BatakJava: An Object-Oriented Programming Language with Versions

Luthfan Anshar Lubis
luthfanlubis@prg.is.titech.ac.jp
Tokyo Institute of Technology
Meguro, Tokyo, Japan

Yudai Tanabe
yudaitnb@prg.is.titech.ac.jp
Tokyo Institute of Technology
Meguro, Tokyo, Japan

Tomoyuki Aotani
tomoyuki-aotani@mamezou.com
Mamezou Co., Ltd.
Tokyo, Japan

Hidehiko Masuhara
masuhara@acm.org
Tokyo Institute of Technology
Meguro, Tokyo, Japan

Abstract

Programming with versions is a recent proposal that supports multiple versions of software components in a program. Though it would provide greater freedom for the programmer, the concept is only realized as a simple core calculus, called λVL, where a value consists of λ-terms with multiple versions. We explore a design space of programming with versions in the presence of data structures and module systems, and propose BatakJava, an object-oriented programming language in which multiple versions of a class can be used in a program. This paper presents BatakJava's language design, its core semantics with subject reduction, an implementation as a source-to-Java translator, and a case study to understand how we can exploit multiple versions in BatakJava for developing an application program with an evolving library.

CCS Concepts: • **Software and its engineering** → **Object oriented languages**; **Software evolution**; • **Theory of computation** → *Type theory*.

Keywords: backward compatibility, dependency problem, Java

ACM Reference Format:
Luthfan Anshar Lubis, Yudai Tanabe, Tomoyuki Aotani, and Hidehiko Masuhara. 2022. BatakJava: An Object-Oriented Programming Language with Versions. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3567512.3567531

SLE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9919-7/22/12…$15.00
https://doi.org/10.1145/3567512.3567531

1 Introduction

Software is developed by composing many building blocks, which are continuously updated [18]. Each release of the software is distinguished from one another using versions. Through updates, developers improve performance, fix bugs, and provide new features [3, 5].

Some updates of components are *incompatible*, which are not welcomed but necessary. An incompatible update occurs when a software using an older version of its components no longer work correctly when the components are replaced with new version. Incompatible updates of the library programs (we call them *upstream* programs) are not always welcomed by the developers of the library users (we call them *downstream* developers) because they require the downstream developers to change their programs; the cost for update can discourage the developers to use newer versions [17, 21]. Incompatible updates are necessary for many reasons such as performance, simplicity, and modularity [4, 5], and in fact common [16, 22].

Existing techniques regarding incompatible updates mostly try to identify or evade incompatible changes. Semantic versioning [27] dictates conventions into versioning to help indicate which update introduces incompatible changes. Foo et al. introduced efficient static analysis to find API incompatibilities in library upgrades [11]. Other tools automatically repair dependency errors in programs and project builds [19, 23].

In contrast, *programming with versions* in λVL [30] proposed a novel approach to *embrace* incompatible changes by allowing a program to use both older and newer versions at the same time. λVL, a core calculus based on the concept of programming with versions, has *versioned values*, record-like entities that denote multiple possibilities of a value computed in different versions. The main focus of the calculus is a well-defined semantics and a static type system that chooses "version compatible" computations from different versions.

However, λVL has not explored a design space with data structures and module systems. Extending λVL to support

```
class Rectangle {
  int x; int y; float h; float w;
  Rectangle(int x, int y, float h, float w) {
    this.x = x; this.y = y; this.h = h; this.w = w; }
  int x() { return x; }
  int y() { return y; }
  float h() { return h; }
  float w() { return w; }
  float perimeter() { return 2*h+2*w; }
  boolean equivalent(Rectangle r) {
    return x == r.x() && y == r.y()
      && h == r.h() && w == r.w();
}}
```

```
class Square extends Rectangle {
  Square(int x, int y, float h) { super(x, y, h, h);
}}
```

```
class Example {
  void main() {
    Rectangle rec = new Rectangle(1,1,2,3);
    Square sq = new Square(1,1,2);
    float perSq = sq.perimeter();
    boolean eq = rec.equivalent(sq);
}}
```

Listing 1. Initial definition of the graphics library and downstream program

data structures and modules would require significant development in its formalizations. λVL manages versions as types' resources through the coeffect system. This system restricts variables from having different types in different versions.

Hence, rather than extending λVL, we propose a class-based object-oriented programming BatakJava based on the concept of programming with versions. This paper presents the language design along with the core calculus FBJ and its implementation as a source-to-Java translator. In BatakJava, multiple implementations of the same class are annotated with different versions to allow them to coexist in a program. An instance can represent multiple versions of a class without requiring users to specify the version for each instance.

BatakJava addresses λVL's limitation in updating declarations. Different versions of class declarations in BatakJava can freely modify the previously existing structure of a class, from its field, constructor, and method declarations. BatakJava also supports class inheritance and allows version updates to alter superclass. To allow these modifications, BatakJava employs constraint-based typing to check version availability along with the compatibility of definitions and expressions in the program.

```
class Rectangle {
  Point p; float h; float w;
  Rectangle(int x, int y, float h, float w) {
    this.p = new Point(x,y);
    this.h = h; this.w = w; }
  Rectangle(Point p, float h, float w) {
    this.p = p; this.h = h; this.w = w; }
  int x() { return p.x(); }
  int y() { return p.y(); }
  float h() { return h; }
  float w() { return w; }
  float area() { return 2*h+2*w; }
  boolean equal(Rectangle r) {
    return x() == r.x() && y() == r.y()
      && h == r.h() && w == r.w();
}}
```

```
class Point {
  int x; int y;
  Point(int x; int y) { this.x = x; this.y = y; }
  int x() { return x; }
  int y() { return y; }
}
```

```
class Example {
  void main() {
    Rectangle rec = new Rectangle(1,1,2,3);
    float area = rec.area();
    Square sq = new Square(1,1,2);
    float perSq = sq.perimeter();
    boolean eq = rec.equal(sq);
}}
```

Listing 2. Updated definition of the graphics library and downstream program

Through programming with versions, BatakJava aims to allow users to *safely* program in a multi-version environment. With a multi-version environment, users can avoid dependency hell and reduce migration cost through gradual version updates.

BatakJava focuses on the fundamental mechanism to detect and resolve version incompatibilities within the scope of API changes. To handle semantic changes, BatakJava provides low-level version selections on expression level and import declarations to choose the desired semantic.

The prototype implementation of BatakJava is used to demonstrate our proposal's usability with real-world programs. Programs in BatakJava are compiled into Java.

In summary, our contributions are as follows:

– The design of BatakJava, a Java-like object-oriented programming language based on the concept of programming with versions (Section 2)

– FBJ, a formal semantics of the core part of BatakJava, its type safety statement and proof (Section 3)
– An implementation of the prototype language BatakJava (Section 4)
– A case study that applies BatakJava to a real-world program containing dependency problems (Section 5)

2 Programming with Versions

We demonstrate the features of BatakJava through an example program using an evolving graphics library. We first show an implementation in pure Java so as to highlight the problems with versions. Then, we show an implementation in BatakJava where the version problems are resolved. Note that the example demonstrates the version problems on both the library and the application (library user) sides.

2.1 Motivating Example in Java

The example scenario has two steps: defining the upstream (library) and downstream (application) components, and changing the upstream.

Initial Definition. The upstream graphics library consists of class `Rectangle` and `Square` that inherits `Rectangle` (Listing 1).

