TOKYO INSTITUTE OF TECHNOLOGY

MASTER THESIS

BatakJava: An Object-oriented Programming Language With Versions

Author:
Luthfan Anshar LUBIS

Student Number:
19M30436

Supervisor:
Hidehiko Masuhara

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Programming Research Group
Department of Mathematical and Computing Science

February 26, 2021

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

School of Computing
Department of Mathematical and Computing Science

Master of Science

BatakJava: An Object-oriented Programming Language With Versions

by Luthfan Anshar LUBIS

Programs are often bundled into a package with versions. However, versioning is a concept that exists outside of the language semantics. This makes software evolution a process that is vulnerable to issues, such as dependency hell, when dependencies are updated with breaking changes. We propose a language called *BatakJava*, where versions are included as an attribute of types. A package may have multiple implementations of a class, each distinguished by a version. In BatakJava, programmers can use multiple versions of a package simultaneously alongside a version inference system that allows users to program without explicitly annotating each class, while maintaining type safety. We demonstrate Batakjava's capabilities through a case study on the development of a program involving dependencies.

Acknowledgements

Throughout the writing of this thesis, I have been blessed with a great amount of support and assistance.

I would like to express gratitude to my supervisor, Professor Hidehiko Masuhara for providing not only guidance but also support through this thesis.

I would also like to thank Yudai Tanabe, whose cooperation and discussion with has helped shape this topic.

In addition, I would like to thank Youyou Cong and other members of the Programming Research Group for the support and constructive comments on the thesis.

I am grateful for my parents' constant emotional support through this especially perilous year. Thank you for being there to always give me the words of encouragement I need.

A very special gratitude to East Asian Circle of Applied Technology and Tsuji International Scholarship Foundation for providing financial support through my masters studies.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

As part of the software lifecycle, software go through *version updates*. Lehman mentioned continuing change as one of the laws of software evolution [16]. Through updates, software maintainers may fix recently discovered bugs, optimize performance, or provide new features for their downstream users. These improvements are achieved by possibly adding new functions or classes into existing software. It can also be achieved by modifying, fixing, or removing already defined functionalities. The software update then produces a new version of the package, possibly with different behavior. A *version* is then used to distinguish not only a whole software product but also a programming unit before and after an update.

Yet, studies have shown that software updates often contain breaking changes. Take the Java systems as an example. Most version updates in Maven repository are shown to include breaking changes that do not allow a particular version to substitute all other existing versions [25]. It was also found that behavioral backward incompatibilities are prevalent among Java software libraries [18].

Also, software reuse has become the norm for contemporary software development. Software reuse is the process of creating software systems from existing software rather than building from scratch [13]. Software reuse, including programming languages' libraries, has been found to provide significant positive effects on software quality and productivity [17]. The extent of software reuse in Java is shown to be common in almost all open source Java projects studied [9]. This is also evident with the increase of published jars in Maven's Central repository that continues every year[1].

The combination between a constant software update and the prevalence of software reuse leads to issues in maintaining updated dependencies. Updates of third-party library dependencies are not regularly practiced in Java systems, especially to fix vulnerabilities [14]. The heavy reliance of libraries results in complex inter-dependency relationships (also called *dependency hell*) that may influence whether dependencies are updated or not [14]. Existing techniques also have been observed to be insufficient in dealing with library migration [5].

Programming languages principally only allow one version of software to be used at a time. We argue that a single version of software is a coarse unit that leads to extra efforts in maintaining dependencies. Suppose a programmer wants to update their software to use an updated version of a library to use a particular feature available in that version. The update might contain other changes that are not necessarily needed. Such changes might break the other parts of the software or require different versions of other dependencies.

Here, we propose *programming with versions* in the context of object-oriented programming. In this concept, versions are not considered only meta-data existing outside of the program, but an element of the context accessible by programmers.

[1]`https://mvnrepository.com/repos/central`

By introducing the notion of version into the language semantics, each definition made within the program is associated with a version identifier that can be statically checked. Users can include multiple versions of the same dependency without the need to rename them. We argue that it will allow more fine-grained control of dependencies that may help programmers circumvent complex dependency's issues. With a more fine-grained control, programmers have the ability to avoid breaking change in the upstream by selectively choosing which version of an implementation they wish to use.

As proof-of-concept, we implemented a Java-based programming language *BatakJava* to illustrate this concept. Version numbers are included as an attribute of types, explicitly annotatated on the package declaration. Through the annotation, the static check incorporates version numbers, allowing the use of multiple versions of a package simultaneously. BatakJava is also equipped with a version inference system that allows users to program without having to explicitly annotate version on each class used within the program.

Chapter 2

Background

2.1 Breaking Changes

The crux of the incompatibility that occurs during software evolution lies in breaking changes. Figure 2.1 shows a simplified situation of a problematic dependency relation between software that involves breaking changes. (1) In the figure, an upstream package provides some functionalities. These functionalities are used by both midstream and downstream packages. (2) Afterward, the upstream package is updated to a new version (version `Y` is more recent that `X`) with breaking changes. However, in this case, the midstream package does not adopt the update. (3) Conversely, the downstream package attempts to adapt the new upstream package in its version update. In most programming languages, depending on the kinds of upstream package's breaking changes, the new downstream package cannot be built.

An example of generic taxonomy of breaking changes proposed in [15] grouped breaking changes into structural, behavioral, resourcing, and auxiliary. Programming with versions focuses on two of these: structural and behavioral. [6] listed the categories of API change in Java. We can correlate structural changes with API binary compatibility and behavioral changes with API contract compatibility.

Structural changes involve the change in signature. Plenty of examples can be found in Java. In the Google's library Guava[1], up to version 24.0, the class `Graphs` has a static method `equivalent(Graph,Graph)` used to check the equivalence between two `Graphs`. This method is then removed version 28.0 and users are expected to use the method `equals(Object)` inherited from the `Object`.

On the level of ecosystems, the Apache Hive[2] and Spark[3] are both parts of the Apache ecosystem, where Hive depends on Spark. Unfortunately, up until the time of writing, both depend on a different version of Guava, where Spark depends on version 14.0.1, while Hive depends on version 19.0. This can lead to both structural or behavioral incompatibility.

To handle structural incompatibilities, the tool needs to provide the feature that allows programmers to access the definition of the class before and after an update, preferably inferring the correct definitions to use automatically for the users.

An example of behavioral change in Java can be found in the Android Platform API. Up until version 18, the API provided the method `set` in class `AlarmManager` which schedules the alarm at the specified time. Android Platform 19 changed the behavior of `set` to not treat its argument as exact anymore, in its place a new method `setExact` is introduced to perform that function.

In the case of behavioral incompatibilities, an update may keep the same signature even after the update. Therefore, to handle this issue, the tool cannot only allow

[1] `https://github.com/google/guava`
[2] `https://github.com/apache/hive`
[3] `https://github.com/apache/spark`

usage of multiple versions simultaneously but give the programmers a way to specify which behavior is intended, as behaviors generally cannot be checked statically.

2.2 LambdaVL

The core calculus LambdaVL [27] first introduced the notion of versions as a resource attached to types.

Values, functions or literals, can take the form of *versioned values* that have different value depending on their version, although the base type cannot change by version. An example of a versioned value is given as follows.

```
{ V3_20 = 20, V3_22 = 22 | V3_22 }
```

The above line denotes a versioned value that is available in the versions V3_20 and V3_22, with value of 20 and 22 respectively. The version V3_22 after the vertical line denotes the default value. Versioned functions can be built in LambdaVL in the same manner.

LambdaVL provides the suspension operator ! that takes one expression and suspends the evalution until the value is required. Given a versioned function g and value u, both available in V3_20 and V3_22, the following program returns a suspended computation that returns an integer value in V3_20 and V3_22.

```
let !f = g in
  let !x = u in
    !(f x)
```

There is also extraction expression [e.v] to extract value in the version v. For example, the following program computes the application in the version V3_20.

```
let !f = g in
  let !x = u in
    !(f x).V3_20
```

Coeffect System

The calculus for LambdaVL is built on $\ell\mathcal{R}$PCF [3], an extension of the simply-typed lambda calculus that handles resources of types as coeffects.

The coeffect system is unfortunately not applicable for handling version in a class-based object-oriented programming language such as Java. The coeffect system requires a uniform type for a value across different versions. On the other hand, a version update on a Java class can change any part of the signature of a class. A version update may change a class' field, constructor, and method types. These kinds of changes cannot be handled by coeffects.

