsumed by single range R_2, under the assumptions of Δ and the unifying type substitutions of $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$.
- $\Delta\vdash \mathit{disjoint}(\Lambda, \Lambda')$. The range of Λ and Λ' are disjoint under the assumptions of Δ.

Single range subsumption. In determining the subsumption between two reflective environments, we must first see how subsumption is determined between two single ranges. Rule SB-R (Figure 4) states that first, the reflective type of the larger range, R_2, should be a subtype of R_1's reflective type. Secondly, R_2's pattern should be more general than R_1's pattern. This means that a *one-way* unification exists from the pattern of R_2 to the pattern of R_1, where only the pattern type variables in R_2 are considered variables in the unification process. $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$ are the substitutions that satisfy such one-way unification. Unification is defined by two relations:

- $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \mathit{unify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{V}_0{:}\overline{\mathtt{V}})$. Rule UNI (Figure 6) describes a standard unification condition with a twist: unifying substitutions (for pattern type variables) must respect the subtyping bounds of the type variables. For example, the substitution $[\mathtt{Y}/\mathtt{Object}]$, where $\Delta\vdash\mathtt{Y}\mathtt{<:Number}$, does *not* unify `Y` and `Object`, because the bound of `Y` is tighter than `Object`.
- $\Delta\vdash\mathtt{T}\prec:_{\overline{\mathtt{Z}}}\mathtt{S}$ (Figure 6) indicates that type `T` is a valid substitution of `S`, i.e., it obeys the bound of `S`, using $\overline{\mathtt{Z}}$ as pattern type variables.

Reflective (nested) range subsumption. SB-Λ (Figure 4) defines the conditions for the range of reflective environment $\Lambda{=}\langle R_p, oR_n\rangle$ to be subsumed by the range of $\Lambda'{=}\langle R'_p, o'R'_n\rangle$. These conditions reflect precisely the informal rules of Section 4. First, regardless of nested patterns, the primary range of Λ should at least be subsumed by the primary range of Λ'. Secondly, for every method in R_p that satisfies the nested pattern oR_n, the corresponding method in R'_p should satisfy the nested pattern $o'R'_n$. There are a couple of ways to guarantee oR_n *implies* $o'R'_n$. If $+R_n$ is true, and R_n is subsumed by R'_n, then $+R'_n$ is also true. This condition is expressed by $\Delta;\bullet\vdash R_n\sqsubseteq_R[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n$, if $o{=}o'{=}+$. We apply the unifying type substitutions for the primary ranges to the nested range R'_n: in order to properly compare the ranges of R_n and R'_n, we need to restrict R'_n to what it can be for the methods that are matched by both R_p and R'_p. Note that we are using an empty sequence of type substitutions ($\bullet$) in determining that R_n is subsumed by $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n$. This is because nested patterns do not have pattern type variables of their own, and pattern type variables from the primary pattern are treated as constants in the nested patterns. Similarly, if $-R_n$ is true, and R_n subsumes R'_n, $-R'_n$ is also true.

Reflective range disjointness. Disjointness of reflective ranges is defined by rules DS-Λ1 and DS-Λ2. DS-Λ1 specifies the conditions for disjointness when Λ and Λ' reflect over types from the same subtyping hierarchy. In this case, Λ and Λ' are disjoint if their primary ranges are disjoint. However, if the two primary ranges *do* have overlap, (i.e., $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Z}}]\vdash \mathit{unify}(\mathtt{U}_0{:}\overline{\mathtt{U}}, \mathtt{U}'_0{:}\overline{\mathtt{U}'})$: a *two-way* unification exists between the primary ranges) it is still possible for Λ

Reflective range subsumption:

$$\frac{\Lambda{=}\langle R_p, oR_n\rangle \quad \Lambda'{=}\langle R'_p, o'R'_n\rangle \quad R'_p{=}(\mathtt{T}'_p,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \quad \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_p \sqsubseteq_R R'_p \quad \Delta;\bullet\vdash R_n \sqsubseteq_R [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n \text{ if } o{=}o'{=}+ \quad \Delta;\bullet\vdash [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n \sqsubseteq_R R_n \text{ if } o{=}o'{=}-}{\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \Lambda \sqsubseteq_\Lambda \Lambda'} \quad \text{(SB-}\Lambda\text{)}$$