The class `Rectangle` has fields `x`, `y`, `h`, and `w` representing the x-y coordinates of its top left corner, height, and width. A constructor initializes all four fields. There are getter methods for all fields. The method `perimeter` calculates the perimeter of a `Rectangle` instance. The method `equivalent` checks the equivalence between the instance with another `Rectangle` instance. `Square` inherits the structure from `Rectangle`, with a slightly modified constructor.

The `main` method in the class `Example` in Listing 1 uses both `Rectangle` and `Square`. It creates a `Rectangle` and a `Square` instances, and then invokes methods on them.

Updated Definition. The class `Rectangle` in Listing 2 is the updated version containing several incompatible[1] changes to the original implementation.

We changed the structure of the class by replacing the x-y coordinates with a `Point`, changing `Rectangle`'s getter methods, and added a new constructor that accepts a `Point`. The method `perimeter` is removed, `area` is added, and `equivalent` is renamed into `equal`. The class `Example` (Listing 2) remains the same except for a new invocation of the method `area` on `rec`; `equivalent` is replaced by `equal`.

This program does not compile in Java because `Example` requires the new version of `Rectangle` for `area` and the older version for `perimeter` (through `Square`) at the same time. Note that manually renaming with versions (name mangling) cannot resolve the problem. For example, we could rename the initial `Rectangle` as `Rectangle_old`, let `Square` inherit from `Rectangle_old`, and keep the updated

[1]By incompatibility, we refer to binary incompatibility in Java [12]

```
class Rectangle!1 {
  Rectangle!1(int x, int y, float h, float w) {...}
  ... }
```

```
class Square!1 {
  Square!1(int x, int y, float h) {...}
  ... }
```

```
class Example!1 {
  void main() {
    Rectangle rec  = new Rectangle(1,1,2,3);
    Square sq = new Square(1,1,2);
    float perSq = sq.perimeter();
    boolean eq = rec.equivalent(sq);
}}
```

Listing 3. Initial upstream and downstream definitions renamed with versions

`Rectangle` with the same name. However, it would fail to type check `rec.equal(s)` as the method requires the new version of `Rectangle` but receives a subtype of `Rectangle_old`.

2.2 Motivating Example in a Language with Versions

We rewrite the example in the syntax of BatakJava. We show the usage of versions and their effect from the two perspectives, namely the upstream graphics library and the downstream user, then demonstrate how multi-version programming solves the problem presented in the updated `Example`.

```
class Rectangle!2 {
  Rectangle!2(int x, int y, float h, float w) {...}
  Rectangle!2(Point p, float h, float w) {...}
  ... }
```

```
class Point!1 {
  Point!1(int x, int y) {...}
  ... }
```

```
1  class Example!2 {
2    void main() {
3      Rectangle rec = new Rectangle(1,1,2,3);
4      float area = rec.area();
5      Square sq = new Square(1,1,2);
6      float perSq = sq.perimeter();
7      boolean eq = rec.equal(sq);
8  }}
```

Listing 4. Updated upstream and downstream definitions renamed with versions

Upstream. We annotate the graphics library definition with versions. Versions are integer numbers treated as elements of types. A class and its instances are identified by their class names and versions. Syntactically, the definitions only change in class names and their constructor declarations. The graphics library in Listing 1 and 2 are rewritten into Listing 3 and 4.

The suffix `!1` attached to `Rectangle` denotes that the class is the *version 1* definition of the class. We annotate the initial definition of `Rectangle`, `Square`, and `Point` as version 1 and the updated `Rectangle` as version 2. Constructor declarations are also renamed to match their class names. The different versions annotations on both `Rectangle!1` and `Rectangle!2` distinguish them and allow a program to use both simultaneously.

Other definitions requiring class annotations, such as formal parameters, field, and method declarations, remain unchanged. In the implemented language, users are not expected to annotate each class annotation with their desired versions, as to avoid human errors in choosing versions. Determining the compatible version for each class annotation is left to the typing system. To handle semantic changes, manual version selections are provided for expressions and import declarations.

BatakJava tackles the limitations of λVL which does not allow modifications of data structure across versions. BatakJava does not impose any restriction on how a class can evolve in a different version, allowing all changes made in the class `Rectangle!2`. Newer versions can modify all aspects of a class definition, including changes in fields, methods, constructors, and superclass.

Another key aspect of BatakJava is the flexible usage of multiple versions in the program. Not only that a downstream program can access multiple versions of an upstream library, but also a newer version's definition can access an older version's definition and vice versa. For example, an instance of class `Rectangle!1` can receive constructor and method arguments of class `Rectangle!2`. This is allowed as long as the arguments are compatible with the constructor and method's body.

This idea of mixing versions also exhibits a form of adhoc version polymorphism in which a single method definition can represent multiple different signatures. This can be observed in the method `equal` in Listing 2 that takes a `Rectangle` instance as its argument `r`. The argument `r` is required to invoke the getter methods `x`, `y`, `h`, and `w`. All of these methods are available in both `Rectangle` version 1 and 2, hence the method `equal` in `Rectangle!2` is able to accept both versions of `Rectangle` as its argument.

Downstream. The updated header of `Example` that uses both `Rectangle!1` and `Rectangle!2` is shown in Listing 4. We suffix the class name with `!2` since it is the version 2 definition of the class.

The body of the `main` method in `Example!2` in Listing 4 does not require any version annotation, but is assumed to have access to both versions of `Rectangle`. The inclusion of two versions of `Rectangle` here changes the interpretation of each statement in the method.

The candidate versions for an instance are determined by the constructor used to create the instance and the consequent computations executed on that instance. Eventually, after all the related computations are considered, one specific version that is compatible with all the computations is chosen for the instance.

The version of an instance has to be consistent and an instance's version cannot switch from one to another version during runtime. BatakJava requires this because an instance may have different representations in different versions. Switching version during runtime without a well-defined adapter means an instance can change into a different instance completely. From a user standpoint, this will make it difficult for users to keep track of the instances existing in the program. From a design standpoint, it is also difficult to manage objects whose structures can change in the middle of an operation.

`new Rectangle(1,1,2,3)` in line 4 is instantiated using the constructor with signature `(int,int, int,int)` that is defined in both `Rectangle!1` and `Rectangle!2`. The instance is treated as an instance of `Rectangle!1` or `Rectangle!2` and the functionalities found in both versions can be used by this instance. In contrast, `new Square(1,1,2)` in line 6 which has only `Square!1` to instantiate, does not have any other variation.

The variable assignment on `rec` in line 3 declares a variable of class `Rectangle` initialized with `new Rectangle (1,1,2,3)`. This assignment binds the class and version of the instance with the class and version of variable `rec`. This implies that if `rec` uses computations specific to `Rectangle!2`, the instantiation `new Rectangle(1,1,2,3)` in line 4 will also be restricted to `Rectangle!2`.

Such a restriction is applied by the method invocation `rec.area()` in line 5. Since the method `area` is only available in `Rectangle!2`, this invocation constrains its receiver `rec` and the instantiation to `Rectangle!2`.

Another important element that can vary due to the inclusion of versions is the superclass. As class definitions do not specify which version of the superclass it inherits, the version has to be determined in other ways. Similar to the instance's version, the superclass of an instance is determined by the constructor used during instantiation and subsequent computations such as method invocations.

We can observe a superclass restriction for the variable `sq` by the method invocation `sq.perimeter()` in line 7. The method `perimeter` is a method that `Square!1` can possibly inherit from `Rectangle!1`. However, the class definition of `Square!1` does not specify which version of `Rectangle` it inherits. This method invocation then imposes constraint

indirectly on the superclass `Rectangle` that `Square` inherits from `Rectangle!1`.