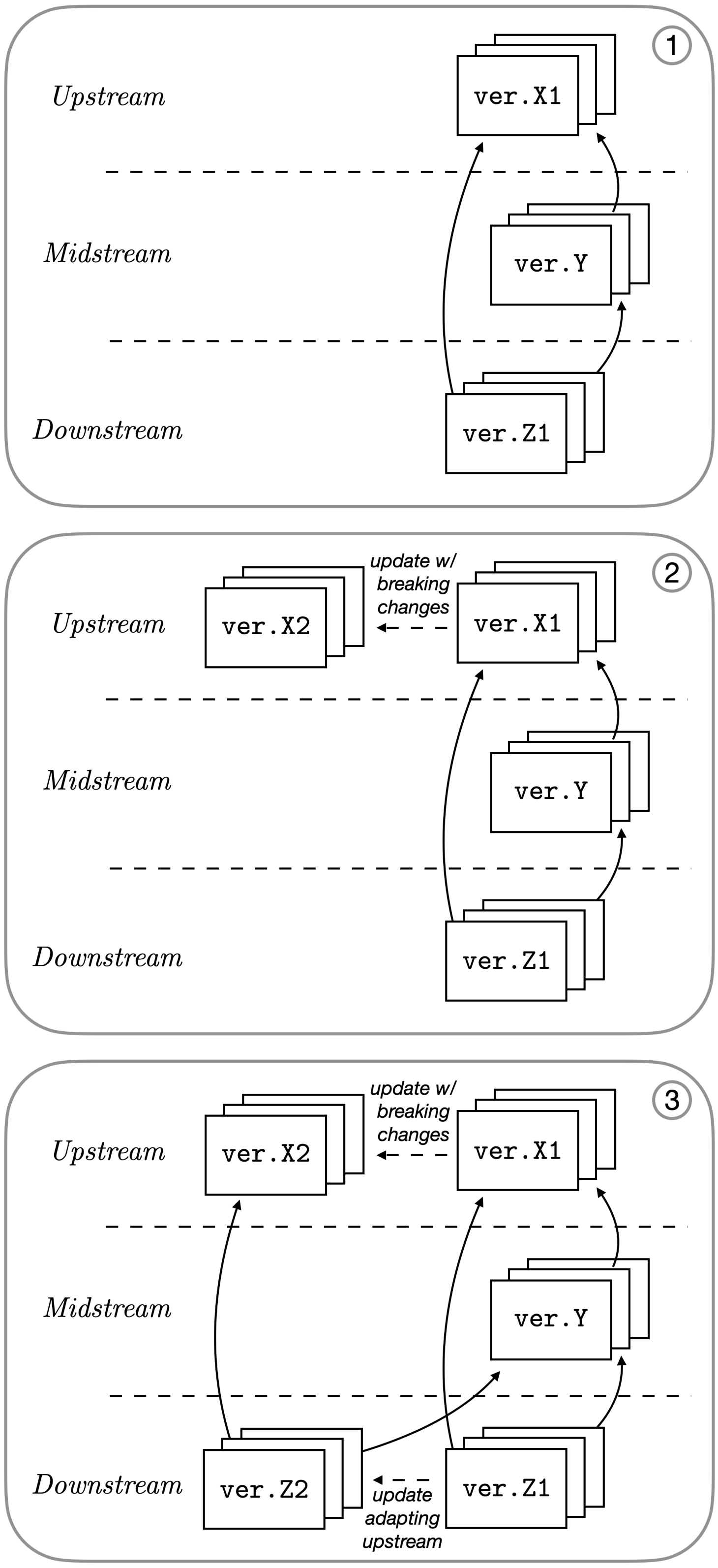


FIGURE 2.1: Dependency relation involving breaking changes

Chapter 3

Programming With Versions

We propose the concept of *programming with versions* to solve and avoid conflicts among multiple versions of a dependency. We apply the idea of versions as a resource of types in the framework of object-oriented programming. We implemented the concept in a language called BatakJava, a subset of Java with explicit version annotations. Programs in BatakJava are compiled into Java and can use Java classes.

The concept will be introduced using a running example shown by Figure 3.1. The example replaces the packages' content shown in Figure 2.1 with concrete classes. The upstream package contains the class `Point`, the midstream package contains depends on the upstream package and provides additional utility methods to handle `Point`, and the downstream package uses both the upstream and mid packages. The example involves both structural and behavioral changes.

3.1 Example

3.1.1 Java

Listing 1 shows the class `Point` before and after update. The Figure 3.1 distinguishes the before and after as version 1 and 2. It contains both structural and behavioral changes. The constructor for the class changed from `(float,float)` to `(float,int)`. In addition, the role of the fields change from x-y axis to radius and angle. Behaviorally, the method body of `distance(Point)` adapts the change of fields by invoking different methods. For brevity, the code will not include access modifier such as `public`, etc.

```java
// BEFORE UPDATE
package up;
class Point {
  float x; float y;
  Point(float x, float y) {
    this.x = x; this.y = y; }
  float getX() { return x; }
  float getY() { return y; }
  double distance(Point other) {
    return java.lang.Math.sqrt(
    /* uses getX()/getY() */); }
}
```

```java
// AFTER UPDATE
package up;
class Point {
  float r; int a;
  Point(float r, int a) {
    this.r = r; this.a = a; }
  float getR() { return r; }
  int getA() { return a; }
  double distance(Point other) {
    return java.lang.Math.sqrt(
    /* uses getR()/getAngle() */); }
}
```

LISTING 1: The class `Point` in the upstream package

The class PointEx and Utility in the midstream package are shown in Listing 2. As shown in Figure 3.1, both these classes depend on the upstream package before update. The method rev() and equals(Point,Point), the methods getX() and getY() that are only available before upstream's update are invoked.

```
package mid;
import up.*;
class PointEx extends Point {
  PointEx(float x, float y) {
    super(x,y); }
  Point rev() {
    return new Point(getY(),getX()); }
}
```

```
package mid;
import up.*;
class Utility {
  boolean equals(Point p1, Point p2) {
    return (p1.getX() == p2.getX())
    && (p1.getY() == p2.getY());
  }
}
```

LISTING 2: The classes in the midstream package

Lastly, the class Main is shown by Listing 3. Before update, the downstream package uses the old implementation Point in the method before(). After update, a new method after() is added where the object Point invokes getR() and also distance (Point) from the new implementation. At the same time, in the method before(), the class Main still uses the original Point through PointEx and Utility, In Java, this can still be compiled by using the new Point, but it will fail during runtime because it has two choose one particular version of Point. Selecting the old Point will cause error in the method after(). while selecting the new Point will cause error in the method before().

In this situation, Main's programmers have several options. Option one is to reverse its update of adding after() and wait until the midstream package adopts the new upstream. Option two is to forcefully rename its dependencies using some external tools.

```
// BEFORE UPDATE
package down;
import up.*; // upstream before update
import mid.*;
class Main {
  void before() {
    Point p1 = new Point(0,0);
    PointEx e = new PointEx(0,1);
    Point p2 = e.rev();
    double d = p1.distance(p2);
    boolean b = Utility.equals(p1,p2); }
}

// AFTER UPDATE
package down.*;
```

```
import up.*; // upstream after update
import mid.*;
public class Main {
  void before() {
    PointEx e1 = new PointEx(0,0);
    PointEx e2 = new PointEx(0,1);
    double d =
      e1.rev().distance(e2.rev());
    boolean b = Utility.equals(
      e1.rev(),e2.rev()); }
  void after() {
    Point p1 = new Point(1,0);
    Point p2 = new Point(1,90)
    float f = p1.getR();
    double d = p1.distance(p2); }
}
```

LISTING 3: The class Main in the downstream package

3.1.2 BatakJava

The code and compilation of the upstream package before update are shown by Listing 4. The first implementation of the upstream package is annotated with ver 1 on its package declaration. During compilation, the version annotation will propagate to each class declaration made within the package, attaching version onto each class.

The compiler also finds the appropriate version selection for every BatakJava type access in the program. In the BatakJava class `Point`, a version needs to be selected for each instance of `Point` itself. But, in the current context there is only version 1 of the class `Point`, so all instances of `Point` are replaced by the version specific class `Point_ver_1`. Compiling the class `Point` will result in two classes, `Point` and `Point_ver_1`. The generated class `Point` is used to distinguish BatakJava from Java classes, while `Point_ver_1` is the class `Point` with the specific version attached to it.

```
// SURFACE LANGUAGE
package up ver 1;
class Point {
  float x; float y;
  Point(float x, float y) {
    this.x = x; this.y = y; }
  float getX() { return x; }
  float getY() { return y; }
  double distance(Point other) {
    return java.lang.Math.sqrt(
    /* uses getX()/getY() */); }
}
// COMPILATION RESULT
package up;
```

```
class Point {
  boolean BATAKJAVACLASS; boolean VER_1;
}
class Point_ver_1 {
  float x; float y;
  Point_ver_1(float x, float y) {
    this.x = x; this.y = y; }
  float getX() { return x; }
  float getY() { return y; }
  double distance(Point_ver_1 other) {
    return java.lang.Math.sqrt(
    /* uses getX()/getY() */); }
}
```

LISTING 4: Compilation of the upstream package version 1

The code and compilation of the midstream package are shown by Listing 5. We annotate the implementation of the package with `ver 1`, treating it as the first version. The compilation of `PointEx` and `Utility` generate the class `PointEx`, `PointEx_ver_1` and `Utility`, `Utility_ver_1` respectively. The generated classes `PointEx` and `Utility` mark them as BatakJava classes, while `PointEx_ver_1` and `PointUtility_ver_1` are the version attached classes. Since there is only version 1 available for every involved class, the compilation replaces all instances of `Point`, `PointEx`, and `Utility` with `Point_ver_1`, `PointEx_ver_1`, and `Utility_ver_1` respectively. Beside from annotating the type access, methods that return BatakJava class are also annotated with version, as shown by the method `rev_ver_1` that returns the class `Point_ver_1`.

The code and compilation of the downstream package are shown by Listing 6. The implementation of the package is also annotated with `ver 1`. The compilation of `Main` generates the class `Main` and `Main_ver_1`. The generated class `Main` marks the class as a BatakJava class and `Test_ver_1` is the version attached class. Similar as in the upstream and midstream package, the compilation replaces all instances of `Point`, `PointEx`, and `Utility` with the only available version specific classes `Point_ver_1`, `PointEx_ver_1`, and `Utility_ver_1`.

At this point, the program written in BatakJava works identical to the Java program before the update, different only in the version annotation attached to each type access.

Next, the code and compilation of the updated upstream package that contain breaking changes are shown by Listing 7. To separate the compilation result from overwriting the older implementation (`Point_ver_1`), the package declaration is annotated differently with `ver 2`. The compilation will result in `Point` and `Point_ver_2`. The generated class `Point` updates the older class' field `VER_1` into `VER_2`, annotating it with the most recent version. All instances of `Point` in `Point_ver_2` are replaced by `Point_ver_2` because in this case, the context is limited only to version 2 of the class `Point`.

```
package mid ver 1;
import up.*;  // only ver.1 available
class PointEx extends Point {
  PointEx(float x, float y) {
    super(x,y); }
  Point rev() {
    return new Point(getY(),getX()); }
}
class Utility {
  boolean equals(Point p1, Point p2) {
    return (p1.getX() == p2.getX())
    && (p1.getY() == p2.getY());
  }
}

// COMPILATION RESULT
package mid;
import up.*; // only ver.1 available
```

```
class PointEx {
  boolean BATAKJAVACLASS; boolean VER_1;
}
class PointEx extends Point_ver_1 {
  PointEx_ver_1(float x, float y) {
    super(x,y); }
  Point_ver_1 reverse_ver_1() {
    return new Point_ver_1(
    getX(), getY()); }
}
class Utility {
  boolean BATAKJAVACLASS; boolean VER_1;
}
class Utility_ver_1 {
  boolean equals(
      Point_ver_1 p1, Point_ver_1 p2) {
    return (p1.getX() == p2.getX())
    && (p1.getY() == p2.getY()); }
}
```

LISTING 5: Compilation of the midstream package version 1

```
package down ver 1;
import up.*; // only ver.1 available
import mid.*;
class Main {
  void before() {
    Point p1 = new Point(0,0);
    PointEx e = new PointEx(0,1);
    Point p2 = e.rev();
    double d = p1.distance(p2);
    boolean b = Utility.equals(a,c); }
}

// COMPILATION RESULT
package down;
import up.*; // only ver.1 available
```

```
import mid.*;
class Main {
  boolean BATAKJAVACLASS; boolean VER_1;
}
class Main_ver_1 {
  void before() {
    Point_ver_1 p1 = new Point_ver_1(0,0);
    PointEx_ver_1 e =
      new PointEx_ver_1(0,1);
    Point_ver_1 p2 = e.rev();
    double d = p1.distance(p2);
    boolean b = Utility_ver_1.equals(a,c);
  }
}
```

