jUCM: Universal Class Morphing (position paper)

Aggelos Biboudis
University of Athens
biboudis@di.uoa.gr

George Fourtounis
University of Athens
gfour@di.uoa.gr

Yannis Smaragdakis
University of Athens
yannis@smaragd.org

Abstract

We extend prior work on *class-morphing* to provide a more expressive pattern-based compile-time reflection language. Our *MorphJ* language offers a disciplined form of metaprogramming that produces types by statically iterating over and pattern-matching on fields and methods of other types. We expand such capabilities with "universal morphing", which also allows pattern-matching over types (e.g., all classes nested in another, all supertypes of a class) while maintaining modular type safety for our meta-programs. We present informal examples of the functionality and discuss a design for adding universal morphing to Java.

1. Introduction

The ultimate flexible software component is one that safely adapts its behavior *and* its interface depending on its uses. When a component's interface is statically defined (as in the case of classes in a statically typed language), such adaptation requires a meta-programming facility. Metaprogramming is typically low-level and unwieldy, with few guarantees of safety. Mechanisms for compile-time reflection [2, 5] have been proposed to address such safety needs.

In our previous work [3–5] we presented and extended *MorphJ*. MorphJ is a language that adds compile-time reflection capabilities to Java. A programmer is able to capture compile-time patterns and encode them in (meta-)classes. Each pattern is associated with a generative scenario. For instance, a morphed class `Listify` may statically iterate over all the methods of another, unknown, type, `Subj`, pick those that have a single argument, and offer isomorphic methods: whenever `Subj` has a method with argument `A`, `Listify` accepts a `List<A>` . (The implementation of every method in `Listify` can then, e.g., iterate over all list elements, and manipulate them using `Subj`s methods.)

```
class Listify<Subj> {
  Subj ref;
  Listify(Subj s) {ref = s;}

  <R,A>[m] for (public R m(A): Subj.methods)
  public R m (List<A> a) {
    ... /* e.g., call m for all elements */
  }
}
```

MorphJ offers program transformation capabilities but with modular type-safety guarantees: type-checking (via MorphJ) the code of `Listify` guarantees that all the classes it may produce (for any type `Subj`) also type-check (via the plain Java type system).

In this work we complement MorphJ with the ability to statically reflect over classes, instead of just fields and methods. We discuss our early motivation with examples over nested classes.

2. Application: (Static) Nested Classes

Classes are the typical unit of modularity in an object-oriented language. To form larger modules, one can group classes together into components such as packages, or assemblies. At the language level, the class mechanism itself can serve as a component, encapsulating other classes. This is elegant from a modeling standpoint (a single concept for all levels of modularity) and even captures existing language features that allow the nesting of classes.

Nested classes can be either inner classes or static nested classes in Java. Folklore in the Java community suggests to favor static nested classes over inner classes and use the latter only if it is absolutely needed (Item 22 in [1]). Programmers use static nested classes in various practical scenarios. In compiler engineering, static nested classes are usually used when representing abstract syntax tree (AST) nodes. `javac` in fact, contains static nested classes for AST nodes that also extend the top-level class, `JCTree`.[1] Tools such as ANTLR that generate parsers also generate code of this form. In UI engineering, several tools generate class definitions that contain static nested classes—e.g., the Android Asset Packaging Tool that generates the `R` class, a strongly-typed view of resource IDs for all the resources in the resources directory.[2]

Our universal morphing techniques find interesting applications in (static) nested classes.

Ex1. Replace inheritance with delegation for all classes in a library. In this example we want to replace inheritance

[1] `http://hg.openjdk.java.net/jdk8/jdk8/langtools/file/jdk8-b132/src/share/classes/com/sun/tools/javac/tree/JCTree.java`

[2] `http://developer.android.com/guide/topics/resources/accessing-resources.html`

with delegation automatically for all static nested classes of `Library`. This feature is offered as a refactoring mechanism in IDEs today but the user may need to generate a delegation-view via an existing hierarchy for all classes. Such existing hierarchy is enclosed in the class `Library` below:

```
class Library {
  static class Vector {
    boolean isEmpty() {}
  }
  static class Stack extends Vector { }
}
```