In line 8, the variable `rec` invokes the method `equal` which is also specific to `Rectangle!2`. This does not contradict with the previous method invocation that requires `rec` to be restricted to `Rectangle!2`. `equal` requires an argument of class `Rectangle`, which as we have observed before, is compatible with both `Rectangle!1` and `Rectangle!2`. The variable `sq` has `Rectangle!1` as its possible superclass, hence the method invocation is allowed.

For the typing to work in the program, the version for each object has to be consistent. For example, invoking `area` by `rec` constrains the variable to `Rectangle!2`. If then `perimeter`, a method that is only available in `Rectangle!1`, is invoked by `rec`, the program will result in type error because the variable `rec` cannot be both `Rectangle!1` and `Rectangle!2` during runtime. `sq` also cannot invoke both methods because during runtime it cannot have both `Rectangle!1` and `Rectangle!2` as its superclass.

3 Core Calculus

We introduce the core calculus Featherweight BatakJava (FBJ) based on Featherweight Java (FJ) [14] to capture the essence of the object-oriented programming paradigm with class. FBJ extends FJ through the addition of version numbers to class declarations.

To handle variations that appear from the introduction of version numbers, FBJ is equipped with a constraint-based typing system. Solving the constraint generated during typing will provide version assignments for expressions and declarations inside the program. The typing system generates constraints on the variables in the program, to then be inferred by a separate solver. The dynamic semantics of the calculus is defined for the version-assigned program.

3.1 Paths

Paths:

$$paths(\texttt{Object}) = []$$

$$\frac{\texttt{class C!n extends D\#N \{...\}} \quad \texttt{p} \in paths(\texttt{D})}{\texttt{n :: p} \in paths(\texttt{C})}$$

Figure 1. Paths definition

A *path* `p` is a sequence of versions where the elements trace the version of the class and its ancestors. *paths* is a collection of path whose definition is shown in Figure 1. In the class definition `L`, the superclass of `C!n` is not fixed, denoting that an instance of such class can inherit one of the possibly many versions of `D`. This implies that there are multiple paths that the class `C!n` can inherit from. In other words, the type of an object does not depend only on its class version, but also on the version of its ancestors.

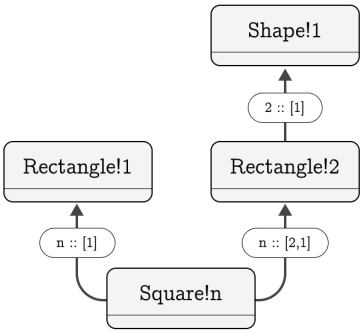

Figure 2. Paths example

```
P  ::=  L̄ e;
L  ::=  class C!n extends D#N {C#N f̄; K M̄}
K  ::=  C!n(C#N f̄){super(f̄); this.f̄ = f̄;}
M  ::=  C#N m(C#N x̄){ return e; }
e  ::=  x | e.f | e.m(ē) | new C#N(ē)
        | new C#p(ē) | (C#p) e
T  ::=  X#N | C#p
v  ::=  new C#p(ē)
p  ::=  n::p | n
```

Figure 3. Syntax of FBJ

3.2 Syntax

Figure 3 shows the syntax of FBJ. In most parts FBJ syntax is identical with FJ; it differs only in the requirement for version-related annotations as highlighted in the figure. The metavariables $\overline{\texttt{C}}$ and $\overline{\texttt{D}}$ range over class names; `f` and `g` range over field names; `m` ranges over methods names; `n`, `o` range over version numbers; `p`, `q` range over paths. `X` represents class variables, while `N` and `Q` are path variables. `this` is a reserved variable that is implicitly bound in every method declaration, while `super` calls the method or constructor of the superclass. Overlines are used to denote sequences separated by commas or semicolons. For example, $\overline{\texttt{e}}$ is shorthand for a possibly empty sequence $\texttt{e}_1, \cdots, \texttt{e}_n$, similarly also with $\overline{\texttt{C\#N}}$, $\overline{\texttt{M}}$, $\overline{\texttt{n}}$, etc. Sequences of field declarations, formal parameters, and method declarations are assumed to contain no duplicate names.

FBJ extends FJ through the addition of version number annotation `n` in the class declaration `L`. This denotes declaring a class `C` with version `n`. Accordingly, the constructor declaration `K` also has the version number attached.

A type can be a type variable `X#N`, where `X` is a class variable and `N` is a path variable. `C#p` is a type of concrete class `C` with a concrete path `p`. For convenience, we will call this a

Fields Lookup:

$$fields(\texttt{Object}, []) = \emptyset$$

$$\frac{\texttt{class C!n extends D\#N } \{\overline{\texttt{C\#N f}}\texttt{; K } \overline{\texttt{M}}\} \quad fields(\texttt{D}, \texttt{p}) = \overline{\texttt{D\#Q g}}}{fields(\texttt{C}, \texttt{n} :: \texttt{p}) = \overline{\texttt{D\#Q g}}, \overline{\texttt{C\#N f}}}$$

Method Type Lookup:

$$\frac{\texttt{class C!n extends D\#N } \{\overline{\texttt{C\#N f}}\texttt{; K } \overline{\texttt{M}}\} \quad \texttt{B\#Q m}(\overline{\texttt{B\#Q x}})\texttt{\{ return e; \}} \in \overline{\texttt{M}}}{mtype(\texttt{m}, \texttt{C}, \texttt{n} :: \texttt{p}) = \overline{\texttt{B\#Q}} \rightarrow \texttt{B\#Q}}$$

$$\frac{\texttt{class C!n extends D\#N } \{\overline{\texttt{C\#N f}}\texttt{; K } \overline{\texttt{M}}\} \quad \texttt{m} \notin \overline{\texttt{M}}}{mtype(\texttt{m}, \texttt{C}, \texttt{n} :: \texttt{p}) = mtype(\texttt{m}, \texttt{D}, \texttt{p})}$$

Method Body Lookup:

$$\frac{\texttt{class C!n extends D\#N } \{\overline{\texttt{C\#N f}}\texttt{; K } \overline{\texttt{M}}\} \quad \texttt{B\#Q m}(\overline{\texttt{B\#Q x}})\texttt{\{ return e; \}} \in \overline{\texttt{M}}}{mbody(\texttt{m}, \texttt{C}, \texttt{p}) = \overline{\texttt{x}}.\texttt{e}}$$

$$\frac{\texttt{class C!n extends D\#N } \{\overline{\texttt{C\#N f}}\texttt{; K } \overline{\texttt{M}}\} \quad \texttt{m} \notin \overline{\texttt{M}}}{mbody(\texttt{m}, \texttt{C}, \texttt{n} :: \texttt{p}) = mbody(\texttt{m}, \texttt{D}, \texttt{p})}$$

Figure 4. Auxillary definitions for FBJ

concrete type. We illustrate this with the example shown in Figure 2. An instance of class `Square!n`, depending on which version of `Rectangle` it inherits, will have different paths of different lengths and values. If it inherits `Rectangle!1`, then the path is `[n,1]`, otherwise it is `[n,2,1]`.

An FBJ program P consists of a sequence of classes $\overline{\texttt{L}}$ and an expression e. Class, constructor, and method declarations are similar to FJ except that the types of the fields, formal parameters, and method return types are affixed with path variables. Object instantiations and cast expressions have two expressions, one where the class is affixed with a path variable N and another one with a concrete path p. The expression with the concrete path is the expression we obtain after assigning the program with versions obtained from the inference.