LISTING 6: Compilation of the downstream package version 1

Lastly, the code and compilation of the updated downstream package are shown by Listing 8. The key feature of BatakJava is allowing different implementations of the same class to be used simultaneously, during compilation and runtime. In BatakJava, programmers are not expected to annotate the class with versions, therefore every `Point` are left as is, while the versions are inferred later during compilation. In this case, after compilation, `p1` in `after()` is assigned `ver_2` because it also invokes `getR()`. At the same time, `PointEx` and `Utility` in `before()` work properly because `Point_ver_1` used by these classes are also available.

```
package up ver 2;
class Point {
  float r; int a;
  Point(float r, int a) {
    this.r = r; this.a = a; }
  float getR() { return r; }
  int getA() { return a; }
  double distance(Point other) {
    return java.lang.Math.sqrt(
    /* uses getR(), getAngle() */); }
}
```

```
// COMPILATION RESULT
package up;
class Point {
  boolean BATAKJAVACLASS; boolean VER_2;
}
class Point_ver_2 {
  float r; int a;
  Point_ver_2(float r, int a) {
    this.r = r; this.a = a; }
  float getR() { return r; }
  int getA() { return a; }
  double distance(Point_ver_2 other) {
    return java.lang.Math.sqrt(
    /* uses getR(), getAngle() */); }
}
```

LISTING 7: Compilation of the upstream package version 2

```
package down ver 2;
import up.*; // ver.1 and 2 available
import mid.*;
class Main {
  void before() {
    PointEx e1 = new PointEx(1,0);
    PointEx e2 = new PointEx(0,0);
    double d = e1.rev()
      .distance(e2.rev());
    boolean b = Utility.equals(
      e1.rev(), e2.rev());
  }
  void after() {
    Point p1 = new Point(1,0);
    Point p2 = new Point(1,90);
    float f = p1.getR();
    double d = p1.distance(p2);
  }
}
```

```
// COMPILATION RESULT
package down;
import up.*; // ver.1 and 2 available
import mid.*;
class Main {
  boolean BATAKJAVACLASS; boolean VER_2;
}
public class Main_ver_2 {
  void before() {
    PointEx_ver_1 e1 =
      new PointEx_ver_1(1,0);
    PointEx_ver_1 e2 =
      new PointEx_ver_1(0,0);
    double d = e1.rev()
      .distance(e2.rev());
    boolean b = Utility_ver_1.equals(
      e1.rev_ver_1(), e2.rev_ver_1());
  }
  void after() {
    Point_ver_2 p1 =
      new Point_ver_2(1,0);
    Point_ver_2 p2 =
      new Point_ver_2(1,90);
    float f = p1.getR();
    double d = p1.distance(p2);
  }
}
```

LISTING 8: Compilation of the downstream package version 2

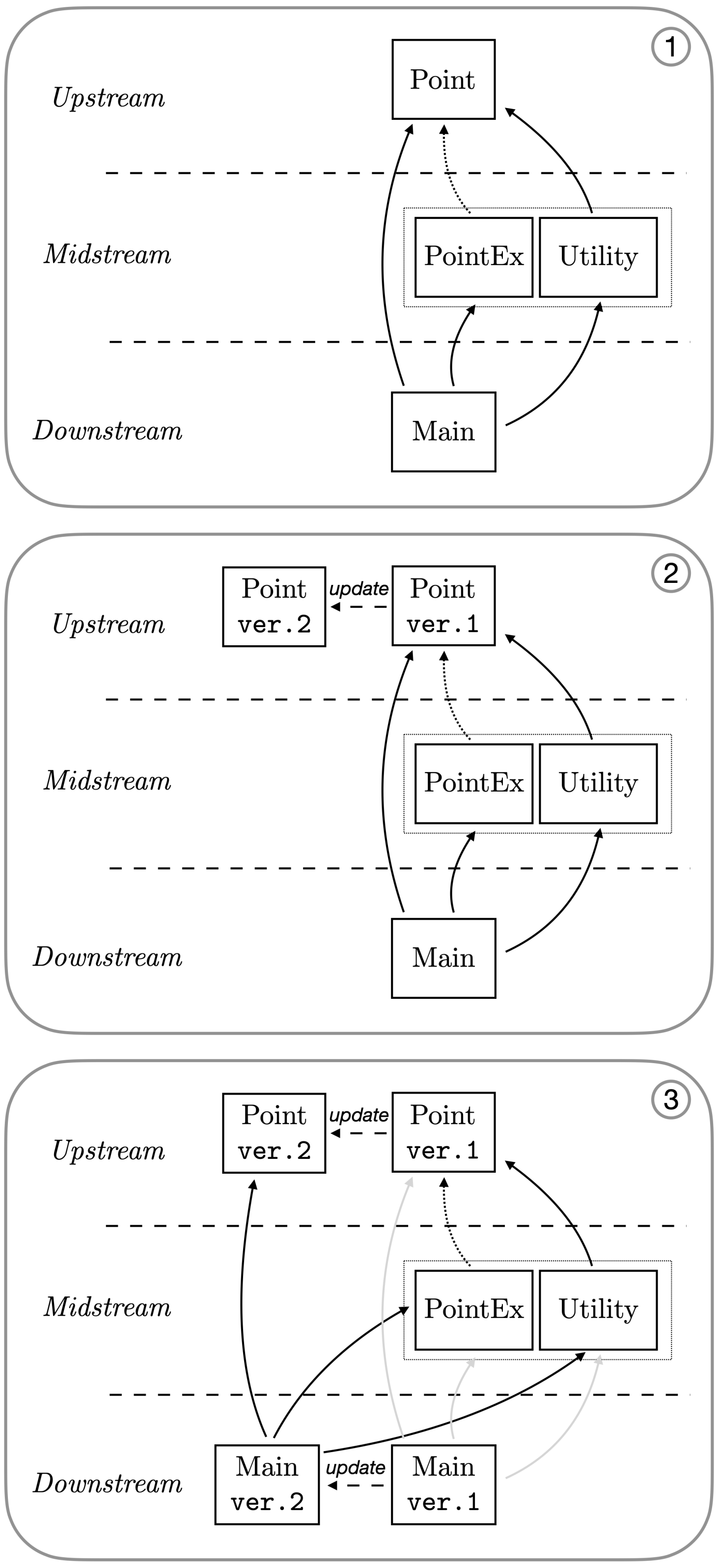


FIGURE 3.1: An example of the dependency relation with concrete classes

Chapter 4

Implementation

4.1 Overview

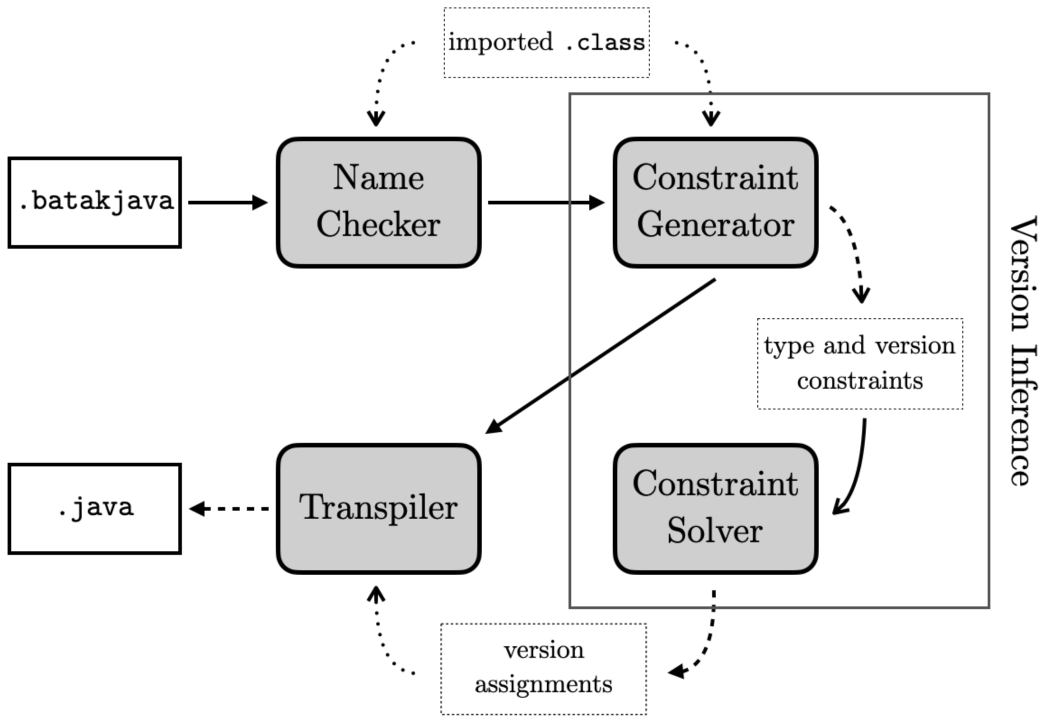


FIGURE 4.1: Overview of BatakJava transpiler

The overview of BatakJava compiler is shown by Figure 4.1. The compiler is implemented using ExtendJ [19], an extensible Java compiler. The compiler consists of a name checker, constraint generator, and a transpiler. First, the compiler name checks the given program together with the imported classes. Then, the program is passed on to the constraint generator. The generated constraints are passed to a separate constraint solver ChocoSolver [24]. The constraint generator and solver make up the version inference system of the compiler. Lastly, the program is compiled into Java using the solutions obtained from the solver.

The main feature of BatakJava is allowing the use of multiple versions of a class simultaneously in the same program. Programmers are not demanded to specify a version for each instance of class appearing in their programs, because BatakJava compilation infers the appropriate version for each class.

Class modification in the newer version is not restricted in any way. A newer version of a class may add new fields, constructors, or methods. It can also modify older fields, constructors, or methods by modifying their behaviors or signatures. There is also no restriction to remove older fields, constructors, or methods.

The implementation of the compiler can be found in https://www.github.com/ansharlubis/batakjava.

4.2 Syntax

The syntax of BatakJava is a subset of Java with several extensions. BatakJava extends Java's package and import declarations. On the other hand, it puts limitations on class body declarations and local variable declarations.

In the current implementation, versions only apply to class declarations. Other classes such as abstract classes, interfaces, and exception classes are not supported in BatakJava.

