Journal of Information Processing Vol.26 1–10 (Jan. 2018)

[DOI: 10.2197/ipsjjip.26.1]

Regular Paper

Compilation for a Version Polymorphic Object-Oriented Programming Language

Luthfan Anshar Lubis[1,a)] Yudai Tanabe[1] Masuhara Hidehiko[1]

Received: March 4, 2016, Accepted: August 1, 2016

Abstract:
Programming with versions (PWV) introduces the notions of versions into the language semantics and allows for version-safe multi-version programming. BatakJava is an object-oriented implementation of PWV that adds versions as an attribute of a class, where different versions of a class can coexist and interact with each other in the same program. To ensure version safety, BatakJava uses a constraint-based type system during compilation that solves for a fixed version assignment for every type instance in the program. However, this compilation scheme significantly reduces the flexibility of multi-version programming that may require type instances to be used with different version assignments. To solve this problem, we propose the use of parameterized versions to allow version polymorphic definitions. This paper presents the design for a version polymorphic extension of BatakJava, its implementation that modifies the original type analysis and code generation, and a case study to demostrate the improved flexibility of multi-version programming with version polymorphism.

Keywords: object-orientation, versioning, software evolution

[1] Tokyo Institute of Technology, Meguro, Tokyo 152–8550, Japan
[a)] luthfanlubis@prg.is.titech.ac.jp

1. Introduction

... similar opening as the other PWV papers before talking about PWV

Programming with versions is a paradigm that introduces versions as part of the language semantics, managing them as a type's resources through a coeffectful type system **?**. It was demonstrated in λVL through versioned values that enables multi-version programming and allows for static dependency analysis by analyzing version resources in type aiming at increasing programming's flexibility and encouraging gradual dependency migration.

BatakJava **?**, an extension of Java, builds on the idea of λVL and explores programming with versions in an object-oriented setting, treating versions as an element of class. It relaxes λVL's limitation by allowing cross-version computations and includes versions in data structures and the module system.

However, BatakJava does not support *version polymorphism*, which means a single definition cannot represent multiple implementations requiring different versions even when they are compatible. This inflexibility contradicts with the nature of multi-version programming that would require definitions to support instantiations by different versions.

We propose to add genericity to versioning in BatakJava through version parameterization, similar to type parameters in Java generics **?**. Parameters would abstract version assignments to specific class instances in the definition.

Our contributions are as follows:

- We propose a version polymorphic extension of BatakJava that allows for version parameterization in class definitions - We implement the extension by modifying BatakJava's type analysis and code generation - We demonstrate the flexibility of programming with version polymorphism through a case study

2. Version Polymorphism

2.1 An Illustrating Example

Let's illustrate this with a simple example involving different hashing algorithms defined in different versions. Listing 1[*1] shows version 1 and 2 of the class Hash that respectively implements MD5 and SHA-3 hashing algorithms written in BatakJava. The method mkHash hashes a String into a HashValue. Both versions are statically compatible with each other, meaning that each definition can replace the other version without breaking compilation. However, they are semantically incompatible because of the different algorithms. If a user intends to use any version of Hash in their program, they must not check a hash value generated in one version against a hash value generated from another version. To ensure that only the correct version of hash value is returned by and passed into Hash, the versions for each respective HashValue are fixed through HashValue#1# and HashValue#2#.

Suppose then that the hashing algorithm defined in Hash is used through a Dir class shown in Listing 2 that can access files and check their contents. The method exists in Dir accepts a hash value c, hashes it with a specific hashing algorithm, and checks its equality with already stored files using match.

Under the assumption of programming with versions, users may want to use different versions of statically compatible Hash and HashValue through a single definition of Dir. In Listing 3, an App class processing different types of string data through Dir may wish to use different versions of the algorithm as required. Due to possible constraints in the data storage, the user may need to use different algorithms in processID and processLog. Here, the IDs are processed using the older algorithm by passing a hash value generated from new Hash#1#(), while the logs are processed by the new algorithm with a hash value generated from new Hash#2#().