Figure 4 shows the auxiliary definitions required for the typing and evaluation rules. The definition only differs from FJ with its inclusion of path.

3.3 Constraints

The subtyping and typing rules in FBJ generate constraints on the variables found in the program. A constraint R is in disjunctive normal form, where a literal is either a constraint on a class variable or a path variable.

An example of constraint on class variable is $\{\texttt{X} = \texttt{C}\}$, constraining the variable X to the class C. While an example of

Concrete Type Subtyping:

$$\texttt{C\#p} <: \texttt{C\#p}$$

$$\frac{\texttt{class C!n extends D\#N \{...\}} \quad \texttt{p} \in paths(\texttt{D})}{\texttt{C\#(n :: p)} <: \texttt{D\#p}}$$

$$\frac{\texttt{C\#p} <: \texttt{D\#p}' \quad \texttt{D\#p}' <: \texttt{E\#p}''}{\texttt{C\#p} <: \texttt{E\#p}''}$$

Type Variable Subtyping:

$$\frac{\texttt{C}_j \in dom(\texttt{CT}) \quad \texttt{p}_{jk} \in paths(\texttt{C}) \quad \texttt{q}_i \in paths(\texttt{D}) \quad \texttt{C}_j\texttt{\#p}_{jk} <: \texttt{D\#q}_i}{\texttt{X\#N} <: \texttt{D\#Q} \mid \bigoplus_{i,j,k}\{\texttt{X\#N} = \texttt{C}_j\texttt{\#p}_{jk} \wedge \texttt{Q} = \texttt{q}_i\}}$$

Figure 5. Subtyping in FBJ

constraint on path variable is $\{\texttt{N} = \texttt{p}\}$, constraining the variable N to the path p. Both constraints can also be expressed together as $\{\texttt{X\#N} = \texttt{C\#p}\}$.

Addition $\oplus$ and multiplication $\otimes$ are defined as follow.

$$\texttt{R} \oplus \texttt{P} \triangleq \texttt{R} \vee \texttt{P}$$
$$\texttt{R} \otimes \texttt{P} \triangleq \bigvee [\texttt{r} \wedge \texttt{p} \mid \texttt{r} \in \texttt{R}, \texttt{p} \in \texttt{P}]$$

The addition of two constraints is the disjunction of the two constraints. The multiplication of two constraints is the disjunction of pair-wise conjunction between the two constraints.

A solution σ is a mapping from class and path variables to class names and paths. $\texttt{R} \vDash \sigma$ is read as "solution σ does not contradict with constraint R". Given that the constraint is in disjunctive normal form, it has the following properties.

1. If $\texttt{R}_1 \vDash \sigma$ and $\texttt{R}_2 \vDash \sigma$, then $\texttt{R}_1 \otimes \texttt{R}_2 \vDash \sigma$.
2. If $\texttt{R}_1 \vDash \sigma$ then for any constraint $\texttt{R}_2$, $\texttt{R}_1 \oplus \texttt{R}_2 \vDash \sigma$.

3.4 Subtyping

Figure 5 shows the subtyping relation in FBJ. The first three relations are subtyping relations between concrete types. The first rule is reflection, where a class is the subtype of itself, only if the paths are equal. The second relation is the relation induced by the class definition. Given the path n :: p, the subtyping relation checks whether the rest of the path p is a valid path for the superclass D. The third relation is the transitive subtyping relation.

The final subtyping relation is between a type variable X#N and a type with concrete class and path variable D#Q. To check if the relation holds, all applicable types for X#N and path Q are searched through and those that apply are added to the constraint.

3.5 Typing

The typing rules for FBJ is shown in Figure 6. The typing judgement for expressions has the form $\Gamma \vdash \texttt{e} : \texttt{T} \mid \texttt{R}$ which

Typing Rules:

$$\frac{\Gamma(\mathtt{x}) = \mathtt{C\#N} \qquad \mathtt{p}_k \in \mathit{paths}(\mathtt{C})}{\Gamma \vdash \mathtt{x} : \mathtt{X\#N} \mid \bigoplus_k \{\mathtt{X} = \mathtt{C} \wedge \mathtt{N} = \mathtt{p}_k\}} \text{(T-Var)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{X}_0\mathtt{\#N}_0 \mid \mathtt{R}_0 \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{X\#N}} \mid \overline{\mathtt{R}} \qquad \mathtt{C}_j \in \mathit{dom}(\mathtt{CT}) \\ \mathtt{p}_{jk} \in \mathit{paths}(\mathtt{C}_j) \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_j, \mathtt{p}_{jk}) = \overline{\mathtt{D}_{jk}\mathtt{\#Q}_{jk}} \to \mathtt{D}_{jk}\mathtt{\#Q}_{jk} \\ \overline{\mathtt{X\#N}} <: \overline{\mathtt{D}_{jk}\mathtt{\#Q}_{jk}} \mid \overline{\mathtt{P}_{jk}} \qquad \mathtt{q}_l \in \mathit{paths}(\mathtt{D}_{jk})\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{X\#N} \mid \bigoplus_{j,k,l}(\{\mathtt{X\#N} = \mathtt{D}_{jk}\mathtt{\#Q}_{jk} \\ \wedge\ \mathtt{Q}_{jk} = \mathtt{q}_l \ \wedge\ \mathtt{X}_0\mathtt{\#N}_0 = \mathtt{C}_j\mathtt{\#p}_{jk}\} \otimes \mathtt{R}_0 \otimes \overline{\mathtt{R}} \otimes \overline{\mathtt{P}_{jk}})\end{array}} \text{(T-Invk)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{X}_0\mathtt{\#N}_0 \mid \mathtt{R}_0 \qquad \mathtt{C}_j \in \mathit{dom}(\mathtt{CT}) \qquad \mathtt{p}_{jk} \in \mathit{paths}(\mathtt{C}_{jk}) \\ \mathit{fields}(\mathtt{C}_j, \mathtt{p}_{jk}) = \overline{\mathtt{C}_{jk}\mathtt{\#N}_{jk}}\ \overline{\mathtt{f}_{jk}} \qquad \mathtt{p}_l \in \mathit{paths}(\mathtt{C}_{jki})\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{X\#N} \mid \bigoplus_{j,k,l}(\{\mathtt{X\#N} = \mathtt{C}_{jki}\mathtt{\#N}_{jki} \\ \wedge\ \mathtt{N}_{jki} = \mathtt{p}_l \ \wedge\ \mathtt{X}_0\mathtt{\#N}_0 = \mathtt{C}_j\mathtt{\#p}_{jk}\} \otimes \mathtt{R}_0)\end{array}} \text{(T-Field)}$$

$$\frac{\begin{array}{c}\mathtt{n}_j :: \mathtt{q}_j = \mathtt{p}_j \in \mathit{paths}(\mathtt{C}) \qquad \mathit{fields}(\mathtt{C}, \mathtt{p}_j) = \overline{\mathtt{C}_j\mathtt{\#N}_j}\ \overline{\mathtt{f}_j} \\ \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{X\#N}} \mid \overline{\mathtt{R}} \qquad \overline{\mathtt{X\#N}} <: \overline{\mathtt{C}_j\mathtt{\#N}_j} \mid \overline{\mathtt{R}_j} \\ \mathtt{class\ C!n}_j\ \mathtt{extends\ D\#Q\ \{}\overline{\mathtt{C\#Q}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}\ OK} \mid \mathtt{P}_j\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{new\ C\#N}(\overline{\mathtt{e}}) : \mathtt{X\#N} \mid \bigoplus_j(\{\mathtt{X} = \mathtt{C} \\ \wedge\ \mathtt{N} = \mathtt{p}_j \wedge \mathtt{Q} = \mathtt{q}_j\} \otimes \mathtt{P}_j \otimes \overline{\mathtt{R}} \otimes \overline{\mathtt{R}_j})\end{array}} \text{(T-New)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{X}_0\mathtt{\#N}_0 \mid \mathtt{R}_0}{\Gamma \vdash \mathtt{(C\#p)\ e}_0 : \mathtt{X\#N} \mid \bigoplus_k(\{\mathtt{X} = \mathtt{C} \wedge \mathtt{N} = \mathtt{p}\} \otimes \mathtt{R}_0)} \text{(T-Cast)}$$