4.2.1 Extensions

Package Declaration

Package declarations in BatakJava require explicit versioning. A `.batakjava` file that does not specify version number in the package declaration will result in compile error. The grammar for package declarations replaces Java's grammar.

PackageDeclaration :
  `package` *PackageName* `ver` *IntegerLiteral* `;`

The other non-terminals and terminals, such as *PackageName* and *IntegerLiteral* follow the Java Language Specification [8]. The version annotation is still limited to integer number, starting from 1.

An example of a package declaration in BatakJava is shown below.

```
package up ver 1;
```

The above snippet denotes that the declarations made within the following compilation unit would belong to the version 1 of package `up`. This versioning will be applied to the class declared within the compilation unit, allowing the system to generate the Java code properly.

Import Declaration

BatakJava changes the import declarations of Java. Java allows two types of import declarations, *single-type-import declarations* and *type-import-on-demand declarations*. The former imports a single type and the latter imports all the `public` types of a named package as needed. However, BatakJava limits import declarations to type-import-on-demand declarations.

To explain the reason, let's reexamine the Figure 2.1 where `Point` version 1 and 2 are available. Suppose that we declare the following single-type-import declaration.

```
import up.Point;
```

During the compilation, the class `Point` within the program will eventually be transpiled into either `Point_ver_1` or `Point_ver_2`. This means that the compilation will have to compile the import declaration as well, to make sure that the resulting Java code be compiled by a Java compiler. Yet, by only allowing type-import-on-demand declarations, the import declarations can be left untouched.

In addition to the usual import declarations, BatakJava also provides the syntax to allow programmers to specify which version of the imported packages they wish to prioritize. This is provided because a program in BatakJava may involve multiple versions of packages simultaneously. BatakJava replaces Java's rule for import declarations. The grammar is given below.

ImportDeclaration :
  TypeImportOnDemandDeclaration
  TypeImportOnDemandDeclarationWithVersion
TypeImportOnDemandDeclaration :
  `import` *Name* `. * ;`
TypeImportOnDemandDeclarationWithVersion :
  `import` *Name* `. * prioritizes ver` *IntegerLiteral* `;`

Supposing that the class `Point` belongs to the package `up.*`, an example of an import declaration with version priority in BatakJava is shown below.

```
import up.* prioritizes ver 1;

class Test {
  static void main(String[] args) { Point p = new Point(0,0); }
}
```

The annotation `prioritizes ver 1` in the above snippet implies that if possible, the programmer prefers that the object `new Point(0,0)` is instantiated with version 2 of the class `Point`. This is applicable when multiple solutions for version inference are obtained. In this way, programmers still have control over the version of imported packages used within the compilation unit. The mechanism of `prioritizes` will be explained in detail in Section 4.5.

Type Instances

BatakJava allows users to specify version for each type instance made within program. This is done using the annotation `#n`, where `n` denotes a version number. An example is shown below.

```
Point#1 p = new Point#1(0,0);
```

The object instantiation `new Point#1(0,0)` denotes instantiation for class `Point` version 1, similarly the type instance `Point1` on the left hand side denotes the class `Point` version 1.

4.2.2 Limitations

Member Declaration

BatakJava limits the possible member declarations made inside a class body declaration, in which nested class or interface declarations are not allowed. The rule for member declaration in BatakJava is shown below.

```
MemberDecl:
    MethodOrFieldDecl
    void Identifier VoidMethodDeclaratorRest
    Identifier ConstructorDeclaratorRest
```

Local Variable Declaration

In Java, local variable declarations are allowed to declare one or more local variable names, such as shown below.

```
Point p1, p2;
```

However, due to the fact that in BatakJava, both `p1` and `p2` may have different versions, to simplify compilation, local variable declarations can only contain one variable declarator. The local variable declaration rule in BatakJava is shown below.

```
LocalVariableDeclaration:
    Type VariableDeclarator
```

Therefore, in BatakJava, the above example would have to be separated into two different variable declarations.

```
Point p1; Point p2;
```

4.3 Name Checking

Name checks are performed on type declarations, type instance, and import declarations with versions. In BatakJava, static name check for method and field accessibilty are performed during version inference phase.

4.3.1 Type Declaration and Access

This checks whether there is any duplicate type declaration made within the program and that each declared type access points to a type declaration that can be found within the program. The check is done to ensure that the constraint generation does not attempt to generate constraints for problematic types.

4.3.2 Import Declaration

This checks whether there is any duplicate import declaration with different versions as priority.

4.4 Version Inference

The inference aims to find the correct version assignment for each BatakJava expression declared inside the program so that the whole program passes Java's type check after being transpiled using the version assignment. A program consists of multiple compilation units and a compilation unit can consist of multiple class declarations. Therefore, the constraints are not generated and inferred per class declaration, but generated from all to be compiled class declarations and joined together to be inferred. We took the idea from [20] that introduced approach to infer types in untyped object-oriented programs with inheritance.

4.4.1 Variable Assignment

Solving version for each class instance inside the program requires the compiler to also account for the type of each expression as it may differ depending on the assigned version, because BatakJava allows a newer version of a class to change the type of a field and the return types of a method. Listing 9 shows an update involving field and method's return type changes.

Listing 9 shows an update involving breaking changes. The update changes the field and method's return type. Listing 10 shows a package using both version 1 and 2 of up. If the inference only accounts for accessible versions, the inference will decide that both version 1 and 2 are applicable to the variable p, when in fact the type changes. To handle this, we also generate constraints that ensure the type between the variable i and p.getA() are equal.

```
package up ver 1;
public class DifPoint {
  public int a;
  public DifPoint(int a) {
    this.a = a; }
  public int getA() { return a; }
}
```

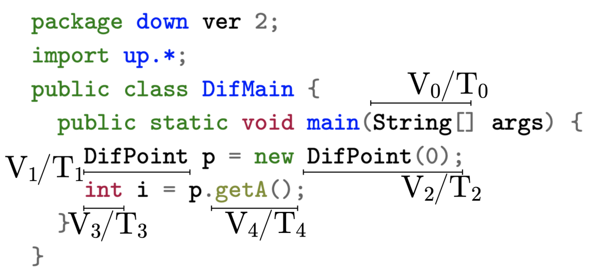
```
package up ver 2;
public class DifPoint {
  public float a;
  public DifPoint(float a) {
    this.a = a; }
  public float getA() { return a; }
}
```

LISTING 9: Update with a different signature

```
package down ver 2;
import up.*;
public class DifMain {
  public static void main(String[] args) {
    DifPoint p = new DifPoint(0);
    int i = p.getA();
  }
}
```

LISTING 10: Using multiple versions with different signatures

To do so, expressions in the program are assigned two fresh variables, a version variable V_n and a type variable T_n, where n is a natural number that starts from 0. As an example, the variable assignment for Listing 10 is shown by Listing 11. In this example, fresh variables are assigned to the type instances, `String[]` of the argument `args`, `DifPoint` of the variable declaration `DifPoint p`, `DifPoint` of the object instantiation `new DifPoint(0)`, and `int` of the variable declaration `int i`. Variables are also assigned to method invocation, `getA()` on `p`. Version and type variable of method access such as `getA()` refer to the method's return type. For the local variable `p`, it would refer to the type `DifPoint` in the local variable declaration `DifPoint p = ..` for its version and type variable.



```java
package down ver 2;
import up.*;
public class DifMain {
  public static void main(String[] args) {
    DifPoint p = new DifPoint(0);
    int i = p.getA();
  }
}
```

LISTING 11: An example of variable assignment

From here on, we will use the notation $V_{var}(\texttt{e})$ and $T_{var}(\texttt{e})$ to refer to an expression `e`'s version and type variable, respectively.

4.4.2 Constraint

A constraint C can be:

1. A *version constraint* $V_{var}(\texttt{e}) \doteq n$, where the version variable of expression `e` is mapped to the version number n.
2. A *type constraint* $T_{var}(\texttt{e}) \doteq T$, where the type variable of expression `e` is mapped to the type T.
3. A *conjunction constraint* $C_1 \wedge C_2$, denoting the constraint C_1 **and** C_2.
4. A *disjunction constraint* $C_1 \vee C_2$, denoting the constraint C_1 **or** C_2.
5. An *unknown constraint*, generated in cases where no applicable candidate mapping can be found for an expression. If an unknown constraint is generated, it can be understood that the expression is not typable.

Types to Constraints

Constraints for type instances are always generated in a pair of version and type constraints joined together as a conjunction constraint. When the inference found a BatakJava class with multiple possible candidate versions, it will generate a disjunction constraint consisting of several conjunction constraints that represent each possible candidate version.

As an example, the type instance `int` will generate the following constraint $V_{var}(\texttt{int}) \doteq 0 \wedge T_{var}(\texttt{int}) \doteq int$. Given two versions exist (1 and 2), a type instance of a BatakJava class such as `Point` will generate the following constraint. $(V_{var}(\texttt{Point}) \doteq 1 \wedge T_{var}(\texttt{Point}) \doteq Point) \vee (T_{var}(\texttt{Point}) \doteq 2 \wedge T_{var}(\texttt{Point}) \doteq Point)$.

To simplify explanation, in the context of constraint generation, we will use the following notations. $cstr(\texttt{e})$ is the constraint generated from `e`. Given a class `t`, $ver(\texttt{t})$ returns the set of version the class `t` is available in.

Constraints to Types

For some expressions, generating constraints require building new constraints based on other expressions' constraints. Take the expression `e.m()`. The constraints generated by the method invocation `m()` will depend on the type of `e` which is understood here as the type mapped in the constraints generated by `e`.

To help explain the constraint generation, we use the following notation. $\Gamma(e)$ returns the set of version and type pair mapping for the expression `e` in the constraint $cstr(\texttt{e})$. As an example, given the constraint $(V_{var}(\texttt{Point}) \doteq 1 \wedge T_{var}(\texttt{Point}) \doteq Point) \vee (T_{var}(\texttt{Point}) \doteq 2 \wedge T_{var}(\texttt{Point}) \doteq Point)$, we can find the candidate types for the type instance `Point` through $\Gamma(\texttt{Point})$. In this case, it would return $\{(1, Point), (2, Point)\}$.

4.4.3 Constraint Generation

Literals

Literals that generate constraints include integer, floating point, character, string, and boolean literal. Given a literal `e` of base type `T`, the expression would generate the following constraint.

$$cstr(\texttt{e}) = (V_{var}(\texttt{e}) \doteq 0) \wedge (T_{var}(\texttt{e}) \doteq T)$$

Type Instances

A type instance is any mention of a class or type name inside the program. Constraints are generated from the following type instances.