Single range subsumption:

$$\frac{R_1{=}(\mathtt{T}_1,\ \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0) \quad R_2{=}(\mathtt{T}_2,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \quad \Delta\vdash \mathtt{T}_2\mathtt{<:}\mathtt{T}_1 \quad \Delta'{=}\Delta,\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{Q}},\overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}} \quad \Delta';[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \mathit{unify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{V}_0{:}\overline{\mathtt{V}})}{\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1 \sqsubseteq_R R_2} \quad \text{(SB-}R\text{)}$$

Reflective range disjointness:

$$\frac{\begin{array}{c}\Lambda{=}\langle R_p, oR_n\rangle \quad \Lambda'{=}\langle R'_p, o'R'_n\rangle \quad R_p{=}(\mathtt{T}_p,\ \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0) \quad R'_p{=}(\mathtt{T}'_p,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}'\mathtt{\to}\mathtt{U}'_0)\\ \Delta\vdash \mathtt{T}_p\mathtt{<:}\mathtt{T}'_p \text{ or } \Delta\vdash \mathtt{T}'_p\mathtt{<:}\mathtt{T}_p \qquad \Delta'{=}\Delta,\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{Q}},\overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}} \qquad \overline{\mathtt{Z}}{=}\overline{\mathtt{X}},\overline{\mathtt{Y}}\\ \text{for all } \overline{\mathtt{W}},\ \Delta';[\overline{\mathtt{W}}/\overline{\mathtt{Z}}]\vdash \mathit{unify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{U}'_0{:}\overline{\mathtt{U}'}) \text{ implies } \begin{cases}\Delta';\bullet\vdash [\overline{\mathtt{W}}/\overline{\mathtt{Z}}]R_n \sqsubseteq_R [\overline{\mathtt{W}}/\overline{\mathtt{Z}}]R'_n & \text{if } o=+, o'=-\\ \Delta';\bullet\vdash [\overline{\mathtt{W}}/\overline{\mathtt{Z}}]R'_n \sqsubseteq_R [\overline{\mathtt{W}}/\overline{\mathtt{Z}}]R_n & \text{if } o=-, o'=+\end{cases}\end{array}}{\Delta\vdash \mathit{disjoint}(\Lambda,\Lambda')} \quad \text{(DS-}\Lambda\text{1)}$$

$$\frac{\Lambda{=}\langle R_p, oR_n\rangle \quad \Lambda'{=}\langle R'_p, o'R'_n\rangle \quad R_p{=}(\mathtt{T}_p,\ \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0) \quad R'_p{=}(\mathtt{T}'_p,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}'\mathtt{\to}\mathtt{U}'_0) \quad \Delta'{=}\Delta,\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{Q}},\overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}} \quad \begin{cases}\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R'_p \sqsubseteq_R R_n & o=- \quad \text{or}\\ \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_p \sqsubseteq_R R'_n & o'=-\end{cases}}{\Delta\vdash \mathit{disjoint}(\Lambda,\Lambda')} \quad \text{(DS-}\Lambda\text{2)}$$

Figure 4. Range subsumption and disjointness rules.