Method Typing:

$$\frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{C\#Q}}, \mathtt{this} : \mathtt{C\#(n} :: \mathtt{p}_k) \vdash \mathtt{e}_0 : \mathtt{X}_0\mathtt{\#N}_0 \mid \mathtt{R}_0 \\ \mathtt{X}_0\mathtt{\#N}_0 <: \mathtt{C}_0\mathtt{\#Q}_0 \mid \mathtt{P} \\ \mathtt{class\ C!n\ extends\ D\#N\ \{...\}} \qquad \mathtt{p}_k \in \mathit{paths}(\mathtt{D}) \\ \mathit{mtype}(\mathtt{m}, \mathtt{D}, \mathtt{p}_k) = \overline{\mathtt{D}_k\mathtt{\#N}_k} \to \mathtt{D}_k\mathtt{\#N}_k\end{array}}{\begin{array}{c}\mathtt{C}_0\mathtt{\#Q}_0\ \ \mathtt{m(}\overline{\mathtt{C\#Q}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e}_0\mathtt{;\ \}\ OK\ in\ C!n} \\ \mid \bigoplus_k \big(\{\mathtt{N} = \mathtt{p}_k \wedge \overline{\mathtt{Q}} = \overline{\mathtt{N}_k} \wedge \mathtt{Q}_0 = \mathtt{N}_k \\ \wedge\ \overline{\mathtt{D}_k} = \overline{\mathtt{C}} \wedge \mathtt{D}_k = \mathtt{C}_0\} \otimes \mathtt{R}_0 \otimes \mathtt{P}\big)\end{array}} \text{(T-Method)}$$

Class Typing:

$$\frac{\begin{array}{c}\mathtt{K = C!n(}\overline{\mathtt{D'\#Q'}}\ \overline{\mathtt{g}}\mathtt{,}\ \overline{\mathtt{C\#N}}\ \overline{\mathtt{f}}\mathtt{)\{super(}\overline{\mathtt{g}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{ = }\overline{\mathtt{f}}\mathtt{;\}} \\ \mathtt{p}_k \in \mathit{paths}(\mathtt{D}) \qquad \mathit{fields}(\mathtt{D}, \mathtt{p}_k) = \overline{\mathtt{D\#Q}}\ \overline{\mathtt{g}} \\ \overline{\mathtt{M}}\ \mathtt{OK\ in\ C!n} \mid \overline{\mathtt{R}}\end{array}}{\begin{array}{c}\mathtt{class\ C!n\ extends\ D\#N\ \{}\overline{\mathtt{C\#N}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}\ OK} \\ \mid \bigoplus_k(\{\mathtt{N} = \mathtt{p}_k \wedge \overline{\mathtt{D'}} = \overline{\mathtt{D}} \wedge \overline{\mathtt{Q'}} = \overline{\mathtt{Q}}\} \otimes \overline{\mathtt{R}})\end{array}} \text{(T-Class)}$$

Figure 6. Typing rules for FBJ

reads as "in typing environment Γ, expression e has type T under constraint R."

All the typing rules in FBJ assign either a fresh type variable X#N. The rules are analogous to typing rules of FJ, except

Computation rules:

$$\frac{\mathit{fields}(\mathtt{C}, \mathtt{p}) = \overline{\mathtt{C\#N}}\ \overline{\mathtt{f}}}{\mathtt{(new\ C\#p(}\overline{\mathtt{e}}\mathtt{))}.\mathtt{f}_i \xrightarrow{\sigma} \mathtt{e}_i} \text{(R-Field)}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}, p) = \overline{\mathtt{x}}.\mathtt{e}_0}{\mathtt{(new\ C\#p(}\overline{\mathtt{e}}\mathtt{))}.\mathtt{m(}\overline{\mathtt{d}}\mathtt{)} \xrightarrow{\sigma} [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C\#p(}\overline{\mathtt{e}}\mathtt{)/this}]\ \sigma(\mathtt{e}_0)} \text{(R-Invk)}$$

$$\frac{\mathtt{C\#[}\overline{\mathtt{n}}\mathtt{]} <: \mathtt{D\#[}\overline{\mathtt{o}}\mathtt{]}}{\mathtt{(D\#[}\overline{\mathtt{o}}\mathtt{])(new\ C\#p(}\overline{\mathtt{e}}\mathtt{))} \xrightarrow{\sigma} \mathtt{new\ C\#p(}\overline{\mathtt{e}}\mathtt{)}} \text{(R-Cast)}$$

Congruence rules:

$$\frac{\mathtt{e}_0 \xrightarrow{\sigma} \mathtt{e}_0'}{\mathtt{e}_0.\mathtt{f} \xrightarrow{\sigma} \mathtt{e}_0'.\mathtt{f}} \text{(RC-Field)}$$

$$\frac{\mathtt{e}_0 \xrightarrow{\sigma} \mathtt{e}_0'}{\mathtt{e}_0.\mathtt{m(}\overline{\mathtt{e}}\mathtt{)} \xrightarrow{\sigma} \mathtt{e}_0'.\mathtt{m(}\overline{\mathtt{e}}\mathtt{)}} \text{(RC-Invk-Recv)}$$

$$\frac{\mathtt{e}_i \xrightarrow{\sigma} \mathtt{e}_i'}{\mathtt{v}_0.\mathtt{m(}\ldots,\mathtt{e}_i,\ldots\mathtt{)} \xrightarrow{\sigma} \mathtt{v}_0.\mathtt{m(}\ldots,\mathtt{e}_i',\ldots\mathtt{)}} \text{(RC-Invk-Arg)}$$

$$\frac{\mathtt{e}_i \xrightarrow{\sigma} \mathtt{e}_i'}{\mathtt{new\ C\#p(}\ldots,\mathtt{e}_i,\ldots\mathtt{)} \xrightarrow{\sigma} \mathtt{new\ C\#p(}\ldots,\mathtt{e}_i',\ldots\mathtt{)}} \text{(RC-New-Arg)}$$

$$\frac{\mathtt{e}_0 \xrightarrow{\sigma} \mathtt{e}_0'}{\mathtt{(C\#p)e}_0 \xrightarrow{\sigma} \mathtt{(C\#p)e}_0'} \text{(RC-Cast)}$$

Figure 7. Computation and congruence rules for FBJ

that the rules in FBJ handle all the possible concrete typing combinations for the expressions inside the constraint.

T-Var types a variable x which belongs to the typing environment Γ. A variable can be mapped to a type with a path variable, in which case this typing rule adds a constraint to the path variable by checking through available paths for class C.