In case of base types and Java classes, the version number assigned is always 0. Given a type instance `T`, it will generate the following constraint.

$$cstr(\texttt{T}) = (V_{var}(\texttt{T}) \doteq 0) \wedge (T_{var}(\texttt{T}) \doteq T)$$

In case of BatakJava classes, the type instances will generate disjunction constraint based on its accessible versions. Given a type instance `T` with type t and $ver(\texttt{t}) = \{v_1, .., v_n\}$, it will generate the following constraint.

$$cstr(\texttt{T}) = (V_{var}(\texttt{T}) \doteq v_1) \wedge (T_{var}(\texttt{T}) \doteq t) \vee \ldots \vee (V_{var}(\texttt{T}) \doteq v_n) \wedge (T_{var}(\texttt{T}) \doteq t)$$

Since name check has been performed before this to confirm the accessibility of each type and class, a type instance will never generate an unknown constraint.

A `super` instance will generate constraints based on the enclosing class declaration's superclass.

Example (1). The type instance `int` will generate

$$cstr(\texttt{int}) = (V_{var}(\texttt{int}) \doteq 0) \wedge (T_{var}(\texttt{int}) \doteq int)$$

Example (2). As another example, the type instance `Point`, where $ver(\texttt{Point}) = \{1, 2\}$, will generate the following constraint.

$$\begin{aligned} cstr(\texttt{Point}) = \big(& (V_{var}(\texttt{Point}) \doteq 1) \wedge (T_{var}(\texttt{Point}) \doteq Point)\big) \vee \\ & (V_{var}(\texttt{Point}) \doteq 2) \wedge (T_{var}(\texttt{Point}) \doteq Point) \end{aligned}$$

Object Instantiation

Suppose we have an object instantiation `new C(a1,...,aj)`. The compiler searches for applicable constructors among Γ(`C`). Each possible constructor is then checked against the given arguments $\mathtt{a}_1,\ldots,\mathtt{a}_j$. Constraints are generated based on the applicable constructor and arguments.

Example. Consider the object instantation `new DifPoint(0)` in Listing 10. First, the type instance `DifPoint` will generate the following constraint *cstr*(`DifPoint`).

$$\begin{aligned} cstr(\texttt{DifPoint}) = {} & \big((V_{var}(\texttt{DifPoint}) \doteq 1) \land (T_{var}(\texttt{DifPoint}) \doteq \texttt{DifPoint})\big) \lor \\ & \big((V_{var}(\texttt{DifPoint}) \doteq 2) \land (T_{var}(\texttt{DifPoint}) \doteq \texttt{DifPoint})\big) \end{aligned}$$

The compiler then search for applicable constructors in each possible type for `DifPoint`. Referring back to Listing 9, the constructor's signature is `(int)` and `(float)` in version 1 and 2, respectively.

Next the given argument `0` generates the following constraint.

$$cstr(\texttt{0}) = (V_{var}(\texttt{0}) \doteq 0) \land (T_{var}(\texttt{0}) \doteq \texttt{int})$$

This denotes that the possible type for `0` is `int`. Since `int` is a subtype for both `int` and `float`, both constructors are applicable.

Therefore, this object instantatiation will generate the following constraint.

$$\begin{aligned} & cstr(\texttt{new DifPoint(0)}) = \\ & \big((V_{var}(\texttt{DifPoint}) \doteq 1) \land (T_{var}(\texttt{DifPoint}) \doteq \texttt{DifPoint}) \\ & \quad \land (V_{var}(\texttt{0}) \doteq 0) \land (T_{var}(\texttt{0}) \doteq int)\big) \lor \\ & \big((V_{var}(\texttt{DifPoint}) \doteq 2) \land (T_{var}(\texttt{DifPoint}) \doteq \texttt{DifPoint}) \\ & \quad \land (V_{var}(\texttt{0}) \doteq 0) \land (T_{var}(\texttt{0}) \doteq int)\big) \lor \end{aligned}$$

Super Constructor Access

A super constructor access `super(a1,...,aj)` made inside a class declaration `class T extends U` generates constraint in a similar manner as object instantation, where the instantiated type `C` is replaced by the superclass type instance `U`.

Example. Consider the super constructor `super(x,y)` in the class `PointEx` in Listing 5. The enclosing class `PointEx` extends the class `Point`, which is available in version 1 and 2 (Listing 4, 7). The constructor's signature for `Point` is `(float,float)` and `(float,int)` in version 1 and 2, respectively. Given that the arguments `x` and `y` are both of type `float`, the constraint can only be built using the constructor in `Point` version 1. Therefore, this super constructor access will generate the following constraint.

$$\begin{aligned} & cstr(\texttt{super(x,y)}) = \\ & \big((V_{var}(\texttt{Point}_1) \doteq 1) \land (T_{var}(\texttt{Point}_1) \doteq Point) \\ & \quad \land (V_{var}(\texttt{x}) \doteq 0) \land (T_{var}(\texttt{x}) \doteq float) \\ & \quad \land (V_{var}(\texttt{y}) \doteq 0) \land (T_{var}(\texttt{y}) \doteq float)\big) \end{aligned}$$

Where `Point` denotes the superclass on the class header `class PointEx extends` $\texttt{Point}_1$.

Method Invocation

The constraint generation for method invocation depends on whether the access is qualified or not. A qualified method invocation occurs after a "." token, such as the method invocation `getA()` on `new Point(0,0).getA()`. On the other hand, a non-qualified method invocation does not occur after a "." token, such as `getX()` in Listing 5.

Non-qualified. In case of the non-qualified method invocation $\mathtt{m(a_1,...,a_j)}$, the constraint generation depends on the enclosing class declaration. Let's suppose that the enclosing class declaration `class T extends U` belongs to version n.

The compiler searches for applicable methods in the enclosing class declaration `T`, then traces up to the ancestor classes. If an ancestor class `S` happens to be a BatakJava class with multiple versions, the compiler branches the search to each version of the ancestor class `S`. Constraints are then generated based on the applicable constructors and the enclosing classes where they belong to. If no applicable method is found, then an unknown constraint is generated, prompting the compiler to stop.

Qualified. In case of the method invocation $\mathtt{e.m(a_1,..,a}_j\mathtt{)}$, the constraint generation for $\mathtt{m(a_1,...,a_j)}$ depends on $cstr(\mathtt{e})$. The qualifier expression `e` can be an object instantiation, another method invocation, a field access, or a super access.

The compiler searches for applicable methods in the classes belonging to $\Gamma(\mathtt{e})$, also tracing the search up to the ancestor classes.

Example (1). Consider the qualified method invocation `p.getA()` in Listing 10.

The constraints generated by the receiver object `p` are as follow.

$$\begin{aligned} cstr(\mathtt{p}) = \big((&V_{var}(\mathtt{p}) \doteq 1) \land (T_{var}(\mathtt{p}) \doteq \mathtt{DifPoint}) \\ &(V_{var}(\mathtt{p}) \doteq 2) \land (T_{var}(\mathtt{p}) \doteq \mathtt{DifPoint})\big) \end{aligned}$$

Then, we search for applicable method declarations in both `DifPoint` ver.1 and 2. We find `getA()` with signature `() -> int` and `() -> float` in ver.1 and 2, respectively. Given that there are no arguments to check, both methods are considered applicable. Hence, the following constraints are added to `cstr(p.getA())`.

$$\begin{aligned} &\big((V_{var}(\mathtt{p}) \doteq 1) \land (T_{var}(\mathtt{p}) \doteq \mathtt{DifPoint}) \\ &\quad \land (V_{var}(\mathtt{getA()}) \doteq 0) \land (T_{var}(\mathtt{getA()}) \doteq int)\big) \lor \\ &\big((V_{var}(\mathtt{p}) \doteq 2) \land (T_{var}(\mathtt{p}) \doteq \mathtt{DifPoint}) \\ &\quad \land (V_{var}(\mathtt{getA()} \doteq 0) \land (T_{var}(\mathtt{getA()}) \doteq float)\big) \end{aligned}$$

Note that following the return type for each version, $T_{var}(\mathtt{getA()})$ is mapped to `int` in ver.1 and `float` in ver.2.

The search continues to `DifPoint`'s ancestor class, which is `Object`. Here, the method `getA()` is not found, thus no additional constraints are added to `cstr(p. getA())`. Additionally, since the search has reached `Object`, the search for applicable methods ends. It means that `cstr(p.getA())` is equal to the constraints above.

Example (2). Consider the non-qualified method invocation `getX()` in Listing 5.

First, the applicable method is searched in the enclosing class declaration `PointEx`, where no method candidate can be found. Next, it searches for applicable methods in the superclass `Point`. Due to the fact that `Point` is a BatakJava class, then the search branches to both available versions of `Point`. The applicable method `getX()` with signature `() -> float` is then found in `Point` ver.1. The following constraint is added to the cstr(getX()).

$$(V_{var}(\texttt{getX()}) \doteq 0) \wedge (T_{var}(\texttt{p}) \doteq float) \wedge$$
$$(V_{var}(\texttt{Point}) \doteq 2) \wedge (T_{var}(\texttt{Point}) \doteq Point)$$

Here, `Point` denotes the superclass instance in `class PointEx extends Point`.

From both `Point` ver.1 and 2, the method search continues to the superclass of each class, both happen to be `Object`. In class `Object`, no applicable method is found and the search ends.

Field and Variable Access

The constraint generation for field and variable access also depends on whether the access is qualified or not. Unlike method invocation's constraint generation however, once an applicable field or variable is found, be it in the enclosing method body, class declaration, or the superclass, the search stops and does not attempt to find any other applicable fields any further.

Non-qualified. In case of the access `f`, we first search for a corresponding local variable declaration. If a local variable declaration `T f;` or `T f = e;` is found, then $cstr(\texttt{f})$ is generated from `T`.

If no local variable declaration is found, then we search for an applicable field declaration in the enclosing class declaration. If a field declaration `T f;` or `T f = e;` is found, then a $cstr(\texttt{f})$ is generated from `T`.

Again, if no applicable field is found in the enclosing class declaration, we recursively search for an applicable field in the ancestor class. If the ancestor class is a class with multiple versions, we search the field in each version of the ancestor class. If the field is found in a specific version ancestor class, then the $cstr(\texttt{f})$ will also include a constraint on the ancestor class.

Qualified. Constraint generation for `e.f` is identical to method invocations, only that unlike method invocations, we do not need to generate constraints for arguments.