The programmer's intention is to have a view of the library that relies on delegation like the one below:

```
class Library {
  static class Vector {
    boolean isEmpty() {}
  }
  static class Stack {
    Vector subobject;
    boolean isEmpty() { subobject.isEmpty(); }
  }
}
```

We introduce the static `for` keyword for static reflection over classes. In line 2 of the `LibraryDelegated` we use it to iterate over all classes in the type `Library`. The pattern that we look for is that of classes that extend some other class. All classes inside `L` that are going to be captured will have a corresponding definition in `Delegate<L>`. Inside each class definition we define a `subobject` field of type `S` (the supertype). In lines 5-6 we rely on the `static-for` we introduced in MorphJ.

```
class Delegate<L> {
  <C,S> for (C extends S : L.classes)
  static class C {
    S subobject#S;
    <R, A*> [m] for(public R m(A) : S.methods)
    R m(A a){ return subobject#S.m(a); }
  }
}
```

Ex2. Introduce interface and add a new method. In the following we introduce an interface that is implemented by all static nested classes. Again this is realized by reflecting over all classes of the type that is going to parameterize the `AlertingGraph` type.

```
interface Alert { void alert(); }

class AlertingGraph <class X> {
  [N] for (N : X.classes)
  static class N extends X.N implements Alert {
    [m] for(public void m () : N.methods)
    public void m() {
      alert();
      m();
    }
    void alert() { System.out.println("Alerted!"); }
  }
}
```

Ex3. Merge two classes into one (including nested classes). We can create a highly generic class that consists of the union of members (methods and classes) of two others, with one of them taking precedence.

```
class Union<class B, class C> {
  <R, A*> [m] for (R m(A) : B.methods)
  R m(A a) { super.m(a); }
  <R, A*> [m] for (R m(A) : C.methods;
                   no R m(A): B.methods)
  R m(A a) { super.m(a); }

  [N] for (N : B.classes)
  class N {
    <R,A> [m] for (R m(A) : N.methods)
    R m(A a) { b.m(a); }
    <NB> for (NB : N.classes)
    class NB extends N.NB {  }
  }
  [N] for (N : C.classes; not N : B.classes)
  class N {
    <R,A*>[m] for (R m(A) : N.methods)
    R m(A a) { b.m(a); }
    <NC> for (NC : N.classes)
    class NC extends N.NC {  }
  }
}
```

There is a wealth of other examples of universal morphing. For instance, we can iterate over all interfaces implemented by a class, we can offer highly-generic *mixin layers* [6], we can scrap the traversal boilerplate in external visitor patterns.

3. Conclusion

We are working on jUCM, an extension of MorphJ that enables more compile-time reflection patterns. A major challenge includes designing the type system extension that will ensure modular type-safety of meta-programs.

Acknowledgments. We gratefully acknowledge funding by the Greek Secretariat for Research and Technology under the "MorphPL" Excellence (Aristeia) award.

References

[1] J. Bloch. *Effective Java*. Addison-Wesley, Upper Saddle River, NJ, 2nd edition, May 2008. ISBN 9780321356680.

[2] M. Fähndrich, M. Carbin, and J. R. Larus. Reflective program generation with patterns. In *GPCE '06: Proc. of Generative Programming and Component Engineering*, pages 275–284, New York, NY, 2006. ACM Press. ISBN 1-59593-237-2.

[3] P. Gerakios, A. Biboudis, and Y. Smaragdakis. Forsaking Inheritance: Supercharged Delegation in DelphJ. In *Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 233–252, New York, NY, USA, 2013. ACM.

[4] S. S. Huang and Y. Smaragdakis. Expressive and safe static reflection with MorphJ. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, volume 43, pages 79–89, Tucson, AZ, USA, 2008.

[5] S. S. Huang and Y. Smaragdakis. Morphing: Structurally Shaping a Class by Reflecting on Others. *ACM Transactions on Programming Languages and Systems*, 33(2):1–44, Feb. 2011. ISSN 0164-0925.

[6] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.