T-Invk types a method invocation m with arguments $\overline{\mathtt{e}}$ on the receiver $\mathtt{e}_0$. The typing rule searches for the method m by going through the class table. For every method m found, the subtyping relation between the arguments and the method's parameters adds another set of constraints $\overline{\mathtt{P}_{jk}}$. X#N is then bound to the type $\mathtt{D}_{jk}\mathtt{\#Q}_{jk}$ found in the definition of method m. The typing joins all the constraints in the assumptions with the constraint on the fresh type variable X#N and the receiver's type $\mathtt{X}_0\mathtt{\#N}_0$.

Starting with a receiver $\mathtt{e}_0$ with the type variable $\mathtt{X}_0\mathtt{\#N}_0$, T-Field searches through the class table for concrete types that contain the field $\mathtt{f}_i$. If found, the possible concrete type for this field is added to the constraint along with the constraint on the receiver type.

T-New behaves similar to T-Invk, except it checks the arguments against *fields* instead of *mtype*'s parameters. An important part of T-New is having the class typing in the assumption. This is necessary because the superclass' path Q can change depending on what arguments are actually given during object instantiation.

T-Cast packs FJ's upcast, downcast, and stupid cast into one rule. The correctness of the cast is checked during runtime.

The typing judgement for method declarations has the form M OK in C!n. The first and second lines of the assumption add a subtyping constraint on the method body expression. The remaining assumptions are necessary for overriding. In FBJ, a class C!n has possibly multiple options of superclass to inherit. Each of the superclass options can possibly have different typing for a method m, therefore it is impossible to apply the same overriding rule that FJ has on FBJ. Instead, we add constraint in the resulting method typing that ensures if there are multiple superclasses that have method m defined, the overriding class C!n must have the same typing with at least one of them. A constraint is also then added to the superclass' path variable N.

The class typing does a similar check to the method typing, but on the fields' names and types. If it has the same fields with a particular superclass $\texttt{D\#P}_k$, then constraint is added on the superclass' path variable N along with the fields' types $\overline{\texttt{D'\#Q'}}$. The resulting constraint joins this constraint with the constraint from method typing.

3.6 Evaluation

The reduction rules are shown in Figure 7 given by the relation $\texttt{e} \xrightarrow{\sigma} \texttt{e'}$ which reads as "expression e reduces to expression e' under type substitution σ". The type substitution σ is obtained from solving the constraint generated during typing derivation. The rules are defined for expressions whose variables are replaced by the type substitution σ.

The three computation rules are defined for field access, method invocation, and casting. In R-Field, if $\texttt{f}_i$ is found among the fields of C#p, then the argument as the same position as $\texttt{f}_i$ is the result of the reduction. In R-Invk, once the method body is found, the variables in the method body's expression itself need to be replaced by σ. Then, this in the method body is replaced by the expression's receiver object, and the parameters $\overline{\texttt{x}}$ are substituted by the invocation's arguments $\overline{\texttt{d}}$. In R-Cast, if the subtyping relation holds, the reduction proceeds.

3.7 Subject Reduction

The subject reduction property states that given a valid substitution for an expression typing, then after reducing the expression, the substitution will satisfy the constraint of the new typing and also gives the same or narrower type as before the reduction. Subject reduction guarantees that an object after evaluation will maintain the same class and version number.

```
class Square!1 extends Rectangle₁ { ... }

class Rectangle!2 {
  boolean equal(Rectangle₂ r) { ... }
  ... }

class Example!2 {
  void main() {
    Rectangle₃ rec = new Rectangle(1,1,2,3);
    float area = rec.area();
    Square sq = new Square(1,1,2);
    float perSq = sq.perimeter();
    boolean eq = rec.equal(sq);
}}
```

Listing 5. Program with indexed Rectangle

Theorem (Subject Reduction). If $\Gamma \vdash \texttt{e} : \texttt{X\#N} \mid \texttt{R}, \texttt{R} \vDash \sigma$, $\sigma(\texttt{e}) \xrightarrow{\sigma} \texttt{e'}$, then for some $\texttt{X'\#N'}$, $\Gamma \vdash \texttt{e'} : \texttt{X'\#N'} \mid \texttt{R'}, \texttt{R'} \vDash \sigma$, $\sigma(\texttt{X'\#N'}) <: \sigma(\texttt{X\#N})$.

Proof. We prove this theorem by induction on the typing relation and analyzing the last rule used.

4 Implementation

We implemented a prototype language BatakJava, based on FBJ. The language is implemented using ExtendJ [24], an extensible Java compiler that supports bytecode compilation. BatakJava programs are compiled into Java.

The key point of BatakJava implementation is the version annotation on class definitions and the subsequent typing and Java code generation. BatakJava compiler includes a constraint-based version inference system and a Java code generator.

Version inference checks and finds compatible version assignments for each class annotation in a BatakJava program. This consists of constraint generation and solving. The failure to find a solution is equal to a typing error.

4.1 Version Inference

The first step of the version inference is constraint generation. BatakJava follows the constraint generation described by the typing rules of FBJ. A literal constraint in BatakJava is described as {X=C!n}, which constrains a *type* variable X to a specific class C!n. Type variables are assigned to each expression and class annotation.

The generated constraints are then passed to the constraint solver Chocosolver [28] from which the solution is used to guide the subsequent code generation phase. Other constraint solvers can also be used for this purpose.

Constraint Generation Example. We describe the constraint generated by some of the expressions in main method

of `Example!2` in Listing 4 and class annotations related to it. Since there are overlapping mentions of `Rectangle`, Listing 5 shows the definition from Listing 4 with indexed `Rectangle` to help distinguish their type variables. Figure 8 shows the type variables for the expressions and class annotations involved in the method. Each method invocation generates fresh type variables on the parameters, here X_2 refers to the original type variable associated with parameter `r` of `equal`, and X_2' is the fresh type variable for parameter `r` generated by the invocation `rec.equal(sq)`.

The object instantiation `new Rectangle(1,1,2,3)` in line 3 generates the following constraint.

$$\{X_4 = \texttt{Rectangle!1}\} \vee \{X_4 = \texttt{Rectangle!2}\}$$

This denotes that the object instantiation with variable X_4 is typed as `Rectangle!1` or `Rectangle!2`. Constraints are also generated in regards to the arguments and constructor's parameters, but since all the parameters are primitive types, we skip the details for simplicity.

The object instantiation `new Square(1,1,2)` in line 5 generates constraint on its own type variable and the type variable for its superclass $\texttt{Rectangle}_1$. It results in

$$\{X7 = \texttt{Square!1} \wedge X1 = \texttt{Rectangle!1}\} \vee \{X7 = \texttt{Square!1} \wedge X1 = \texttt{Rectangle!2}\}$$

This denotes that the superclass `Rectangle1` with variable `X1` can be typed either as `Rectangle!1` or `Rectangle!2`.

The variable assignment in line 3 of the method generates the following constraint.

$$\{X4 = \texttt{Rectangle!1} \wedge X3 = \texttt{Rectangle!1}\} \vee \{X4 = \texttt{Rectangle!2} \wedge X3 = \texttt{Rectangle!2}\}$$

The constraint generation checks in which types the subtyping relation for the assignment stands. Here, because the assigned variable `rec` and the object instantiation are both of class `Rectangle`, the constraint requires them to be of the same version for the subtyping relation to be valid.