However, compiling the program in BatakJava would result in:

```
No solution found for the program
```

```
class Hash ver 1 {
  Hash() { }
  HashValue#1# mkHash(String s) {
    return new HashValue#1#(/* MD5 algorithm */);
  }
  boolean match(String s, HashValue#1# h) {
    return this.mkHash(s).equals(h);
  }
}

class Hash ver 2 {
  Hash() { }
  HashValue#2# mkHash(String s) {
    return new HashValue#2#(/* SHA-3 algorithm */);
  }
  boolean match(String s, HashValue#2# h) {
    return this.mkHash(s).equals(h);
  }
}
```

Listing 1: Two versions of Hash in BatakJava

```
class Dir ver 1 {
  Dir() { }
  String[] getFiles() { /* retrieve stored strings */}
  boolean exists(HashValue c) {
    Hash hash = new Hash();
    for (String s: getFiles()) {
        if (hash.match(s, c) { return true; }
    }
    return false;
  }
}
```

Listing 2: Dir class using Hash and HashValue

```
class App ver 1 {
  boolean processID(String n) {
    // using MD5 algorithm
    return new Dir().exists(new Hash#1#().mkHash(n));
  }

  boolean processLog(String n) {
    // using SHA-3 algorithm
    return new Dir().exists(new Hash#2#().mkHash(n));
  }
}
```

Listing 3: App instantiating Dir with different versions

3. Design

The problem with BatakJava lies in its limited design that fixes the version for every class instance. To allow this reuse of the same definition, in other words, *version polymorphism*, we have to relax this limitation in the original design of BatakJava.

A straightforward solution is to take a similar approach to Java generics **?**. In Java generics, types of fields or method arguments are polymorphised through type parameterizations to be made concrete during class instantiations or method invocations. The correctness of the type parameters and arguments are checked using their upper bounds as their type identities.

We follow the approach by extending BatakJava with version polymorphism through version parameterization. Version parameters abstract version assignments to class instances in the class definition and method declarations.

We illustrate this with the new class Dir in Listing 4 with polymorphic definition. The parameters V and W surrounded by the double angle brackets <<>> are version parameters declared for the class. The version parameters are used by passing them to class access as version specifications inside the class definition. Class access refers to class annotations appearing in parameters and variable declarations such as Hash#V# hash. The class instance with sharp notation Hash#V# is similar in meaning to Hash#1# where the class instance Hash is specified to version V. Each version parameter is associated with a specific class. In Dir, V and W are respectively associated with the class Hash and HashValue.

Version polymorphic classes can be instantiated by passing version arguments during object instantiations. Listing 5 shows the modified App using the polymorphic Dir. The two object instantiations for Dir in both processID and processLog are distinguished by the version arguments given to each instance

[*1] The code in this chapter does not include modifiers such as public for brevity.

through the angle brackets notation. In `processID`, the `Dir` object is instantiated by passing <<`1,1`>> as version arguments to `Dir`. This denotes that both `Hash` and `HashValue` in this object are set to version 1 and 1. Similarly, <<`2,2`>> is passed to `Dir` instantiation in `processLogs` denoting that `Hash` and `HashValue` are set to version 2 and 2.

Unlike type parameters in Java generics, version parameters do not depend on version bounds for version checking. The compilation solves the set of solutions for the version parameters `V`, `W` and check the version arguments against these solutions. For the given example, the set of compatible solutions are

$$(\mathtt{V},\mathtt{W}) \in \{(1,1),(2,2)\}$$

Hence, based on the solutions found, the two object instantiations for `Dir` in both `processID` and `processLog` are well-typed. Conversely, incorrect object instantiations such as `new Dir<<1,2>>()` would be rejected.

A method declaration can also declare its own version parameters. The illustrating example can be rewritten by polymorphising the method declaration for `exists`. Listing 6 introduces version parameters `V` and `W` in method `exists`. Similarly as before, the parameters `V` and `W` are respectively associated with `Hash` and `HashValue`. Listing 7 shows the version arguments being passed into `exists` invocations.