Method type lookup:

$$\frac{\Lambda{=}\langle R_p, oR_n\rangle \quad R_p{=}(\mathtt{X},\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)}{\Delta;\Lambda\vdash \mathit{mtype}(\eta,\ \mathtt{X}){=}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0} \quad \text{(MT-VAR-R1)} \qquad \frac{\Lambda{=}\langle R_p, {+}R_n\rangle \quad R_n{=}(\mathtt{X},\ \overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)}{\Delta;\Lambda\vdash \mathit{mtype}(\eta,\ \mathtt{X}){=}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0} \quad \text{(MT-VAR-R2)}$$

$$\frac{\Lambda{=}\langle R_p, oR_n\rangle \quad R_p{=}(\mathtt{T},\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \quad \Delta;\Lambda\vdash \mathit{mtype}(\eta,\ \mathit{bound}_\Delta(\mathtt{X})){=}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0}{\Delta;\Lambda\vdash \mathit{mtype}(\eta,\ \mathtt{X}){=}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0} \quad \text{(MT-VAR-S)}$$

$$\frac{\begin{array}{c}CT(\mathtt{C}){=}\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd T}\ \{\ldots\ \overline{\mathfrak{M}}\} \quad \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>for}(\mathbb{M}_p\mathtt{;}o\mathbb{M}_f)\ \mathtt{S}_0\ \eta\ (\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e;}\} \in \overline{\mathfrak{M}}\\ \mathbb{M}_p{=}\mathtt{U}_0\ \eta\ (\overline{\mathtt{U}}){:}\mathtt{X}_i\mathtt{.methods} \quad \mathbb{M}_f{=}\mathtt{V}_0\ \eta\ (\overline{\mathtt{V}}){:}\mathtt{X}_j\mathtt{.methods}\\ R_p{=}(\mathtt{T}_i,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)) \quad R_n{=}(\mathtt{T}_j,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0)) \quad \Lambda_d{=}\langle R_p, oR_n\rangle \quad \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \Lambda \sqsubseteq_\Lambda \Lambda_d\end{array}}{\Delta;\Lambda\vdash \mathit{mtype}(\eta,\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}){=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}\mathtt{\to}\mathtt{S}_0)} \quad \text{(MT-CLASS-R)}$$

$$\frac{\begin{array}{ll}& CT(\mathtt{C}){=}\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{><T}\ \{\ldots\ \overline{\mathfrak{M}}\}\\ \text{for all} & \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>for}(\mathbb{M}_p\mathtt{;}o\mathbb{M}_f)\ \mathtt{S}_0\ \eta\ (\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e;}\} \in \overline{\mathfrak{M}}\\ & \mathbb{M}_p{=}\mathtt{U}_0\ \eta\ (\overline{\mathtt{U}}){:}\mathtt{X}_i\mathtt{.methods} \quad \mathbb{M}_f{=}\mathtt{V}_0\ \eta\ (\overline{\mathtt{V}}){:}\mathtt{X}_j\mathtt{.methods}\\ & R_p{=}(\mathtt{T}_i,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)) \quad R_n{=}(\mathtt{T}_i,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0)) \quad \Lambda_d{=}\langle R_p, oR_n\rangle\\ \text{implies} & \Delta\vdash \mathit{disjoint}(\Lambda,\Lambda_d)\end{array}}{\Delta;\Lambda\vdash \mathit{mtype}(\eta,\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}){=}\mathit{mtype}(\eta,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T})} \quad \text{(MT-SUPER-R)}$$

Figure 5. Method type lookup.

and Λ' to be disjoint if we can establish that for the methods that fall into the overlap, the nested patterns cannot be satisfied simultaneously. There are two ways to establish the exclusivity of two nested patterns. First, if $+R_n$ is true, and R_n is subsumed by R'_n, then $-R'_n$ cannot possibly by true. Similarly, if $+R'_n$ is true, and R'_n is subsumed by R_n, then $-R_n$ cannot be true.

DS-Λ2 specifies a different condition for disjointness: if the primary range of Λ, R_p, can be subsumed by the nested range of Λ', R'_n, and the nested pattern is negative (i.e., $-R'_n$), then it is guaranteed that Λ and Λ' have disjoint names. The reason is that any method matched by the primary range R'_p is guaranteed to *not* satisfy the nested pattern $-R_n$, thus the two nested ranges are disjoint. Similarly, if R_p is subsumed by R_n, and $-R'_n$ is the nested pattern condition, disjointness is also established.