Lastly we observe the constraint generation on `rec.equal(sq)`. The constraint involves the receiver object, method's argument, method's parameter, and the expression itself. It generates the following constraint.

$$\{X3 = \texttt{Rectangle!2} \wedge X6 = \texttt{Square!1} \wedge X1 = \texttt{Rectangle!1} \wedge X2' = \texttt{Rectangle!1} \wedge X9 = \texttt{bool}\} \vee \{X3 = \texttt{Rectangle!2} \wedge X6 = \texttt{Square!1} \wedge X1 = \texttt{Rectangle!2} \wedge X2' = \texttt{Rectangle!2} \wedge X9 = \texttt{bool}\}$$

This method is available only in `Rectangle!2`, hence the type variable of the receiver object is bound to that type. `sq` can only by typed as `Square!1`, so `X6` is bound to that type. The method invocation also generates a fresh variable `X'2` for the parameter `r` in `equal`'s parameters. The types bound to `X1` which represents `Square!1`'s superclass and `X'2` have to be the same. If the parameter is assigned as `Rectangle!1`, then the argument `sq` has to inherit from that type so that the subtyping relation between the argument and the parameter holds.

Expression/Class Access	Var
$\texttt{Rectangle}_1$	X_1
$\texttt{Rectangle}_2$	X_2, X_2'
$\texttt{Rectangle}_3$	X_3
`new Rectangle(1,1,2,3)`	X_4
`rec.area()`	X_5
`Square`	X_6
`new Square(1,1,2)`	X_7
`sq.perimeter()`	X_8
`rec.equal(sq)`	X_9

Figure 8. Type variables for the example

Var	Type Assignment
X_1	`Rectangle!1`
X_2	`Rectangle!1` and `Rectangle!2`
X_2'	`Rectangle!1`
X_3	`Rectangle!2`
X_4	`Rectangle!2`
X_5	`float`
X_6	`Square!1`
X_7	`Square!1`
X_8	`float`
X_9	`boolean`

Figure 9. Solution for the constraint generation

All the constraints, along with those not described here, are joined together as a conjunction. A possible solution for this constraint is shown by the type assignments in Figure 9.

Version Selection. The version of expressions in BatakJava can be ambiguous if there are no following computations that restrain their versions. For example, `new Rectangle(1,1,2,3)` can assume both version 1 and 2 if the method `area` is not invoked afterwards. This ambiguity contradicts the requirement for a specific version for evaluating the expression. To handle this ambiguity, BatakJava chooses the latest (largest) version as the default version for any object instantiation when there are multiple solutions provided.

BatakJava also provides version selection to allow manual control of versions by users. For example, version 1 is selected for the `Rectangle` instance by the following selection.

```
Rectangle rec = new Rectangle#1#(1,1,2,3);
```

This expression will generate the following constraint.

$$\{X_4 = \texttt{Rectangle!1}\}$$

4.2 Code Generation

The main gap between BatakJava and Java is BatakJava's policy of allowing classes with the same fully qualified names

but different versions to be defined in the program. The code generation handles this by renaming definitions and declarations using versions. However, the version assignments remove the variability of a BatakJava program. For example, the variability of the superclass `Rectangle` in `Square!1`.

Class names are rewritten during code generation. A class `C!n` is renamed into `C_ver_n`. The class headers from the ongoing example is rewritten into the following classes.

```
class Rectangle_ver_1 { ... }
class Rectangle_ver_2 { ... }
class Point_ver_1 { ... }
class Example_ver_2 { ... }
```

Rewriting the class `Square!1` requires the type assignment from the solution for the superclass access $\texttt{Rectangle}_1$. Based on the solution in Figure 9, the class header is renamed as follows.

```
class Square_ver_1 extends Rectangle_ver_1 { ... }
```

Renaming class access and expressions are guided by the solution obtained from the constraint solver. This conversion can be observed on each body declaration of a class. Using the solution in Figure 9, `main` method is translated as follows.

```
void main() {
  Rectangle_ver_2 rec = new Rectangle_ver_2(1,1,2,3);
  float area = rec.area();
  Square_ver_1 sq = new Square(1,1,2);
  float perSq = sq.perimeter();
  boolean eq = rec.equal(sq);
}
```

A special care is needed with the compilation of the method `equal` in regards to its compatibility with both versions of `Rectangle`. The compilation creates copies of the method corresponding to the compatible combination of parameters and return type.

```
boolean equal(Rectangle_ver_1 r) { ... }
boolean equal(Rectangle_ver_2 r) { ... }
```

The same is also applicable with the method `equivalent` in `Rectangle!1`.

The other body declarations, such as constructor and field declarations, are not compiled polymorphically.

5 Case Study

We implemented a simple text file processing program using the line reading and text splitting functionality from the Guava library[2]. Classes necessary for the program are rewritten in BatakJava. We simulated backward incompatible changes in the rewritten Guava library and used BatakJava

```
// Before update
class Splitter!1 {
  public static Splitter on(Pattern separator) {
    // calls on(JdkPattern) }
  public static Splitter on(JdkPattern separator {
    ... }
}
```

```
// After update
class Splitter!2 {
  public static Splitter on(Regex separator) {
    // calls on(GuavaPattern) }
  public static Splitter on(GuavaPattern separator) {
    ... }
}
```

Listing 6. Class `Splitter` before and after update

to implement a working program using two versions of conflicting class definitions.

The rewritten Guava library consists of the static method `readlines` found in the class `Files` in package `com.google.common.io` and the class `Splitter` in package `com.google.common.base`. The backward incompatible changes were added into the class `Splitter`. To help introduce variation, we also rewrote the regular expression library JavaVerbalExpressions[3] in BatakJava. Both Guava and the regular expression libraries use several JDK classes, including `List` and `Pattern`.

Guava Library Update. A snippet of the definition of class `Splitter` is shown in Listing 6. In the original implementation, the class `Splitter` is instantiated through the static method `on` that accepts a Java's `Pattern` and calls a different method `on` that takes a Guava class `JdkPattern`. In the updated definition, to simulate backward incompatiblity, we remove both the original methods. The first method's parameter is replaced by `Regex` and the second method's parameter is replaced by `GuavaPattern`. `JdkPattern`, `Regex`, and `GuavaPattern` are also BatakJava classes with versions.

File Processing Program. The main program in Listing 7 mixes the use of both versions of `Splitter`. The lines processed by `readlines` are then split based on a regular expression. `splitter` and `newSplitter` use the method `on` with argument `Pattern` (`Splitter!1`) and `Regex` (`Splitter!2`) respectively. The compilation result is shown in Listing 8 where each `Splitter` is correctly inferred according to the method invoked.

One significant limitation of BatakJava is that the current inference does not scale linearly, so there will be issues in applying this prototype naively to large codebases. For this case study involving 41 class definitions and around 1200

[2] https://github.com/google/guava

[3] https://github.com/VerbalExpressions/JavaVerbalExpressions

```
// BatakJava program
public static void main(String[] args) {
  List<String> lines = new Files()
    .readLines(args[0], Charsets.UTF8);
  Pattern pattern = Pattern.compile("@");
  Splitter splitter = Splitter.on(pattern);
  Regex newPattern = new Regex(...);
  Splitter newSplitter = Splitter.on(newPattern);
  for (String line: lines) {
    splitter.split(line);
    newSplitter.split(line);
}}
```

Listing 7. `main` method using both versions of `Splitter`

```
// Java program
public static void main(String[] args) {
  List<String> lines = Files_ver_1
    .readLines(args[0], Charsets_ver_1.UTF8);
  Pattern pattern = Pattern.compile("@");
  Splitter_ver_1 splitter
    = Splitter_ver_1.on(pattern);
  Regex_ver_1 newPattern = new Regex_ver_1(...);
  Splitter_ver_2 newSplitter
    = Splitter_ver_2.on(newPattern);
  for (String line: lines) {
    splitter.split(line);
    newSplitter.split(line);
}}
```

Listing 8. `main` method after compilation

LOC, the compilation runs 2.1 times longer than the base ExtendJ compilation.