Example. Suppose we add a field access `p1.r` into the class `Main` in Listing 8. The compiler searches for an applicable field in each candidate type in $\Gamma(\texttt{p1}) = \{(1, Point), (2, Point)\}$. Going through both `Point` ver.1 and 2, we find that the field `r` is only available in ver.2. Therefore, this type access generates the following constraints.

$$cstr(\texttt{a1.r}) = (V_{var}(\texttt{p1}) \doteq 2) \wedge (T_{var}(\texttt{p1}) \doteq Point) \wedge$$
$$(V_{var}(\texttt{r}) \doteq 0) \wedge (T_{var}(\texttt{r}) \doteq \texttt{float})$$

Local Variable Declaration

Suppose that the declaration type is `T` and the variable is initialized by `e`. The constraints generated are the pairings of the declaration type `T` and expression `e` that fulfill the subtyping relation.

Example (1). Take the variable declaration `s`, where `s` is $\texttt{Point}_1$ `p = new` $\texttt{Point}_2$`(0, 0)` (Listing 4, 7). The subscripts are used here to distinguish between the declaration type and the object instantiation.

Here, we have

$$\begin{aligned}\Gamma(\texttt{new Point}_2\texttt{(0,0)}) &= \{(1, Point), (2, Point)\}\\ \Gamma(\texttt{Point}_1) &= \{(1, Point), (2, Point)\}\end{aligned}$$

Given that subtyping relation is reflexive and that different versions of a class do not have any subtyping relation between them, we obtain the following constraints.

$$\begin{aligned}cstr(\texttt{s}) = \big(&(V_{var}(\texttt{Point}_1) \doteq 1) \wedge (T_{var}(\texttt{Point}_1) \doteq Point) \wedge\\ &(B_{var}(\texttt{new Point}_2\texttt{(0,0)}) \doteq 1) \wedge (T_{var}(\texttt{new Point}_2\texttt{(0,0)}) \doteq Point)\big) \vee\\ \big(&(V_{var}(\texttt{Point}_1) \doteq 2) \wedge (T_{var}(\texttt{Point}_1) \doteq Point) \wedge\\ &(V_{var}(\texttt{new Point}_2\texttt{(0,0)}) \doteq 2) \wedge (T_{var}(\texttt{new Point}_2\texttt{(0,0)}) \doteq Point)\big)\end{aligned}$$

Example (2). Suppose that the declaration `s` is `Point`$_1$ `p = new PointEx(0,0)`.
Here, we have

$$\begin{aligned}\Gamma(\texttt{new PointEx(0)}) &= \{(1, PointEx)\}\\ \Gamma(\texttt{Point}_1) &= \{(1, Point), (2, Point)\}\end{aligned}$$

There is one option for the left hand side, the class `PointEx` ver.1. Although we know that `PointEx` ver.1 is a subclass of `Point`, given that the superclass for `PointEx` ver.1 is not yet determined, constraints for both possibilites are generated. Also, the superclass instance in `PointEx`'s class header is also included in the constraints.

Suppose `PointEx`'s class header is `class PointEx extends Point`$_2$ `{...}`, the variable declaration will generate the following constraints.

$$\begin{aligned}cstr(\texttt{s}) = \big(&(V_{var}(\texttt{Point}_1) \doteq 1) \wedge (T_{var}(\texttt{Point}_1) \doteq Point) \wedge\\ &(V_{var}(\texttt{PointEx}) \doteq 1) \wedge (T_{var}(\texttt{new PointEx(0,0)}) \doteq PointEx) \wedge\\ &(V_{var}(\texttt{Point}_2) \doteq 1) \wedge (T_{var}(\texttt{Point}_2) \doteq Point)\big) \vee\\ \big(&(V_{var}(\texttt{Point}_1) \doteq 2) \wedge (T_{var}(\texttt{Point}_1) \doteq Point) \wedge\\ &(V_{var}(\texttt{PointEx}) \doteq 1) \wedge (T_{var}(\texttt{new PointEx(0,0)}) \doteq PointEx) \wedge\\ &(V_{var}(\texttt{Point}_2) \doteq 2) \wedge (T_{var}(\texttt{Point}_2) \doteq Point)\big)\end{aligned}$$

Return Statement

A return statement returns control to the invoker of a method. Constraint is generated from the return statement contained in a method declaration's body.

Constraint generation for return statement works similarly as a variable declaration, but instead of the declaration type, we check subtyping relation between the return statement's expression against the enclosing method's return type.

4.4.4 Constraint Solving

Given the constraints $C_1, \ldots, C_n$ generated from the expressions and statements in the program, if any of the constraints is an empty constraint, the compilation will fail. If none of the constraints is an empty constraint, each constraint is converted into the class `Constraint` provided by ChocoSolver and joined together by conjunction. Next, the converted constraints are passed into the library's solver.

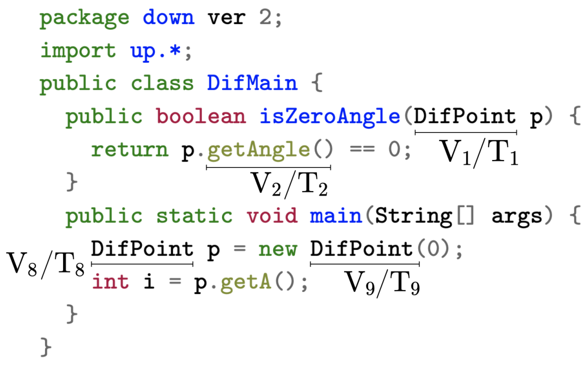
The solver returns a set of *solution*. A solution A is a mapping from version variables to version numbers and type variables to fully qualified type names.

$$A = [V_1 \doteq v_1, T_1 \doteq t_1, \ldots, V_n \doteq v_n, T_n \doteq t_n]$$

Since the solution is a mapping, for each V_i and T_i, $A(V_i) = v_i$ and $A(T_i) = t_i$.

Example. Listing 12 shows the class `DifMain` with an added method. The figure highlights the version and type variables with variations, which are V_1/T_1, V_2/T_2, V_8/T_8, and V_9/T_9. The other variables not mentioned are mapped to primitive types or Java classes, so their version variables are mapped to 0.

```
package down ver 2;
import up.*;
public class DifMain {
  public boolean isZeroAngle(DifPoint p) {
    return p.getA() == 0;
  }
  public static void main(String[] args) {
    DifPoint p = new DifPoint(0);
    int i = p.getA();
  }
}
```



```
package down ver 2;
import up.*;
public class DifMain {
  public boolean isZeroAngle(DifPoint p) {
    return p.getAngle() == 0;
  }
  public static void main(String[] args) {
    DifPoint p = new DifPoint(0);
    int i = p.getA();
  }
}
```

LISTING 12: Class `DifMain` with an added method

The solver returns two solutions A_1 and A_2 shown below.

$$\begin{aligned} A_1 = [&V_1 \doteq 2,\ T_1 \doteq \texttt{DifPoint},\ V_2 \doteq 0,\ T_2 \doteq \texttt{float},\\ &V_8 \doteq 1,\ T_2 \doteq \texttt{DifPoint},\ V_9 \doteq 1,\ T_9 \doteq \texttt{DifPoint}, \ldots]\\ A_2 = [&V_1 \doteq 1,\ T_1 \doteq \texttt{DifPoint},\ V_2 \doteq 0,\ T_2 \doteq \texttt{int},\\ &V_8 \doteq 1,\ T_2 \doteq \texttt{DifPoint},\ V_9 \doteq 1,\ T_9 \doteq \texttt{DifPoint}, \ldots] \end{aligned}$$

We can confirm how each solution is applicable. The relation expression `p.getA() == 0f` works with both implementation of the `getA()`. In version 1, it returns `int`, so in solution A_2 we have $V_1 \doteq 1$ and $T_2 \doteq$ `int`. In version 2, the method `getA()` returns `float`, therefore in solution A_1, we have $V_1 \doteq 2$ and $T_2 \doteq$ `float`. Differently, `int i = p.getA();` has an identical solution in both solutions. This is because the declaration can only hold if `getA()` returns an `int`, which is only available in version 1. Hence, the set consists of two solutions.

4.5 Code Generation

4.5.1 Preparation

To explain the code generation, we need to introduce several notations.

Given the solution $A = [V_1 \doteq v_1, T_1 \doteq t_1, \dots, V_n \doteq v_n, T_n \doteq t_n]$. $vdom(A) = \{V_1, \dots, V_n\}$ denotes the set of version variables and $tdom(A) = \{T_1, \dots, T_n\}$ denotes the set of type variables in the domain of the solution.

Given two solutions A_1 and A_2, the solution A_1 is *larger* than A_2 if for each $V_i \in vdom(A_1)$, $A_1(V_i) \geq A_2(V_i)$ and $A_1(T_i) = A_2(T_i)$. A larger solution can be understood as the solution that maps each class to a type with the same or newer (larger) version number.

4.5.2 Transpilation

The transpilation for a class `T` in version n produces two classes, an overview class `T` marking it as a BatakJava class and a versioned class `T_ver_n` attached with the version number n.

Overview Class

An overview class is necessary to distinguish between a BatakJava and Java class. By distinguishing between two, the compiler can properly generate constraints.

Listing 13 shows the structure of an overview class. The package declaration contains the package name. The class header keeps the class name and superclass. The body of the class contains two fields. The field `BATAKJAVACLASS` marks the class as a BatakJava class and the field `VER_`n marks the largest version the class is defined in.

Example. The overview class of `DifMain` (Listing 12) is shown by Listing 14. Version annotaiton is removed from the package declaration. Additionally, since the class `DifMain` compiled belongs to version 2 of the package, the field `VER_2` is also added.

```
package pkg.name;
public class T extends U {
  public boolean BATAKJAVACLASS;
  public boolean VER_n;
}
```

LISTING 13: Structure of an overview class

```
package down;
public class DifMain {
  public boolean BATAKJAVACLASS;
  public boolean VER_2;
}
```

LISTING 14: Overview class of `DifMain`

Versioned Class

A versioned class is generated from the solutions obtained from the inference.