Note that in the current design we do not consider version polymorphic inheritance. This is because allowing polymorphic superclass definitions highly complicates a class structure. Although it is feasible to implement such a language, we consider leaving the burden of tracking which object inherits from which version of a class to the programmers to be disadvantageous.

```
class Dir ver 1 <<V,W>> {
  String[] getFiles() { /* retrieve stored strings */ }
  boolean exists(HashValue#W# c) {
    Hash#V# hash = new Hash#V#();
    for (String s: getFiles()) {
      if (hash.match(s, c)) { return true; }
    }
    return false;
  }
}
```

Listing 4: Polymorphic `Dir` class declaration

```
class App ver 1 {
  boolean processID(String n) {
    return new Dir<<1,1>>().exists(new Hash().mkHash(n));
  }
  boolean processLog(String n) {
    return new Dir<<2,2>>().exists(new Hash().mkHash(n));
  }
}
```

Listing 5: `App` with polymorphic instantiations of `Dir`

4. Implementation

Polymorphic BatakJava is implemented as an extension of the base BatakJava. It is similarly compiled back into Java. Figure 1 shows the compilation process for a polymorphic BatakJava program. There are four main points of modification over the base

```
class Dir ver 1 {
  String[] getFiles() { /* retrieve stored strings */ }
  <<V,W>> boolean exists(HashValue#W# c) {
    Hash#V# hash = new Hash#V#();
    for (String s: getFiles()) {
      if (hash.match(s, c)) { return true; }
    }
    return false;
  }
}
```

Listing 6: Polymorphic `exists` method declaration

```
class App ver 1 {
  boolean processID(String n) {
    return new Dir().<<1,1>>exists(new Hash().mkHash(n));
  }
  boolean processLog(String n) {
    return new Dir().<<2,2>>exists(new Hash().mkHash(n));
  }
}
```

Listing 7: `App` with polymorphic invocations of `exists`

Fig. 1: Structure of polymorphic BatakJava's compiler

implementation:

(1) Dependency analysis before local type check
(2) Version arguments check on polymorphic instances based on prior solutions
(3) Less restrictive component-wise constraint generation
(4) Version arguments preserving code generation

4.1 Dependency Analysis

Dependency analysis computes the dependency graph that includes the order in which type checking and constraint generation of the classes in the program are conducted. The purpose of determining the order is to allow version checking on version arguments assigned to class instances.

The dependency relationship is computed using Tarjan's algorithm **?**. The algorithm takes the program as input. Each (version of) the class declaration `C ver n` is considered a vertex in the graph. Any reference to a class `D` through class access in a class declaration `C ver n` is considered an edge from the vertex representing `C ver n` to the vertex representing every version of `D`. The algorithm outputs the strongly connected components in the graph. This is described by a list of components consisting of vertices (versioned class declarations) and the order of the list. Class declarations belonging to the same component contain cyclical references. The class declarations composing the first component can be considered as the top-most module in a dependency relationship. Type checking, constraint generation and solving are then conducted from the first component.

Figure 2 shows the dependency analysis result for the hashing example. The arrows show the dependency relation and the dashed lines form the components. The program is composed of five components: the first component consists of `HashValue` ver.1 and 2; the second is `Hash` ver.1; the third is `Hash` ver.2; the fourth is `Dir` ver.1; the last is `App`.

Fig. 2: Dependency analysis result for the hashing example

4.2 Type Checking

Polymorphic BatakJava checks the correctness of polymorphic object instantiations. This is possible because the dependent classes are already checked and their solutions have been enumerated. In the hashing example, the polymorphic instance checks are executed in the class App. Class App is contained in the last component of the program, therefore during its check the solution for class Dir is already available. As mentioned above, the set of solutions $(\mathtt{V}, \mathtt{W})$ was $((1,1),(2,2))$. Using this set, version arguments on instantiations and invocations are checked. Failed checks are considered as type errors.