6 Related Work

6.1 Version-Aware Programming Languages

Versioned Featherweight Java (VFJ) [6, 7] takes a similar approach to BatakJava by introducing versions into the language. However, VFJ treats versions as a first-class citizen in a functional OOP language. While BatakJava incorporates versions as elements of types, VFJ defines classes within the context of version. It focuses on ensuring forward and backward compatibility of programs.

Unison[4] implements the idea of content-addressed code to identify code not by their names, but by their hashes, through the Unison codebase manager. This is analogous to declaring a new version of a definition in BatakJava. However, Unison only creates different hashes for different definitions where both changes in the signature and body are uniformly treated as differences. Unison and BatakJava are not different with respect their inability to detect semantic changes.

6.2 Programming Paradigms

Context-oriented programming [13] (COP) focuses on modularizing behavioral variations of an object. The behavior of an object is adapted dynamically following the runtime context. This paradigm treats context explicitly in the form of *layer*, in which the behavior of a class can be modified. The concept has been implemented as extensions to Java [2], Squeak/Smalltalk. Our proposal can be interpreted as specializing context as version, although, unlike COP, BatakJava resolves the dependency during compile time.

Variational programming [8], based on the choice calculus [10], is a functional programming paradigm that allows variations in the form of *choices* made up of left and right alternative that supports pattern matching. From the perspective of versions, choices can be interpreted as an object having different values in different versions.

Different versions of a class can be interpreted as a *family* of class [9, 15]. Class updates can be encoded as class inheritances. However, the family polymorphism approach is restricted in the way that it cannot remove existing definitions in further updates.

6.3 Package Manager Supports

Some package ecosystems such as npm[5] for JavaScript, Cargo[6] for Rust and Maven with its shade plugin[7] include multiple versions of the same definition through the use of name mangling. However, these methods are not incorporated into the language's semantics itself, resulting in wrongly identifying the same package as two completely different packages.

6.4 Software Product Line

The introduction of version into the language allows software's components to be broken down and its variations to be analyzed separately. A similar approach is taken in software product line [25] technologies. Implementations have been introduced, such as feature-oriented and delta-oriented programming [31].

Feature-oriented programming [26] introduced *features*, with a formalization in Feature Featherweight Java [1]. Features can be seen as abstract subclasses or mixins. Objects can be created by selecting desired features, allowing flexibility and promoting code reuse. Delta-oriented programming [29] works similarly as feature-oriented programming, with a core module modifiable by delta modules that may refine or remove existing code.

[4] https://www.unisonweb.org/

[5] https://www.npmjs.com/

[6] https://doc.rust-lang.org/cargo/

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

7 Future Work and Conclusion

7.1 Comprehensive Evaluation

We need to conduct a more thorough evaluation of our prototype to assess its viability. An empirical evaluation with larger codebases will be conducted to measure the limitation of the current compilation scheme. We also intend to conduct a use case experiment with programmers to measure the usability of BatakJava in software evolution as well as comparing it with already existing methods such as name mangling.

7.2 Improving Compilation

The current version inference generates constraints exponentially in regard to version numbers. Although BatakJava works in smaller codebases, such as the case study introduced in this paper, it does not scale with larger codebases. First, BatakJava has to enable separate compilation to minimize the burden of global version inference. Second, BatakJava requires high-level version management that allows users to better specify the version requirements of a program. By doing so, the number of versions included during version inference can be minimized.

Another issue with the current compilation scheme is that it generates many code clones. Creating a new version of a class without any change from the previous version may generate two classes with identical implementations but different naming. A possible approach to allow code reuse is by extracting common parts from different versions of a class and collecting them in a common ancestor.

7.3 Version Polymorphism

The absence of inheritance version polymorphism restricts the flexibility in which the users can use multi-version environment. Extending the calculus with version polymorphism and proving its type safety are left as future work. This extension can be approached by assigning a unique variable for each class annotation in the program, disallowing the possibility of creating definitions that can assume more than one version at the same time. A possible apporach to version polymorphism is by extending the current calculus with let polymorphism [20] on version numbers.

7.4 Semantic Versioning Support

Other approaches to version management, such as semantic versioning, can also be incorporated into programming with versions. Currently, a version is only an integer number that does not hold meaning on its own. Incorporating the idea of semantic versioning into BatakJava can be useful because the typing system then can statically check whether the version assigned to the program is appropriate or not.

7.5 Conclusion

Previous work on introducing versions as part of the language semantics first introduced in λVL has shown potential in handling dependency issues rooted in backward incompatible changes. However, its system lacks flexible support for evolving data structures.

This paper aims to expand the support of version programming to allow flexible definitions of data structures. We formalized the idea in the calculus FBJ and a prototype language BatakJava. The inclusion of versions in BatakJava allows class declarations to be explicitly annotated with versions, allowing multiple versions of a class to be included in a program. BatakJava does not put any special restrictions on how a class of different versions can be declared.

Acknowledgments

We would like to thank our fellow lab members at Tokyo Institute of Technology for the advice and discussions during research for this paper. This work was supported by JSPS KAKENHI grant numbers 18H03219.

References

[1] Sven Apel, Christian Kästner, and Christian Lengauer. 2008. 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*. 101–112.

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. 2011. ContextJ: Context-oriented programming with Java. *Information and Media Technologies* 6, 2 (2011), 399–419.

[3] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 109–120.

[4] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and how to make breaking changes. *ACM Trans. Softw. Eng. Methodol* 1, 1 (2021), 10–1145.

[5] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. Why and how Java developers break APIs. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 255–265.

[6] Luís Carvalho and João Costa Seco. 2019. Software Evolution with a Typeful Version Control System. In *International Conference on Software Engineering and Formal Methods*. Springer, 145–161.

[7] Luís Carvalho and João Costa Seco. 2021. Deep Semantic Versioning for Evolution and Variability. In *23rd International Symposium on Principles and Practice of Declarative Programming*. 1–13.

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

[9] Erik Ernst. 2001. Family polymorphism. In *European Conference on Object-Oriented Programming*. Springer, 303–326.

[10] Martin Erwig and Eric Walkingshaw. 2011. The choice calculus: A representation for software variation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, 1 (2011), 1–27.

[11] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient static checking of library updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering. 791–796.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java language specification*. Addison-Wesley Professional.

[13] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-oriented programming. *Journal of Object Technology* 7, 3 (2008), 125–151. https://doi.org/10.5381/jot.2008.7.3.a4

[14] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS* 23, 3 (2001), 396–450.

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

[16] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2020. Understanding type changes in java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 629–641.

[17] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empir Software Eng* 23, 1 (Feb. 2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5

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

[19] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 106–117.

[20] Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375.

[21] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 84–94.

[22] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. 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*. 215–225.

[23] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 439–451.

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

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

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

[27] Tom Preston-Werner. 2013. *Semantic Versioning*. https://semver.org/

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

[29] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-oriented programming of software product lines. In *International Conference on Software Product Lines*. Springer, 77–91.

[30] Yudai Tanabe, Luthfan Anshar Lubis, Tomoyuki Aotani, and Hidehiko Masuhara. 2021. A Functional Programming Language with Versions. *arXiv preprint arXiv:2107.07301* (2021).

[31] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 1–45.

Received 2022-08-08; accepted 2022-09-30