Example. Consider the class `DifMain` in Listing 12. We will show the code generation for this class. Note that the solutions for this program is as follows.

$$\begin{aligned} A_1 = [&V_1 \doteq 2,\ T_1 \doteq \texttt{DifPoint},\ V_2 \doteq 0,\ T_2 \doteq \texttt{float},\\ &V_8 \doteq 1,\ T_2 \doteq \texttt{DifPoint},\ V_9 \doteq 1,\ T_9 \doteq \texttt{DifPoint}, \ldots]\\ A_2 = [&V_1 \doteq 1,\ T_1 \doteq \texttt{DifPoint},\ V_2 \doteq 0,\ T_2 \doteq \texttt{int},\\ &V_8 \doteq 1,\ T_2 \doteq \texttt{DifPoint},\ V_9 \doteq 1,\ T_9 \doteq \texttt{DifPoint}, \ldots] \end{aligned}$$

The transpilation removes the version annotation from the package declaration, while import declarations do not change.

```
package down;
import up.*;
```

Then, next the transpilation attaches the version number of the package declaration to the class header.

```
public class DifMain_ver_2 { ... }
```

Afterwards, the transpilation works on each class body declaration based on the given solutions. In this example, we have two method declarations. Consider the first method `boolean isZeroAngle(DifPoint p)`. To transpile method declarations, we need to filter the inference solutions based on the method's signature and return type. Consider the argument `DifPoint p` whose version/type variable are V_1/T_1. In the two given solutions, V_1 is mapped to a different version number, to version 1 in A_2 and version 2 in A_1. In this case, two methods will be generated, using each solution.

```
public boolean isZeroAngle(DifPoint_ver_1 p) {...}
public boolean isZeroAngle(DifPoint_ver_2 p) {...}
```

The first and second method are generated from A_1 and A_2 respectively. Then, the method body for each method is simply generated by attaching version according to solution used. In the current example, the method body does not declare or return any BatakJava class so there is no change. Therefore, the transpilation of the method will generate these two methods.

```
public boolean isZeroAngle(DifPoint_ver_1 p) {
  return p.getA() == 0f;
}
public boolean isZeroAngle(DifPoint_ver_2 p) {
  return p.getA() == 0f;
}
```

Next, the transpilation for the other method `void main(String[] args)`. In both given solutions, the version and type variable of the method's arguments and return type remain the same, meaning that only one method can be generated. To choose between the two solutions, the current implementation chooses the larger solution. Between the solutions A_1 and A_2, $A_2 \geq A_1$, so we choose this solution to transpile the method `main(String[])`. The class `DifPoint` in the variable declaration has the version/type variable V_8/T_8 and the object instantiation `new DifPoint(0)` has the version/type variable V_9/V_9 The result of the transpilation is as follows.

```
public static void main(String[] args) {
  DifPoint_ver_1 p = new DifPoint_ver_1(0);
  int var2 = p.getA();
}
```

Joining everything, the transpilation of the compilation unit containing the class `DifMain` is shown by Listing 15.

```
package down;
import up.*;
public class DifMain_ver_2 {
  public boolean isZeroAngle(DifPoint_ver_1 p) {
    return p.getA() == 0f;
  }
  public boolean isZeroAngle(DifPoint_ver_2 p) {
    return p.getA() == 0f;
  }
  public static void main(String[] args) {
    DifPoint_ver_1 p = new DifPoint_ver_1(0);
    int var2 = p.getA();
  }
}
```

LISTING 15: Transpilation of `DifMain`

Chapter 5

Case Study

5.1 DocuSign Version Conflict

We base this case on a closed issue posted on the issue page of DocuSign[1]. A programmer was working on developing an application using Dropwizard, a Java framework. This framework uses many libraries, amongst them is Jackson[2], a Java library commonly used to handle JSON. At the time of the issue, the framework uses Jackson version 2.6.3. During development, the programmer attempted to add another dependency, the library DocuSign[3], a Java client library used to handle e-signature. Unfortunately, DocuSign was using an older version of Jackson, version 2.4.2. This caused a `java.lang.NoSuchMethodError`, when the framework Dropwizard attempted to call the method `JsonParser.isExpectedStartObjectToken()`, because this method was introduced from version 2.5. During runtime, the older version 2.4.2 of Jackson used by DocuSign appeared to have shadowed the later version used by Dropwizard.

The original issue was later solved by having DocuSign's developers update their Jackson's. Before that, DocuSign's dependents are bound to use the older Jackson' dependency before DocuSign was updated.

We simulate the same dependency relation and attempt to solve this issue without having to fiddle with DocuSign's dependency on Jackson. Figure 5.1 shows the simplified dependency relation between Jackson, DocuSign, and the application.

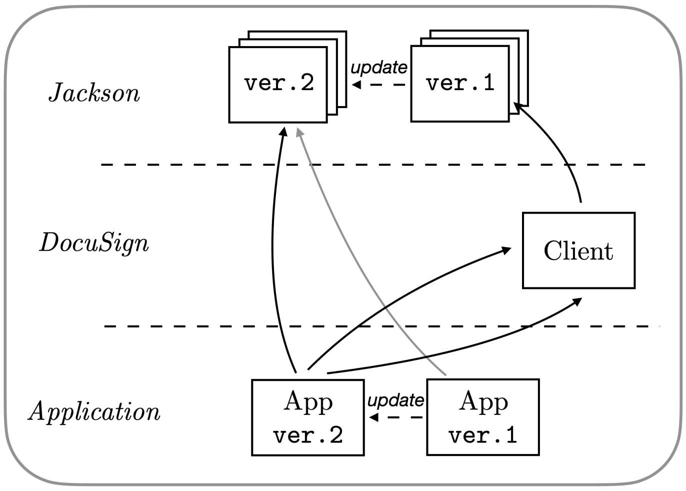


FIGURE 5.1: Version conflict involving DocuSign

[1] https://github.com/docusign/docusign-java-client/issues/22
[2] https://github.com/fasterxml
[3] https://www.docusign.com/

Here, we introduce two packages inside the library Jackson, `com.fasterxml.jackson.core` containing the class `JsonParser` and `com.fasterxml.jackson.databind` containing the class `ObjectMapper`. The different between before and after update in the class `JsonParser` lies in the new method `isExpectedStartObjectToken()` introduced in the new version. In the class `ObjectMapper`, the difference lies with the method `setFilters(Object)` removed in the new version. In Figure 5.1, the before and after update are labeled by version 1 and 2.

```
// BEFORE UPDATE
package com.fasterxml.jackson
    .core ver 1;
public class JsonParser {
  public JsonParser() {}
}
```

```
// AFTER UPDATE
package com.fasterxml.jackson
    .core ver 2;
public class JsonParser {
  public JsonParser() {}
  public boolean
    isExpectedStartObjectToken()
  { return true; }
}
```

LISTING 16: The class `JsonParser` before and after update

```
// BEFORE UPDATE
package com.fasterxml.jackson
    .databind ver 1;
public class ObjectMapper {
  public ObjectMapper() {}
  public void setFilters(Object obj) {}
}
```

```
// AFTER UPDATE
package com.fasterxml.jackson
    .databind ver 2;
public class ObjectMapper {
  public ObjectMapper() {}
}
```

LISTING 17: The class `ObjectMapper` before and after update

In Docusign, the class `ApiClient` uses `ObjectMapper` in its implementation, as shown in Listing 18. As previously shown in Figure 5.1, this class depends on the library Jackson before update.

```
package com.docusign.esign.client ver 1;
import com.fasterxml.jackson.databind.*;
public class ApiClient {
  private ObjectMapper mapper;
  public ApiClient() {
    this.mapper = new ObjectMapper();
    this.mapper.setFilters(new Object());
  }
}
```

LISTING 18: The class `ApiClient` in DocuSign

The class `App` in the application is shown by Listing 19. Before the update, the class only uses the class `JsonParser` from the library Jackson. Here, the object invokes the method `isExpectedStartObjectToken()` defined in Jackson after update. At this stage, in Java, `App` compiles and runs without issue. After the update, the class also uses the `ApiClient` that depends on the older version of Jackson. In Java, `App` after update can be compiled by including the new Jackson and DocuSign. However, it is impossible to have the compiled program running. If we include the new Jackson and DocuSign, the runtime will fail to find the method setFilters(Object) necessary in DocuSign. If we include the old Jackson and DocuSign, the runtime failure mentioned in the issue will occur, where the method isExpectedStartObjectToken() cannot be found.

```
// BEFORE UPDATE
package com.application ver 1;
import com.fasterxml.jackson.core.*;
public class App {
  public static void main(String[] args) {
    JsonParser parser = new JsonParser();
    boolean b =
      parser.isExpectedStartObjectToken();
  }
}
```

```
// AFTER UPDATE
package com.application ver 2;
import com.fasterxml.jackson.core.*;
import com.docusign.esign.client.*;
public class App {
  public static void main(String[] args) {
    JsonParser parser = new JsonParser();
    boolean b =
      parser.isExpectedStartObjectToken();

    ApiClient client = new ApiClient();
  }
}
```

LISTING 19: The class App before and after update

The listings we have shown so far are written in BatakJava. When written in BatakJava, the same issue does not occur. The compilation of ApiClient and version 2 of App are shown in Listing 20. As can be seen, the the version of ObjectMapper in ApiClient is fixed to version 1, while the class of JsonParser in App is fixed to version 2, allowing both methods to be used as long as both versions of Jackson are included during runtime.

```
package com.docusign.esign.client;
import com.fasterxml.jackson.databind.*;
public class ApiClient_ver_1 {
  private ObjectMapper_ver_1 mapper;
  public ApiClient_ver_1() {
    this.mapper = new ObjectMapper_ver_1();
    this.mapper.setFilters(new Object());
  }
}

package com.application;
import com.fasterxml.jackson.core.*;
import com.docusign.esign.client;
public class App_ver_2 {
  public static void main(String[] args) {
    JsonParser_ver_2 parser = new JsonParser_ver_2();
    boolean b = parser.isExpectedStartObjectToken();

    ApiClient_ver_1 client = new ApiClient_ver_1();
  }
}
```

LISTING 20: The compiled class ApiClient and App

Chapter 6

Related Work

6.1 Semantic Versioning

[23] proposed semantic versioning. The idea is to give meaning to the version number used to identify software. It imposes rules on how version numbers should be written. The triple version number major.minor.patch each represents a different kind of update. Major for backward incompatible, minor for backward compatible, and patch for fixes. This approach has started to be more widely adopted by developers, however, it only gives an outline for developers to describe their programs to downstream developers. However, even with that outline [25] found in their Maven repository study of libraries which follow semantic versioning guidelines that breaking changes are common, even in non-major release.

[15] argued that the semantic versioning rules which are collapsed into three integers rely heavily on the subjectivity of the developers. A more pragmatic manifesto and developer tools are important for developers. The concept of semantic versioning itself may be found to be broken as kinds of upgrades are inexpressible with the triple of integer.

6.2 Package Management Tools

Programmers generally rely on package management tools to help manage their various dependencies. These tools usually interact and replace files from outside of the language semantics. To some extent. some programming languages' package managers are powerful enough to help programmers circumvent some of the existing dependency's issue. But, as they only interact from the outside, there are limits to what a package manager can do.