4.3 Constraint Generation

The problem to solve with the constraint generation in the base BatakJava is that constraints on type variables are generated globally. By doing this, the behavior of a class declaration is determined not only by the implementer but is also affected by how the users instantiate said class declaration. In the hashing example, the behavior of exists is not determined by the Dir itself, but also by how new Dir() in App invokes the method exists (Listing 3).

Suppose that e_1 and e_2 are respectively the body of processID and processLog in Listing 3, these expressions will generate the following constraints in BatakJava, where $\mathtt{T}_{\mathtt{HashValue}}$ is the type variable assigned to the parameter c of the method exists.

$$e_1 : \mathtt{T}_{e_1} \mid \mathcal{R}_1 \otimes \{\mathtt{T}_{\mathtt{HashValue}} = \mathtt{HashValue!1}\}$$
$$e_2 : \mathtt{T}_{e_2} \mid \mathcal{R}_2 \otimes \{\mathtt{T}_{\mathtt{HashValue}} = \mathtt{HashValue!2}\}$$

Eventually, the constraint for the whole program would have the following unsolvable constraint as its subconstraint

$$\mathcal{R}_1 \otimes \mathcal{R}_2 \otimes \{\mathtt{T}_{\mathtt{HashValue}} = \mathtt{HashValue!1} \wedge \mathtt{T}_{\mathtt{HashValue}} = \mathtt{HashValue!2}\}$$

Polymorphic BatakJava solves this by generating constraints and solving them component-wise. The order of components follows the result of the dependency analysis.

In the hashing example, constraints on Dir are already solved by the point Dir instances in App are checked. The expressions e_1 and e_2 therefore do not add constraints to the definition of class Dir. Concretely the new constraint generated by e_1 and e_2 do not constrain the parameter exists

$$e_1 : \mathtt{T}_{e_1} \mid \mathcal{R}_1$$
$$e_2 : \mathtt{T}_{e_2} \mid \mathcal{R}_2$$

The conjunction of the new sub-constraints which have to be solved is the following constraint.

$$\mathcal{R}_1 \otimes \mathcal{R}_2$$

4.4 Code Generation

The key change in polymorphic BatakJava's code generation is by relaxing the code fixing on class access in the resulting Java code.

Specifically in the base BatakJava, Java code is generated naively. The code generation selects one version assignment and transpiles the code into Java by fixing class access with the selected version assignments. Simply applying the new constraint generation approach described above to collect all the versions assignments is not enough to solve the issue because every class access in the code would still be fixed to one version. For example, the resulting Java code for Dir shown in Listing 8 will still not compile into bytecode due to type incompatibility. Specifically in this example, the Hash and HashValue in the method exists is fixed to version 1.

```
class Dir_v1 {
  ...
  boolean exists(HashValue_v1 c) {
    Hash_v1 hash = new Hash_v1();
    ...
  }
}
```

Listing 8: A section of code generation result for the hashing example

In the following section we will demonstrate the transpilation in polymorphic BatakJava by syntactic constructs.

Transpiling Class Declarations and Instantiations

The code generation for class declarations defines an interface for each set of versioned classes. In the example, an interface for every class Hash, HashValue, Dir, and App are added during code generation. The interface plays the role of version polymorphic classes after transpilation. Each specific version of the class is renamed similarly as in base BatakJava, for example Hash version 1 is renamed into Hash_v1 in Java and also made to implement the newly added interfaces. By doing so, it is possible to pass any specific version of a class as an argument to a version polymorphic parameter. Listing 10 shows the headers for Hash and Dir.

Versions for class access in the program, excluding class instantiations and cast expressions are not fixed during transpilation as shown in Listing 9.

In method exists, the versions for the polymorphic Hash and HashValue are not fixed allowing us during runtime to pass any version of HashValue as an argument for invoking exists. Version checking has been done during the previous step, so erasing version information (the parameter V in HashValue#V#) from the signature during this step will not cause a problem.

