Toward Parameterized Versions in Object-oriented Version Programming

Luthfan Anshar Lubis, Yudai Tanabe, Masuhara Hidehiko (Tokyo Institute of Technology), Tomoyuki Aotani (Mamezou Co. Ltd.)

Background

Programming with versions[2] introduced versions as element of types. Implemented in the language **BatakJava**[1].

```
ver.1  class Point {
         m1() {...}
       }
ver.2  class Point {
         m2() {...}
         dist(Point p)
           {...}
       }                                  Java
```

→

```
class Point ver 1 {
  m1() {...}
}
class Point ver 2 {
  m2() {...}
  dist(Point p) {...}
}                                  BatakJava
```

Benefit: Allows statically type-checked multi-version programming → eases handling dependency update

```
Point p1 = new Point(); p1.m1();   — ver.1
Point p2 = new Point(); p2.m2();   — ver.2
p2.dist(p1);
```

Point ver.1 and 2 can be computed together

Compilation: BatakJava is compiled into Java by fixing the versions in the definitions.

```
class Point ver 2 {
  m2() {...}
  dist(Point p) {...}
}                                  BatakJava
```

→

```
class Point_ver_2 {
  m2() {...}
  dist(Point_ver_1 p)
    {...} }                        Java
```

compile into

```
Point p1 = new Point(); p1.m1();
Point p2 = new Point(); p2.m2();
p2.dist(p1);                       BatakJava
```

→

```
Point_ver_1 p1 = new Point_ver_1(); p1.m1();
Point_ver_2 p2 = new Point_ver_2(); p2.m2();
p2.dist(p1);                       Java
```

Problems

Fixing versions during compilation **disallows** polymorphic definitions. Results in inflexible definitions.

```
class Line ver 1 {
  Point a; Point b;
  Line(Point p1, Point p2) {...}}
    Point p1 = ...; p1.m1();
    Point p2 = ...; p2.m2();
    new Line(p1,p2);   instantiation with (ver.1,ver.2)
    new Line(p1,p1);   instantiation with (ver.1,ver.1)
```

compile error!

the version for a and b can't vary between objects

```
Point p1 = new Point(); p1.m1();
Point p2 = new Point(); p2.m2();
p2.dist(p1); p2.dist(p2);
```

call with ver.1 *call with ver.2*

p in `dist(Point p)` can't be both ver.1 and 2

Proposal

Introduce **version parameterization** similar to type parameters in Java[3,4].

```
class Line ver 1 {}   →   class Line ver 1 <V> {}
```

version parameter

Use Cases

1. Instantiating classes with version polymorphic structures.

```
class Line ver 1 <V,W> {
  Point#V# a;
  Point#W# b;
  Line(Point#V# p1, Point#W# p2) {
    a = p1; b = p2;
  }
}
```

 # notation binds `Point` to version V

Different `Line` objects with the same definitions

```
Point p1 = ...; p1.m1();
Point p2 = ...; p2.m2();
new Line<v1,v2>(p1,p2);
new Line<v1,v1>(p1,p1);
```

2. Polymorphic method calls

```
class Point ver 2 {
  void m2() {...}
  <V> int dist(Point#V# p)
    {...}  }

Point p1 = new Point(); p1.m1();
Point p2 = new Point(); p2.m2();
p2.dist<v1>(p1);
p2.dist<v2>(p2);
```

Same `Point` object calls a method with argument of different versions

3. Polymorphic inheritance

```
class Point ver 1 {
  Point(int x, int y)
    {...}
  void m1() {...}
}

class Point ver 2 {
  Point(int x, int y)
    {...}
  void m2() {...}
}
```

works under the condition that a constructor is shared

```
class ColoredPoint ver 1
    <V> extends Point#V#  {
  ColoredPoint(int x, int y) {
    super(x, y);
    ... } }

Point p1 = new ColoredPoint<v1>(...);
Point p2 = new ColoredPoint<v2>(...);
```

can extend different versions of `Point`

4. Defining ranges in version parameters

```
class Line ver 1 <V >= v2> {}
```

requiring the version substituted into V to be newer than ver.2

Future Work

Compilation to Java: different from Java's generics, method dispatch depends on versions

```
class Line ver 1 <V> {
  Point#V# p1;
  Point#V# p2; }
```

what should this be treated as in Java?

Formalization: expanding on previous work

References

[1] Lubis, Luthfan, et al. "BatakJava: An Object-Oriented Programming Language with Versions."
[2] Tanabe, Yudai, et al. "A Functional Programming Language with Versions."
[3] Igarashi, Atsushi, et al. "Featherweight Java: a minimal core calculus for Java and GJ."
[4] Bracha, Gilad, et al. "Making the future safe for the past: Adding genericity to the Java programming language."