Package managers attempt to alleviate some of the issues involved with complicated dependencies. For example, JavaScript's *npm* helps its users to resolve overlapping direct and transitive dependencies by nesting transitive dependencies separately[1]. Rust's *cargo* and npm also provide a renaming feature that allows programmers to use multiple versions of the same dependency side-by-side in the same project[2,3]. However, the renaming only works on the local level and does not apply to transitive dependencies. The above two package managers also have come to support semantic versioning.

[1] `http://npm.github.io/npm-like-im-5/npm3/dependency-resolution.html`
[2] `https://doc.rust-lang.org/cargo/`
[3] `https://docs.npmjs.com/cli/`

Another tool is provided by Maven's shade plugin for Java. This plugin allows users to rename dependencies and include them in the resulting jar, usually called an uber-jar[4].

6.3 Programming Paradigm

From the perspective of programming language design, several ideas have been proposed to cope with software evolution and variations.

Context-oriented Programming

Context-oriented programming [10] focuses on modularizing behavioral variations of an object. The behavior of an object, such as dispatched methods, is adapted dynamically following the runtime context. This paradigm treats context explicitly. The concept of context covers wide-ranging attributes that may be spatial or temporal or even based on hardware or software. The concept has been implemented as extensions to Java, Squeak/Smalltalk, and Common Lisp. LambdaVL can be considered to follow the same paradigm, where versions are treated as contexts.

Variational Programming

Variational programming [4] allows variations to be explicitly supported in programs. Examples of variations include code variation, such as in software product lines, and alternative privacy policies. Values in variational programming include choices, made up of left and right alternative. The paradigm also supports pattern matching on choices.

Family Polymorphism

If we consider classes of different versions, we may interpret them as a family of classes as introduced by family polymorphism [7, 11]. This can be done by encoding program dependencies (i.e. through `import`) and class updates with inheritance. However, the approach is restricted to the fact that updates cannot remove fields, methods, or classes.

6.4 Software Product Line

A *software product line* [21] consists of a set of similar software products that rely on a common code base. Ensuring all product variants satisfy a given property efficiently requires analyzing the variations, instead of every single product. Several implementation techniques have been introduced over the years, such as feature-oriented and delta-oriented programming [28]. Software product lines techniques do not support reconfiguring an object behavior during runtime.

Feature-oriented programming [22] introduce *features*. Features can be seen as abstract subclasses or mixins. Using features, objects can be created by selecting the desired features, hence allowing more flexibility of object creations and promote code reuse. [2] introduced the Feature Featherweight Java, a calculus for feature-oriented programming and stepwise refinement, where each feature can introduce a new class or refine already defined classes with different behaviors.

[4] https://maven.apache.org/plugins/maven-shade-plugin/

Similar to feature-oriented programming, delta-oriented programming [26] allows the creation of programs through the composition of a core and a set of delta modules. The core module is composed of an implementation of the common parts, while the delta modules provide changes to the core module. The main difference with feature-oriented programming is that delta-oriented programming does not only allow refinement of class' behaviors but also allows removing code.

In contrast to software product lines, dynamic software product lines [1] integrates the concept of software product lines and adaptive systems to products to be reconfigured during runtime.

The concept of software product lines is also applicable to producing domain-specific languages (DSLs), also called the language product lines. Several frameworks that implement this idea include MontiCore [12] and FeatureIDE [29].

Chapter 7

Conclusion

Software goes through continual changes that most often than not, contain breaking changes. In contemporary software development that is highly reliant on third-party software, changes in the upstream providers often cause inadvertent problems for downstream users. Downstream users suffering from dependency hell due to these changes then are forced spend much effort to handle migrations. Unfortunately, programming languages do not provide a mechanism that may help alleviate these issues.

We propose BatakJava, a subset of Java that considers versions as an attribute of a class. With this capability, a program can simultaneously use multiple versions of the same class, allowing us to avoid conflicting dependencies and problems resulting from structural or behavioral breaking changes. The examples and case study in this thesis showed the application of BatakJava in simplified real-world programs. The capability offered by BatakJava allows programmers to write programs beyond the bounds of conventional programming languages. In our approach, the versions of a program are variations that can be used according to need.

More research is needed to improve this concept. From the viewpoint of design, there are still many points to explore. The first important issue is that the current compilation scheme creates many code clones. In the current design, if we create a new version of a class even with no change from the previous version, the whole class will still be generated with a different version number, creating two classes with identical implementation but different naming. Code clones may also occur within method and field level. Ideally, code reuse should be encouraged in which one copy of the implementation is represented by different namings. An idea to deal with this would be to extract common parts from different versions of the class and use inheritance to reduce code clones.

The idea to reduce code clone through inheritance would require assigning meaning to the each version number. This relates to the versioning system, another aspect of the design that can be further improved. Currently, a version is an integer number that does not hold meaning on its own. We believe that incorporating the idea of semantic versioning into our concept of programming with versions may be beneficial. In this manner, as an example, it may be possible to express a class of a certain range of version using one common implementation.

Additionally, the current concept where the version is explicitly assigned by the programmer can also be replaced by automatic versioning by the compiler. This can be done by comparing a program with already compiled versions of the program, from where the difference can be statically analyzed and proper semantic versioning can be assigned.

To further prove the feasibility of this concept, it should be applied to a wider variety of cases with different sizes and kinds of breaking changes.

Lastly, since BatakJava is a prototype language that attempts to work on Java, a well-established high-level language, there are many features of Java left to be integrated into BatakJava.

Bibliography

[1] Stephan Adelsberger, Stefan Sobernig, and Gustaf Neumann. “Towards assessing the complexity of object migration in dynamic, feature-oriented software product lines”. In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. 2014, pp. 1–8.

[2] Sven Apel, Christian Kästner, and Christian Lengauer. “Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement”. In: *Proceedings of the 7th international conference on Generative programming and component engineering*. 2008, pp. 101–112.

[3] Aloïs Brunel et al. “A core quantitative coeffect calculus”. In: *European Symposium on Programming Languages and Systems*. Springer. 2014, pp. 351–370.

[4] Sheng Chen, Martin Erwig, and Eric Walkingshaw. “A calculus for variational programming”. In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

[5] Bradley E. Cossette and Robert J. Walker. “Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE ’12. New York, NY, USA: ACM, 2012, 55:1–55:11. ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393661. URL: http://doi.acm.org/10.1145/2393596.2393661 (visited on 03/19/2018).

[6] Jim des Rivieres. *Evolving Java-based APIs*. 2007. URL: https://wiki.eclipse.org/Evolving_Java-based_APIs (visited on 01/21/2021).

[7] Erik Ernst. “Family polymorphism”. In: *European Conference on Object-Oriented Programming*. Springer. 2001, pp. 303–326.

[8] James Gosling et al. *The Java language specification*. Addison-Wesley Professional, 2000.

[9] Lars Heinemann et al. “On the extent and nature of software reuse in open source java projects”. In: *International Conference on Software Reuse*. Springer. 2011, pp. 207–222.

[10] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. “Context-oriented programming”. In: *Journal of Object Technology* 7.3 (2008), pp. 125–151. ISSN: 16601769. DOI: 10.5381/jot.2008.7.3.a4.

[11] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. “Lightweight family polymorphism”. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2005, pp. 161–177.

[12] Holger Krahn, Bernhard Rumpe, and Steven Völkel. “MontiCore: a framework for compositional development of domain specific languages”. In: *International journal on software tools for technology transfer* 12.5 (2010), pp. 353–372.

[13] Charles W Krueger. “Software reuse”. In: *ACM Computing Surveys (CSUR)* 24.2 (1992), pp. 131–183.

[14] Raula Gaikovina Kula et al. "Do developers update their library dependencies?" en. In: *Empir Software Eng* 23.1 (Feb. 2018), pp. 384–417. ISSN: 1573-7616. DOI: 10.1007/s10664-017-9521-5. URL: https://doi.org/10.1007/s10664-017-9521-5 (visited on 06/19/2019).

[15] Patrick Lam, Jens Dietrich, and David J. Pearce. "Putting the Semantics into Semantic Versioning". In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. New York, NY, USA: Association for Computing Machinery, 2020, 157–179. ISBN: 9781450381789. URL: https://doi.org/10.1145/3426428.3426922.

[16] Manny M Lehman. "Laws of software evolution revisited". In: *European Workshop on Software Process Technology*. Springer. 1996, pp. 108–124.

[17] Parastoo Mohagheghi and Reidar Conradi. "Quality, productivity and economic benefits of software reuse: a review of industrial studies". In: *Empirical Software Engineering* 12.5 (2007), pp. 471–516.

[18] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. "Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. Santa Barbara, CA, USA: Association for Computing Machinery, 2017, 215–225. ISBN: 9781450350761. DOI: 10.1145/3092703.3092721. URL: https://doi.org/10.1145/3092703.3092721.

[19] Jesper Öqvist. "ExtendJ: extensible Java compiler". In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. 2018, pp. 234–235.

[20] Jens Palsberg and Michael I Schwartzbach. "Object-oriented type inference". In: *ACM SIGPLAN Notices* 26.11 (1991), pp. 146–161.

[21] David Lorge Parnas. "On the design and development of program families". In: *IEEE Transactions on software engineering* 1 (1976), pp. 1–9.

[22] Christian Prehofer. "Feature-oriented programming: A fresh look at objects". In: *European Conference on Object-Oriented Programming*. Springer. 1997, pp. 419–443.

[23] Tom Preston-Werner. *Semantic Versioning*. 2013. URL: https://semver.org/. (accessed: 28.01.2021).

[24] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. "Choco Solver". In: *Website, March* (2019).

[25] Steven Raemaekers, Arie van Deursen, and Joost Visser. "Semantic versioning and impact of breaking changes in the Maven repository". In: *Journal of Systems and Software* 129 (2017), pp. 140–158.

[26] Ina Schaefer et al. "Delta-oriented programming of software product lines". In: *International Conference on Software Product Lines*. Springer. 2010, pp. 77–91.

[27] Yudai Tanabe, Tomoyuki Aotani, and Hidehiko Masuhara. "A Context-Oriented Programming Approach to Dependency Hell". In: *Proceedings of the 10th International Workshop on Context-oriented Programming: Advanced modularity for run-time composition*. 2018, pp. 8–14.

[28] Thomas Thüm et al. "A classification and survey of analysis strategies for software product lines". In: *ACM Computing Surveys (CSUR)* 47.1 (2014), pp. 1–45.

[29] Thomas Thüm et al. "FeatureIDE: An extensible framework for feature-oriented software development". In: *Science of Computer Programming* 79 (2014), pp. 70–85.