These rules reflect very closely the informal rules of Section 4 modulo the small differences in the formalism mentioned in Section 5.1: we do not need to distinguish between declarations and primary patterns in the formalism, as the uniqueness of entities in the primary pattern implies (through name uniqueness, since there is no overloading) the uniqueness of declared entities.

5.2.2 Valid Method Invocation

The rest of the typing rules add machinery to standard FGJ type checking to express checks using range subsumption and disjointness. For instance, method invocation rules rely on method lookup rules, *mtype*, to determine the correct method type. We have shown in Figure 5 the *mtype* rules pertaining to looking up methods refered to using name variables. Please consult our technical report for the full set of rules.

MT-VAR-R1 and MT-VAR-R2 say that the type of method with a variable name η in a type X, where X is either the reflective type for the primary pattern or the reflective type of a *positive* nested pattern, is exactly the type specified by the primary (or nested, respectively) pattern. Otherwise, if X is a type variable, then we must look for the method type in its bound (MT-VAR-S). Note that in the formalism, since all variables are bound in the primary

pattern, we can always invoke a method guaranteed to exist by a positive nested pattern.

MT-CLASS-R lists conditions for retrieving the type of η in $\texttt{C<}\overline{\texttt{T}}\texttt{>}$, where $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ has reflectively declared methods. If the range of reference, which is the current reflective environment, is subsumed by the declaration reflective environment, the type of η is the declared types in $\texttt{C<}\overline{\texttt{X}}\texttt{>}$, with the substitutions of $[\overline{\texttt{T}}/\overline{\texttt{X}}]$, and the type substitutions for unifying the declaration range and the reference range, $[\overline{\texttt{W}}/\overline{\texttt{Y}}]$. MT-SUPER-R simply states that when the reference reflective environment is disjoint from every declaration reflective environment in $\texttt{C<}\overline{\texttt{T}}\texttt{>}$, we must look to the superclass for the type of η.

5.3 Soundness:

We prove the soundness of FMJ by proving Subject Reduction and Progress.

Theorem 1 [Subject Reduction]: If $\Delta;\Lambda;\Gamma\vdash\texttt{e}\in\texttt{T}$ and $\texttt{e}\ \rightarrow\ \texttt{e}'$, then $\Delta;\Lambda;\Gamma\vdash\texttt{e}'\in\texttt{S}$ and $\Delta\vdash\texttt{S}<:\texttt{T}$ for some S.

Theorem 2 [Progress]: Let e be a well-typed expression. 1. If e has $\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).f}$ as a subexpression, then $\emptyset\vdash \mathit{fields}(\texttt{C<}\overline{\texttt{T}}\texttt{>})=\overline{\texttt{U}}\ \overline{\texttt{f}}$, and $\texttt{f = f}_i$. 2. If e has $\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).m(}\overline{\texttt{d}}\texttt{)}$ as a subexpression, then $\mathit{mbody}(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>})=(\overline{\texttt{x}},\texttt{e}_0)$ and $|\overline{\texttt{x}}|\ =\ |\overline{\texttt{d}}|$.

Theorem 3 [Type Soundness]: If $\emptyset;\emptyset;\emptyset\vdash\texttt{e}\in\texttt{T}$ and $\texttt{e}\longrightarrow^{*}\texttt{e}'$, then $\texttt{e}'$ is a value v such that $\emptyset;\emptyset;\emptyset\vdash\texttt{v}\in\texttt{S}$ and $\emptyset\vdash\texttt{S}<:\texttt{T}$ for some type S.