Version parameters and arguments are converted into Java objects with *factory interfaces* and *factory classes*. In class Dir, versin parameters V and W are respectively converted into factory interfaces for Hash and HashValue. Listing 11 shows the converted parameters V and W as new fields with type Hash_Factory and HashValue_ Factory in Dir_v1. The constructor defined in Dir is correspondingly modified to initialize the new fields for version parameters.

Listing 12 shows the definition of the factory interfaces and factory classes with specific versions for Hash. The interface Hash_Factory defines a method make that returns a Hash object. Each specific version of the factory Hash_v1_Factory and

```
boolean exists(HashValue c) {
  Hash hash = ...;
  ...
}
```

Listing 9: Partial transpilation of exists

```
interface Hash {...}

class Hash_v1 implements Hash {...}

class Hash_v2 implements Hash {...}

interface Dir {...}

class Dir_v1 implements Dir {...}
```

Listing 10: Header for Hash and Dir

```
class Dir_v1 {
  Hash_Factory V;
  HashValue_Factory W;
  Dir_v1(Hash_Factory V, HashValue_Factory W) {
    this.V = V; this.W = W;
  }
  boolean exists(HashValue c) {
    Hash hash = ...;
    for (String s: getFiles()) {
      if (hash.match(s, c)) { return true; }
    }
    return false;
  }
}
```

Listing 11: Transpilation of version parameters in Dir

```
interface Hash_Factory {
  Hash make();
}

class Hash_v1_Factory implements Hash_Factory {
  Hash make() { return new Hash_v1(); }
}

class Hash_v2_Factory implements Hash_Factory {
  Hash make() { return new Hash_v2(); }
}
```

Listing 12: Factory classes for Hash

```
class Dir_v1 {
  ...
  boolean exists(HashValue c) {
    Hash hash = V.make();
  }
}

class App_v1 {
  boolean processID(String n) {
    return new Dir_v1(new Hash_v1_Factory(),
                      new Hash_v1_Factory())
      .mkHash(s);
  }
  boolean processLog(String n) {
    return new Dir_v1(new Hash_v2_Factory(),
                      new Hash_v2_Factory())
      .mkHash(s);
  }
}
```

Listing 13: Runtime object instantiation

```
class Hash ver 3 {
  ...
  HashValue#2# mkHash(String s) {...}
  boolean match(String s, HashValue#2# h) { ... }
  boolean matchWith(String s, HashValue#2# h) { ... }
}
```

Listing 14: A new Hash with a different method

```
# In BatakJava
Hash#V# h1 = new Hash#V#(...);
Hash#W# h2 = new Hash#W#(...);
h1.match("some␣string", h2.mkHash("some␣string"));
h2.matchWith("some␣string", h1.mkHash("some␣string"));
```

Listing 15: Using the new Hash

```
# In Java
Hash h1 = V.make(...);
Hash h2 = W.make(...);
h1.match("some␣string", h2.mkHash("some␣string"));
h2.matchWith("some␣string", h1.mkHash("some␣string"));
```

Listing 16: Transpilation of the code using the new Hash

Hash_v2_Factory implements the method make by constructing the object in its specific version.

Object instantiations such as new Hash#V#() whose version is determined dynamically can preserve its version argument information in Java through the use of factory classes. In Dir_v1 in Listing 13, the object instantiation new Hash#V#() is converted into a make call on the factory object V. Specific version arguments such as <<1,1>> in the class App are converted into constructor arguments with specific factory classes namely new Hash_v2_Factory() and new Hash_v1_Factory().

Transpiling Method Declarations and Invocations

Code generation for methods require modifications in regards to handling the generated interfaces. We illustrate this with the example in Listing 14, where we add a new version of Hash with a new method matchWith while maintaining older methods.

Listing 15 shows a situation where a new version of Hash may be used together with an older version of Hash. Two polymorphic instances of Hash are defined where different methods are invoked on each instance.