Type Unification:

$$\frac{[\overline{\texttt{U}}/\overline{\texttt{Z}}]\overline{\texttt{T}}=[\overline{\texttt{U}}/\overline{\texttt{Z}}]\overline{\texttt{S}}\quad\text{for all } \texttt{Z}_i\in\overline{\texttt{Z}},\ \Delta\vdash\texttt{U}_i\prec:_{\overline{\texttt{Z}}}\texttt{Z}_i}{\Delta;[\overline{\texttt{U}}/\overline{\texttt{Z}}]\vdash \mathit{unify}(\overline{\texttt{T}},\ \overline{\texttt{S}})}\qquad\text{(UNI)}$$

Pattern matching rules:

$$\Delta\vdash\texttt{T}\prec:_{\overline{\texttt{Z}}}\texttt{T}\qquad\text{(PM-REFL)}$$

$$\frac{\Delta\vdash\overline{\texttt{T}}\prec:_{\overline{\texttt{Z}}}\overline{\texttt{S}}}{\Delta\vdash\texttt{C<}\overline{\texttt{T}}\texttt{>}\prec:_{\overline{\texttt{Z}}}\texttt{C<}\overline{\texttt{S}}\texttt{>}}\qquad\text{(PM-CL)}$$

$$\frac{CT(\texttt{C})=\texttt{class C<}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}\texttt{>}\texttt{<:T }\{\ldots\}\quad\Delta\vdash[\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{T}\prec:_{\overline{\texttt{Z}}}\texttt{D<}\overline{\texttt{S}}\texttt{>}}{\Delta\vdash\texttt{C<}\overline{\texttt{T}}\texttt{>}\prec:_{\overline{\texttt{Z}}}\texttt{D<}\overline{\texttt{S}}\texttt{>}}\qquad\text{(PM-CL-S)}$$

$$\frac{\texttt{Z}\in\overline{\texttt{Z}}\quad\texttt{T}\notin\overline{\texttt{Z}}\quad \mathit{bound}_\Delta(\texttt{T})=\texttt{C<}\overline{\texttt{T}}\texttt{>}\quad\Delta\vdash\texttt{C<}\overline{\texttt{T}}\texttt{>}\prec:_{\overline{\texttt{Z}}}[\texttt{C<}\overline{\texttt{T}}\texttt{>}/\texttt{Z}]\mathit{bound}_\Delta(\texttt{Z})}{\Delta\vdash\texttt{T}\prec:_{\overline{\texttt{Z}}}\texttt{Z}}\qquad\text{(PM-VAR)}$$

$$\frac{\texttt{Z}_i\in\overline{\texttt{Z}}\quad\texttt{Z}_j\in\overline{\texttt{Z}}\quad\begin{cases}\Delta\vdash[\texttt{Z}_i/\texttt{Z}_j]\mathit{bound}_\Delta(\texttt{Z}_j)\prec:_{\overline{\texttt{Z}}}\texttt{Z}_i & \textit{or}\\ \Delta\vdash[\texttt{Z}_j/\texttt{Z}_i]\mathit{bound}_\Delta(\texttt{Z}_i)\prec:_{\overline{\texttt{Z}}}\texttt{Z}_j\end{cases}}{\Delta\vdash\texttt{Z}_i\prec:_{\overline{\texttt{Z}}}\texttt{Z}_j}\qquad\text{(PM-PVARS)}$$

Figure 6. Unification and pattern-matching rules

5.4 Decidability

To establish the decidability of our type system, we enforce limitations on possible circularities in either subtyping or static iteration cross-type references. For the former, we inherit a standard technique from Allen et al. [1]. Applying the same restrictions (i.e., a declared supertype cannot be a type with a mixin superclass itself), we can guarantee that there is no cyclic inheritance in FMJ. Another source for non-termination in FMJ is in circularly dependent method definitions. For example,

```
1 class C<X extends D<X>> {
2   <R>[m]for(R m() : X.methods) ...
3 }
4 class D<X> extends C<D<X>> { ... }
```

The methods of C<X> are circularly defined: they reflect over the methods of X, which include the methods of D<X>, which, in turn, include the methods of C<D<X>>! This type of definition would cause infinite recursion in the derivation of *mtype*.

We detect such circularity by constructing a chain of reflective *reachability*. The chain of reachability for a type T is essentially all the types $\mathit{mtype}(\texttt{n},\texttt{T})$ could recursively call upon. For example, the chain of reachability for the above C<X> is C<X>, X, D<X>, C<D<X>>, ... We stop the chain construction as soon as we see a re-occurrence of any type already in the chain, in *any* form of instantiation. We reject classes with such circular dependency. Since there is a finite number of classes, the chain must either see a reoccurrence of some class, or be finitely sized. The length of the chain serves as a measure function for each call of *mtype*. The finite size of the chain means the measure function cannot decrease infinitely, thus proving termination. A more sophisticated protocol would be possible, to make the check less conservative, but we have yet to encounter a realistic use that needs it.

6. Related Work

As discussed earlier, MorphJ's closest relatives are MJ [14] and CTR [7]. CTR is an extension to C# that pioneered the use of patterns for reflective iteration and was one of the first systems to aim for modular type safety. Nevertheless, its modular guarantees concern only validity of references and not the absence of declaration conflicts. Additionally, CTR does not allow matching multiple method argument types, and there is currently no formal type system or soundness guarantees. A unique aspect of CTR (compared to MJ or MorphJ) is that it transforms classes in-place, which enables some interesting applications. MJ, on the other hand, introduced two main elements: static checking for disjointness of reflective declarations, and the integration of static reflection as an extension of standard generics. MJ has a formal type system, with a soundness proof but with no demonstration of its decidability. MorphJ improves over both CTR and MJ by adding more expressiveness through nested patterns, while keeping or strengthening the typing guarantees, and by validating the promise of the overall approach with larger-scale applications.

Static reflection mechanisms such as Genoupe [6] and SafeGen [13] attempted to allow declaration using reflection. Yet none of these mechanisms offer full modular type-checking guarantees. For instance, the Genoupe [6] approach has been shown unsafe, as its reasoning depends on properties that can change at runtime; SafeGen [13] has no soundness proof and relies on the capabilities of an automatic theorem prover—an unpredictable and unfriendly process for a programmer. Additionally, these mechanisms use complex syntax for retrieving reflective members, whereas MorphJ utilizes patterns very similar to method and field signatures.

An extension of traits [21] offers pattern-based reflection by allowing a trait to use name variables for declarations. However, [21] does not offer static iteration over the members of classes—a name-generic trait must be mixed in once for each name instance.

The main capabilities of MorphJ can typically be emulated only with lower-level mechanisms, such as reflection, meta-object protocols [17], aspect-oriented programming [18], or pattern-based program generation and transformation [3, 4, 24]. The goal of our work is to promote these abilities to high-level language features, with full modular type-safety. None of the above mechanisms offer such safety guarantees: a transform, aspect, or meta-class cannot be type-checked independently from the rest of the program, in a way that guarantees it is well-typed for all its possible uses.

An interesting special case of program generation is *staging languages* such as MetaML [23] and MetaOCaml [5]. These languages offer modular type safety: the generated code is guaranteed correct for any input, if the generator type-checks. Nevertheless, MetaML

and MetaOCaml do not allow generating identifiers (e.g., names of variables) or types that are not constant. Neither do they allow generation of code by reflecting over a program's structure. Generally, staging languages target program specialization rather than full program generation: the program must remain valid even when staging annotations are removed. It is interesting that even recent meta-programming tools, such as Template Haskell [22] are explicitly not modularly type safe—its authors acknowledge that they sacrifice the MetaML guarantees for expressiveness.

There has been a line of work focused on providing statically type-safe generic traversal of data structures [19, 16]. For instance, the "scrap your boilerplate" [19] line of work offers extensions of Haskell that allow code to abstract over the exact structures of the data types it acts on, and to have the appropriate functions invoked when their expected data types are encountered during traversal. Abstracting over the structures of data types in functional languages is similar to abstracting over the fields and methods of classes in object-oriented languages. [25] offers such generic traversal capabilities for Java. However, whereas [19, 16, 25] focus on offering structurally-generic *traversal*, MorphJ focuses on structurally-generic *declarations*. Neither of [19, 16, 25] allow more functions to be declared using the names or types retrieved from a non-specific data type. Thus, these techniques fall short of MorphJ (and static reflection work in general [14, 6, 7, 13]) in this respect. On the other hand, MorphJ is not well-suited for writing generic traversal code. Traversing data structures and invoking methods on objects encountered is largely based on the dynamic types of these objects. MorphJ's reflective declarations are based purely on the static types of fields and methods.