Suppose that Listing 15 checks and a solution is available for V and W, then the resulting transpilation is the code shown in Listing 16. Here we can observe that the polymorphic class Hash#V# and Hash#W# on variables h1 and h2 are converted into interface types Hash. Following that, methods are invoked against these interface types. To allow the resulting code to compile and run in Java, all method declarations in each specific version need to be gathered in the interface and implementations for these methods are added retrospectively in all the specific versions that do not implement the methods.

Listing 18 shows how methods for Hash are all aggregated in the interface. The method matchWith missing in the older versions of Hash are correspondingly added the older versions with a default implementation that can be ignored. Again, since version checks have been conducted in the previous step, there should not be any invocation of matchWith from any Hash objects from ver.1 and 2.

Suppose that the additional method shared the same name and parameters, but with different return type as shown in Listing 17.

```
class Hash ver 3 {
  MyBoolean match(String s, HashValue h) { ... }
}
```

Listing 17: A new `Hash` with overlapping name and a different return type

The transpilation will rename the method into `match__MyBoolean` and if necessary replace any method invocations with the new method name.

```
interface Hash {
  // from Hash ver.1-3
  HashValue mkHash(String s);
  // from Hash ver.1-3
  boolean match(String s, HashValue h);
  // from Hash ver.3
  boolean matchWith(String s, HashValue h);
}

class Hash_v1 {
  Hash_v1() { }
  HashValue mkHash(String s) { ... }
  boolean match(String s, HashValue h) { ... }
  boolean matchWith(String s, HashValue h) {
    return false;
  }
}

class Hash_v2 {
  Hash_v2() { }
  HashValue mkHash(String s) { ... }
  boolean match(String s, HashValue h) { ... }
  boolean matchWith(String s, HashValue h) {
    return false;
  }
}

class Hash_v3 {
  Hash_v3() { }
  HashValue mkHash(String s) { ... }
  boolean match(String s, HashValue h) {
    return false;
  }
  boolean matchWith(String s, HashValue h) { ... }
}
```

Listing 18: Transpiling `Hash`es with different methods

Transpiling Field Declarations and Access

Similar concerns arise with accessing fields on version polymorphic instances. As class access are converted into Java's interface types, field access in BatakJava need to converted into method calls in Java.

We illustrate this through the field `val` in class `HashValue`. Listing 19 shows the definition for two versions of `HashValue` with one containing the instance variable `val` and their use in the `main` method.

The resulting transpilation is shown in Listing 20. Getter (`get__<field_name>`) and setter (`set__<field_name>`) methods are generated according to instance variables declared in the class. With the addition of methods in the transpilation of the specific class, those additional methods are also added to the interface and other version of the class where the fields are not defined.

```
class HashValue ver 1 { }
class HashValue ver 2 {
  int val;
}

void main(String[] args) {
  HashValue#V# h = new HashValue#V#(10);
  int hv = h.val;
  h.val = 20;
}
```

Listing 19: Field access and assignment on a `HashValue` object

```
interface HashValue {
  int get__val();
  int set__val(int toAssign);
}

class HashValue_v1 implements HashValue {
  int get__val() { return 0; }
  int set__val(int toAssign) { return 0; }
}

class HashValue_v2 implements HashValue {
  int val;
  int get__val() { return this.val; }
  int set__val(int toAssign) { return this.val = toAssign; }
}

void main(String[] args) {
  HashValue h = V.make(10);
  int hv = h.get__val();
  set__val(20);
}
```

Listing 20: Transpilation of field access and assignment on `HashValue`

Same-named fields in newer versions with different types or fields shadowing other fields in the parent class are handled the same way as the methods with different return types, by properly renaming and replacing field access with the renamed method calls.

5. Core Calculus

6. Case Study

7. Concluding Remarks

Acknowledgments

Luthfan Anshar Lubis

Yudai Tanabe

Hidehiko Masuhara