7. Conclusions

We believe that MorphJ and the general approach of class morphing represent a significant trend in the evolution of programming languages. Most major advances in programming languages are modularity or re-usability enhancements. The first step was taken with *procedural abstraction* in the 50s and 60s, which culminated in structured programming languages. Procedural abstraction captured algorithmic logic in a form that could be multiply reused both in the same program and across programs, over different data objects. The next major abstraction step was arguably *type abstraction* or *polymorphism*, which allowed the same abstract logic to be applied to multiple types of data, although the low-level code for each type would end up being substantially different. The next big step in language evolution can perhaps be called *structural abstraction*. Structural abstraction is abstraction over the structure of other program elements. Mechanisms like CTR, MorphJ, or the "scrap your boilerplate" approach are instances of structural abstraction: they allow safe static reflection over members of a type. We believe that the inclusion of such constructs in mainstream languages will be a topic of major importance for decades to come and that this paper represents a big step forward in this direction.

Availability and Acknowledgments

MorphJ is available at *http://code.google.com/p/morphing/*
This work was funded by the NSF under grant CCR-0735267.

References

[1] E. Allen, J. Bannet, and R. Cartwright. A first-class approach to genericity. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[2] Apache Software Foundation. *Byte-code engineering library. "http://jakarta.apache.org/bcel/manual.html"*. Accessed Mar.'08.

[3] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[4] J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proc. of Programming Language Design and Implementation (PLDI)*, 2002.

[5] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Proc. of Generative Programming and Component Engineering (GPCE)*, 2003.

[6] D. Draheim, C. Lutteroth, and G. Weber. A type system for reflective program generators. In *Proc. of Generative Programming and Component Engineering (GPCE)*, 2005.

[7] M. Fähndrich, M. Carbin, and J. R. Larus. Reflective program generation with patterns. In *Proc. of Generative Programming and Component Engineering (GPCE)*, 2006.

[8] E. Gamma, R. Helm, and R. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[9] M. Herlihy. *SXM: C# Software Transactional Memory. "http://www.cs.brown.edu/ mph/SXM/README.doc"*. Accessed Mar.'08.

[10] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006.

[11] S. S. Huang and Y. Smaragdakis. Easy language extension with Meta-AspectJ. In *Proc. of International Conference on Software Engineering (ICSE)*, 2006.

[12] S. S. Huang and Y. Smaragdakis. Morphing with nested patterns: Making generic classes highly configurable. Technical report, 2007. `http://www.cc.gatech.edu/~ssh/mjnested-tr.pdf`.

[13] S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with SafeGen. In *Proc. of Generative Programming and Component Engineering (GPCE)*, 2005.

[14] S. S. Huang, D. Zook, and Y. Smaragdakis. Morphing: Safely shaping a class in the image of others. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, 2007.

[15] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[16] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997.

[17] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[18] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of European Conf. on Object-Oriented Programming (ECOOP)*, 1997.

[19] R. Lämmel and S. P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, 2003.

[20] M. Mohnen. Interfaces with default implementations in Java. In *Proc. of Principles and Practice of Programming*, 2002.

[21] J. Reppy and A. Turon. Metaprogramming with traits. In *ECOOP '07: Proceedings of the European Conference on Object-Oriented Programming*, 2007.

[22] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proc. of the ACM SIGPLAN workshop on Haskell*, 2002.

[23] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proc. of Partial Evaluation and semantics-based Program Manipulation (PEPM)*, 1997.

[24] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In *Domain-Specific Program Generation*. Springer-Verlag, 2004. LNCS 3016.

[25] S. Weirich and L. Huang. A design for type-directed Java. In *Workshop on Object-Oriented Developments (WOOD)*, 2004.